



POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

EXECUTIVE SUMMARY OF THE THESIS

Efficient Self-Adaptation using Model-Agnostic Interpretable Machine Learning

LAUREA MAGISTRALE IN SOFTWARE ENGINEERING - COMPUTER SCIENCE AND ENGINEERING

Author: PAOLO CORSA

Advisor: PROF. MATTEO CAMILLI

Co-advisor: PROF. RAFFAELA MIRANDOLA

Academic year: 2023-2024

1. Introduction

The increasing complexity of autonomous systems, especially self-adaptive, is affected by machine learning (ML) because of their adapting behavior based on external data, often resulting in “black-box” models that are difficult to interpret. This lack of transparency interferes with decision-making and with costs for engineers. To overcome this challenge, this work presents eXplanation Driven self-Adaptation v2 (XDAv2), an extension of a previous approach presented by Negri, Nicolosi, et al. called eXplanation Drive self-Adaptation (XDA) [6]. XDAv2 incorporates model-agnostic and interpretable ML techniques to enhance the explainability of the adaptation process, making it easier for engineers to understand and manage system behavior. The approach is integrated within the MAPE-K loop, a feedback system for self-adaptive behavior [7], and introduces two key components: an offline pre-processor to establish system knowledge, that is extended with a new feature ranking module using SHAP and permutation feature importance [4, 5] is introduced to identify and prioritize the most critical features for system performance and an online white-box optimizer to select the best adaptations based

on the environment using Partial Dependence Plots (PDPs) [5]. XDAv2 is evaluated through empirical evaluations using three case studies (robotics, autonomous flight, and autonomous driving) and compared with different baselines demonstrating its applicability and adaptability in diverse contexts.

2. Methods

This section explores the challenges of managing the environment and state of an autonomous system in the context of multi-objective optimization and the proposed approach used to address these challenges. The environment is represented by Observable Features (OF), such as weather or road conditions, which are outside the system’s control. In contrast, the Controllable Features (CFs) represent the aspects the system can manipulate to meet its goals. Together with the system’s requirements, these elements form the system’s operating condition at any given time.

Due to the multi-objective nature of the problem, where the system must satisfy multiple requirements simultaneously, NSGA-III and Fitest were selected to have solid baselines. NSGA-III is a generative algorithm and is designed for

multi-objective optimization [2]. At the same time, FITEST generates ideal test cases that cover various objectives, particularly in identifying undesired interactions between system features [1].

2.1. Offline Pre-processor

2.1.1 Algorithm for PDP and SPDP Generation

The offline pre-processor builds on the set of trained classifiers M by generating Partial Dependence Plots (PDPs) that reveal how each feature affects the prediction of the classifiers. For each feature CF_i and requirement M_j , a partial dependence plot PDP_{ij} is created, displaying how different values of CF_i influence the satisfaction of M_j . Using a scoring function, each feature CF_i is evaluated across models, where the score is calculated by multiplying individual conditional expectation curves across models (Algorithm 1).

Algorithm 1: Generation of PDP and SPDP

Input: Set of classifiers
 $M = \{M_1, \dots, M_n\}$, Set of controllable features
 $CF = \{CF_1, \dots, CF_m\}$

Output: Set of PDPs $PDP = \{PDP_{ij} \forall i = 1..n, j = 1..m\}$,
Set of m summary PDPs
 $SPDP = \{SPDP_1, \dots, SPDP_m\}$

- 1 Set $PDP \leftarrow \emptyset$
- 2 Set $SPDP \leftarrow \emptyset$
- 3 **for** $i \leftarrow 1$ to n **do**
- 4 **for** $j \leftarrow 1$ to m **do**
- 5 $PDP_{ij} \leftarrow \text{buildPDP}(M_i, CF_j)$
- 6 $PDP \leftarrow \text{updateSet}(PDP, PDP_{ij})$
- 7 **end**
- 8 **end**
- 9 **for** $j \leftarrow 1$ to m **do**
- 10 $score_j \leftarrow \text{buildScoreFunction}(PDP_{1j}, \dots, PDP_{nj})$
- 11 $SPDP \leftarrow \text{updateSet}(SPDP, score_j)$
- 11 **end**

2.1.2 Feature Ranking with SHAP and Permutation Importance

Two algorithms, SHAP, and permutation feature importance rank controllable features CF . SHAP values provide a global view of feature importance across models M , ordering CFI by cumulative SHAP value for each feature. The SHAP ranking algorithm averages SHAP values per classifier, accumulating them for a global ranking (Algorithm 2).

Algorithm 2: Approximate Shapley Estimation for Single Feature Value

Input: Number of iterations M , instance of interest x , feature index j , data matrix X , and machine learning model f

Output: Shapley value for the value of the j -th feature

- 1 **for** $m \leftarrow 1$ to M **do**
- 2 Draw random instance z from the data matrix X
- 3 Choose a random permutation o of the feature values
- 4 Order instance x :
 $x_o = (x(1), \dots, x(j), \dots, x(p))$
- 5 Order instance z :
 $z_o = (z(1), \dots, z(j), \dots, z(p))$
- 6 Construct two new instances:
- 7 With j : $x_j^+ = (x(1), \dots, x(j-1), x(j), z(j+1), \dots, z(p))$
- 8 Without j : $x_j^- = (x(1), \dots, x(j-1), z(j), z(j+1), \dots, z(p))$
- 9 Compute marginal contribution:
 $\phi_j^m = \hat{f}(x_j^+) - \hat{f}(x_j^-)$
- 10 **end**
- 11 **return** Shapley value as the average:
 $\phi_j(x) = \frac{1}{M} \sum_{m=1}^M \phi_j^m$

Permutation importance sums the averages of feature impacts across requirements, also re-ordering CFI by overall importance (Algorithm 3).

2.2. Online White-box Optimizer

The online optimizer activates if a requirement's satisfaction is below a threshold P . It records the current state (\hat{CF}, \hat{OF}) and generates new values \hat{CF}^* based on offline pre-processor data, aiming to achieve a true class label probability

Algorithm 3: Estimate Feature Importance Using Permutation

Input: Trained model \hat{f} , feature matrix X , target vector y , error measure $L(y, \hat{f})$

Output: Sorted feature importance scores

```

1 Estimate the original model error:
    $e_{\text{orig}} = L(y, \hat{f}(X))$ 
2 for each feature  $j \in \{1, \dots, p\}$  do
3   Generate feature matrix  $X_{\text{perm}}$  by
   permuting feature  $j$  in the data  $X$ 
4   Estimate error:
    $e_{\text{perm}} = L(y, \hat{f}(X_{\text{perm}}))$ 
5   Calculate permutation feature
   importance:
   • As quotient:  $FI_j = \frac{e_{\text{perm}}}{e_{\text{orig}}}$ 
   • Or as difference:  $FI_j = e_{\text{perm}} - e_{\text{orig}}$ 
6 end
7 return Sorted features by descending  $FI$ 

```

above P .

Climbing Step This initial adaptation finds k neighbors of $(\hat{C}F, \hat{O}F)$ in the training set D , adjusting controllable features to maximize requirement satisfaction. The algorithm identifies the feature CF_h that maximizes the satisfaction score and modifies it accordingly.

Descending Step This refinement of adaptation aims to improve satisfaction probabilities while maintaining P over a certain threshold for all requirements, adjusting values (adding or subtracting a value Δ depending on optimization direction) in the order of importance as determined by SHAP and permutation importance.

2.3. Profiling and Optimization

Using Python’s `cProfile` to collect data on the execution of the code, the main adaptation function (`findAdaptation`) was optimized by consolidating multiple feature updates within each call. Additionally, `tracemalloc` was used to monitor peak memory usage, enabling a visual comparison of memory consumption across tests, which helped identify and address memory inefficiencies.

This process optimizes both time and memory usage for adaptation, balancing computational load with the need for accurate feature optimization in real-time scenarios.

3. Evaluations

Three research questions were selected to make evaluations of the accuracy and efficiency of XDAv2:

RQ1: What is the effectiveness of XDA compared to our selected baselines in terms of predicted and actual satisfaction of requirements?

RQ2: What is the cost of the Online white-box optimizer in terms of memory and time?

RQ3: What is the cost of the Offline pre-processor in terms of memory and time?

RQ1 The evaluation began by assigning 200 initial values across the variable space for each system, representing an operational scenario that led to at least one unmet requirement. Two primary metrics were used to analyze the effectiveness of different optimization algorithms in addressing RQ1.

1. Satisfaction Likelihood: For each requirement R_i , given a specific configuration of observable features $\hat{O}F$ and a chosen adaptation $\hat{C}F^*$, this metric quantifies the probability (from the black-box classifier M_i) that the requirement is met under the given conditions:

$$P_{M_i}(y = \text{true} \mid (\hat{C}F^*, \hat{O}F)).$$

2. Success Rate: This metric represents the proportion of scenarios in which a requirement R_i is met under simulated conditions. It is calculated as the number of instances in which $(\hat{C}F^*, \hat{O}F)$ satisfies R_i over the total number of conditions X .

The success rate focuses on observed simulation outcomes, providing a grounded, empirical measure of algorithm effectiveness. In this analysis, **XDAv2** and **XDA** rely on a defined neighborhood size k (set to 10) for finding similar configurations. This value balances effectiveness and computational cost, with Faiss library optimizations applied exclusively to XDAv2 for efficient nearest-neighbor searches [3], thus reducing time and memory usage.

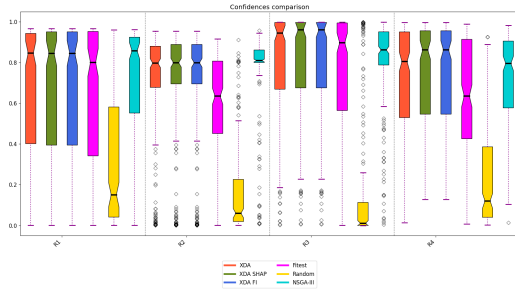


Figure 1: Satisfaction likelihood of rescue robot requirements

Multiple multi-objective optimization algorithms were tested for a consistent comparison, each fine-tuned to achieve comparable baseline performance. Key settings include:

- **NSGA-III**: Population size set to match reference directions, a constraint tolerance of 10^{-6} , and a 100-generation limit.
- **FITEST**: Tuned to match the population size of NSGA-III, the number of objectives, and a minimum satisfaction probability threshold of 0.8 for each requirement R_i .

For the rescue robot XDAv2 shows the highest average confidence in meeting two out of four requirements, while the other two requirements show comparable results across algorithms (Figure 1).

For the UAV XDAv2 demonstrates confidence levels comparable to other algorithms across all requirements. For the autonomous vehicle results mirror the UAV case, with XDAv2 maintaining similar confidence levels across requirements.

The success rates for each algorithm across different systems (rescue robot, UAV, and autonomous driving) show that **XDAv2** consistently achieves equal or greater success rates than FITEST and NSGA-III, though it is typically equal to or slightly lower than XDA base.

RQ2 To address RQ2, the costs of the online white-box optimizer were evaluated, collecting data for all 200 tests and comparing it with the execution times of XDA, NSGA-III, Fitest, and Random.

Execution time was measured using Python’s `time` package, which records the start and end times for each test and calculates total execution

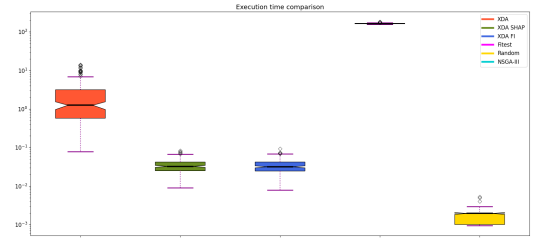


Figure 2: Execution times of rescue robot with double the number of controllable features

time as:

Metric 3: $time_{total} = time_{end} - time_{start}$ (seconds).

Two test setups were selected for comparison: the first used the system’s original number of controllable features, while the second doubled this count by converting some observable features to controllable ones. This allowed for analyzing the effect of increased workload.

In the standard configuration, XDAv2 (SHAP and FI) showed an initial speed-up over base XDA, as well as a more substantial speed-up compared to FITEST and NSGA-III. When controllable features were doubled (e.g., rescue robot double, UAV double), the advantage of XDAv2 SHAP and FI became even more pronounced. For the rescue robot double scenario, NSGA-III was excluded due to excessive memory requirements (approximately 18.9 GB), exceeding the 8 GB available on the testing system. Across all tests, XDAv2 achieved execution times on the order of 10^{-2} seconds in the standard setup, while XDA averaged around 10^{-1} seconds. FITEST and NSGA-III, in contrast, operated on a second-scale range for standard controllable feature counts (4 for rescue robot, 3 for UAV, and 1 for autonomous driving). In cases where controllable features doubled (8 for rescue robot, 6 for UAV, and 2 for autonomous driving), XDAv2 maintained times around 10^{-1} seconds, while XDA remained under one minute, and FITEST and NSGA-III extended to minute-long execution times (Figure 2).

Percentage speed-ups are used to calculate the reduction in execution time relative to the baseline achieved by XDAv2 SHAP and FI. Positive values indicate improved performance, while negative values denote slower performance.

The results indicate a clear increase in speed-up for XDAv2 SHAP and FI as controllable feature count and requirements increase. For example, in the UAV and UAV double scenarios (with 3 controllable features and 12 requirements, and 6 controllable features and 12 requirements, respectively), XDAv2 SHAP’s speed-up over XDA increased from 284.30% to 15,961.63%. Similarly, in the rescue robot and rescue robot double tests (with 4 controllable features and 4 requirements, and 8 controllable features and 4 requirements), speed-up rose from 2,953.33% to 7,194.45%.

Memory usage was assessed using Python’s `tracemalloc` package, which recorded peak memory allocation for each test.

Metric 4: Peak memory allocated by the algorithm during adaptation search (megabytes). Percentage difference of memory allocation is used to compare XDAv2 SHAP and FI to XDA, FITEST, Random, and NSGA-III. Negative values indicate lower memory usage for XDAv2, while positive values indicate greater usage. Generally, XDAv2 SHAP and FI displayed memory efficiency, with negative percentages in most cases. For instance, in the rescue robot scenario, both XDAv2 methods achieved an 89% reduction in memory allocation compared to other algorithms, and similar efficiencies were observed in the UAV and autonomous driving cases. The exception was the rescue robot double scenario, where XDAv2 SHAP and FI used 9.73% more memory. Although FITEST and NSGA-III achieved memory savings, NSGA-III was infeasible for the "Robot Double" scenario due to excessive memory demands exceeding system capacity. The Random algorithm used the least memory, requiring only a single array to randomly adjust controllable feature values.

RQ3 Following the profiling of the online white-box optimizer, the next step involved profiling the offline pre-processor, responsible for building the knowledge base by creating the PDPs and SPDPs, and performing feature ranking using SHAP and permutation feature importance. This profiling aimed to observe the memory and time costs at system deployment, assessing whether this phase is resource-intensive.

To obtain robust data for analysis, the construction of PDP and SPDP was repeated 20 times,

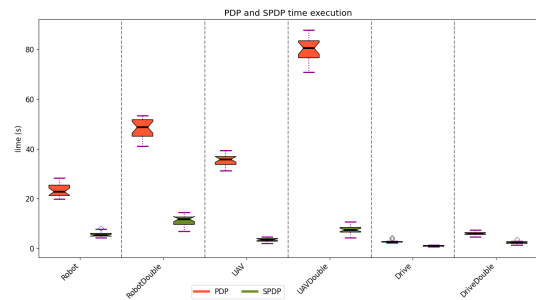


Figure 3: Execution times PDPs and SPDPs construction

creating multiple profiling instances for both execution time and peak memory usage.

In the offline phase, immediately after training the models to meet the requirements, the models are passed to a function that constructs a PDP and an SPDP for each requirement and each controllable feature. Both execution time and peak memory allocation were recorded for each construction.

Metric 5: Execution time for constructing PDPs and SPDPs for each machine learning model (seconds).

As the number of features and requirements increases (e.g., in UAVDouble), the execution time also rises, potentially causing delays during system deployment, particularly in complex systems. This phase is critical, as it demonstrates how feature variations affect the fulfillment of requirements (Figure 3).

Metric 6: Peak memory allocated during the creation of PDPs and SPDPs for each machine learning model (megabytes).

Following the construction of PDPs and SPDPs, the feature ranking phase starts, using controllable features and models representing the requirements. For each requirement, both SHAP and permutation feature importance rank each controllable feature, and the execution time and peak memory usage were recorded for each requirement.

Metric 7: Execution time of SHAP and permutation feature importance ranking for each machine learning model (seconds).

SHAP incurs a significantly higher execution time compared to permutation feature importance for the Logistic Regression and Neural Network models while remaining comparable for other models (Figure 4).

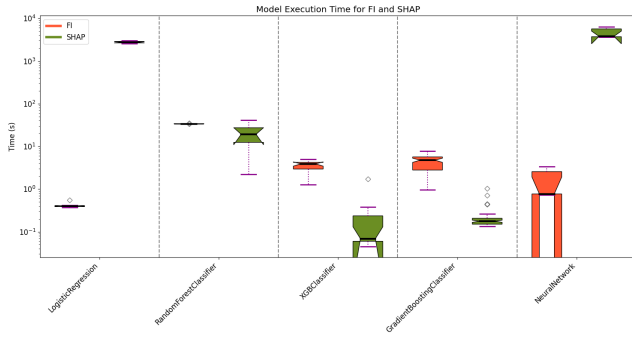


Figure 4: Execution times SHAP and permutation feature importance

Metric 8: Peak memory allocated by SHAP and permutation feature importance during feature ranking for each machine learning model (megabytes).

Memory usage patterns mirror previous results, with SHAP showing a pronounced peak in memory usage for the Logistic Regression and Neural Network model while being comparable with other models.

4. Conclusions

This thesis introduces XDAv2, an enhanced version of the XDA approach that integrates model-agnostic interpretable machine learning within a MAPE-K control loop. XDAv2 improves the interpretability and explainability of system models through two key components: an offline pre-processor and an online white-box optimizer. XDAv2 includes a feature ranking module in the offline pre-processor, utilizing SHAP and permutation feature importance algorithms to identify the most influential features on system requirements.

To assess the performance of XDAv2, 18 empirical experiments were conducted, each comprising 200 tests across three experimental scenarios: rescue robot, UAV, and autonomous driving. For each scenario, constraints were progressively increased in three levels (no label, v2, and v3), with additional tests conducted by doubling the controllable features.

The results demonstrated that XDAv2 is highly effective, outperforming XDA and baseline methods (e.g., FITEST, NSGA-III). The execution time of the offline pre-processor is minimal, except for SHAP, which caused delays in some models. XDAv2 showed impressive

speedups, achieving up to 18,776.81% over XDA, 169,933.72% over FITEST, and 26,096.41% over NSGA-III. Additionally, memory usage was reduced by 60.1% compared to XDA, 99.04% compared to FITEST, and 86.63% compared to NSGA-III.

XDAv2 stands out for its time and memory efficiency. Average execution times remained under 10^{-1} seconds (or 10^{-2} seconds for tests with deployment-level features), and memory savings were significant compared to the baselines. XDAv2’s scalability was demonstrated through its performance with more complex models, where execution time increases were less pronounced compared to XDA. For instance, in the rescue robot scenario, XDAv2’s execution time increased by only 3.84%, while XDA’s increased by 9.15%. Similar improvements were observed for the UAV and autonomous driving scenarios.

While the offline pre-processor introduces overhead during deployment, this can be managed, as XDAv2 maintains efficiency even with increased system complexity.

References

- [1] Raja Ben Abdesslem, Annibale Panichella, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. Testing autonomous cars for feature interaction failures using many-objective search. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE '18*, page 143–154, New York, NY, USA, 2018. Association for Computing Machinery.
- [2] Kalyanmoy Deb and Himanshu Jain. An evolutionary many-objective optimization algorithm using reference-point-based non-dominated sorting approach, part i: Solving problems with box constraints. *IEEE Transactions on Evolutionary Computation*, 18(4):577–601, 2014.
- [3] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library, 2024.
- [4] Aaron Fisher, Cynthia Rudin, and Francesca Dominici. All models are wrong, but many

are useful: Learning a variable's importance by studying an entire class of prediction models simultaneously, 2019.

- [5] Christoph Molnar. *Interpretable Machine Learning*. BOOKDOWN, 2024.
- [6] Francesco Renato Negri, Niccolo Nicolosi, Matteo Camilli, and Raffaella Mirandola. Explanation-driven self-adaptation using model-agnostic interpretable machine learning. In *Proceedings of the 19th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '24*, page 189–199, New York, NY, USA, 2024. Association for Computing Machinery.
- [7] Danny Weyns. Engineering self-adaptive software systems – an organized tour. In *2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pages 1–2, 2018.