



**POLITECNICO**  
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE  
E DELL'INFORMAZIONE

# Efficient Self-Adaptation using Model-Agnostic Interpretable Ma- chine Learning

TESI DI LAUREA MAGISTRALE IN  
SOFTWARE ENGINEERING - COMPUTER SCIENCE AND ENGI-  
NEERING

Author: **Paolo Corsa**

Student ID: 102841

Advisor: Prof. Matteo Camilli

Co-advisors: Prof. Raffaella Mirandola

Academic Year: 2023-24



# Abstract

Self-adaptive systems are increasingly incorporating black-box predictive models, such as neural networks, to guide decision-making and adaptation strategies. However, the opacity of these models complicates efforts to clarify the reasoning behind adaptation choices and assess their potential impacts on the environment. Moreover, these adaptation decisions typically result from resource-intensive optimization procedures that navigate vast search spaces while dealing with multiple, often conflicting objectives. The complexity here derives from the challenge of not being able to directly interpret or fully understand the internal workings of black-box models, often necessitating iterative approaches to achieve satisfactory results.

To mitigate these challenges, this thesis introduces an explanation-driven approach to self-adaptation that leverages model-agnostic interpretable machine learning techniques, specifically feature importance ranking within the system's feedback loop. Feature importance ranking, an interpretability method, identifies the key factors influencing adaptation choices, offering deeper insight into the model's decision-making process. By improving the transparency of predictive models, this technique supports more informed and efficient adaptations, leading to a significant reduction in planning costs. The empirical evaluation, using three case studies (robotics, autonomous flying, and autonomous driving), highlights the cost-effectiveness of our approach in managing the trade-offs between adaptation performance and optimization efficiency.

**Keywords:** Explainable Self-adaptation, Model-agnostic explanations, Interpretable Machine Learning, Feature ranking



# Abstract in lingua italiana

I sistemi auto-adattivi stanno incorporando sempre più modelli predittivi black-box, come le reti neurali, per guidare le decisioni e le strategie di adattamento. Tuttavia, l'opacità di questi modelli complica gli sforzi per chiarire il ragionamento alla base delle scelte di adattamento e valutare i loro potenziali impatti sull'ambiente. Inoltre, queste decisioni di adattamento derivano tipicamente da procedure di ottimizzazione ad alta intensità di risorse che navigano in spazi di ricerca vasti, gestendo al contempo obiettivi multipli e spesso in conflitto. La complessità qui deriva dalla sfida di non poter interpretare direttamente o comprendere appieno il funzionamento interno dei modelli black-box, il che richiede spesso approcci iterativi per ottenere risultati soddisfacenti.

Per mitigare queste sfide, questa tesi introduce un approccio incentrato sull'esplicabilità e sull'auto-adattamento che sfrutta tecniche di apprendimento automatico interpretabile e tecniche agnostiche rispetto ai modelli, in particolare il ranking dell'importanza delle caratteristiche del sistema. Il ranking identifica i fattori chiave che influenzano le scelte di adattamento, offrendo una comprensione più profonda del processo decisionale del modello. Migliorando la trasparenza dei modelli predittivi, questa tecnica supporta adattamenti più precisi ed efficienti, portando a una significativa riduzione dei costi di pianificazione. La valutazione empirica, attraverso tre casi di studio (robotica, volo autonomo e guida autonoma), evidenzia la convenienza di questo approccio nella gestione dei compromessi tra prestazioni di adattamento ed efficienza di ottimizzazione.

**Parole Chiave:** Adattamento autonomo spiegabile, Spiegazioni indipendenti dal modello, Apprendimento automatico interpretabile, Classifica delle caratteristiche



# Contents

<b>Abstract</b>	<b>i</b>
<b>Abstract in lingua italiana</b>	<b>iii</b>
<b>Contents</b>	<b>v</b>
<b>Introduction</b>	<b>1</b>
<b>1 State of the art</b>	<b>5</b>
1.1 Self-Adaptive Systems . . . . .	5
1.2 Multi-Objective Optimization (MOO) . . . . .	7
1.3 Supervised Machine Learning . . . . .	8
1.4 Interpretable Machine Learning . . . . .	10
1.4.1 Approaches to Interpretable Machine Learning . . . . .	10
1.4.2 Partial Dependence Plot (PDP) . . . . .	11
1.5 SHAP and Shapley values . . . . .	12
1.6 Permutation Feature Importance . . . . .	13
1.7 NSGA-III . . . . .	14
1.8 FITEST . . . . .	15
1.9 Faiss . . . . .	16
<b>2 Proposed methods</b>	<b>19</b>
2.1 Problem Statement . . . . .	19
2.2 Explanation-Driven Approach . . . . .	20
2.2.1 Offline Pre-processor . . . . .	22
2.2.2 Online White-box Optimizer . . . . .	27
2.2.3 Profiling . . . . .	30
<b>3 Experimentation</b>	<b>31</b>
3.1 Results . . . . .	35

<b>4 Conclusions</b>	<b>57</b>
<b>Bibliography</b>	<b>59</b>
<b>List of Figures</b>	<b>61</b>
<b>List of Tables</b>	<b>63</b>
<b>List Of Symbols</b>	<b>65</b>
<b>Acknowledgements</b>	<b>67</b>

# Introduction

Autonomous systems, specifically self-adaptive systems, are designed to collect data from the external environment and use it to adjust their behavior in response to changing situations. For this reason, in recent years, engineers have introduced machine learning in these systems to meet these demands more efficiently and accurately. Once properly trained, machine learning models can classify the behaviors of self-adaptive systems according to specific requirements that must be met within a given environment. However, this approach has led to more opaque systems, often considered "black-box" due to the inherent complexity of machine learning models, particularly Neural Networks (NNs) and Deep Neural Networks (DNNs). This complexity makes it more challenging to understand the inner workings of these systems, resulting in higher costs for engineers and staff when making informed decisions.

To address these challenges, this thesis presents eXplanation Driven self-Adaptation v2 (XDv2), an extension of a novel approach introduced by Negri, Nicolosi et al. [13], called eXplanation Driven self-Adaptation (XDA). This approach also leverages machine learning, specifically model-agnostic and interpretable machine learning, to reduce system opacity and enhance explainability. These methods are applied within the MAPE-K Loop, a feedback loop that integrates all elements necessary for self-adaptive behavior. This enables the system to collect data from the external environment and make informed decisions in response [15].

XDv2 extends the MAPE-K Loop with two key components: the offline pre-processor, which operates at system deployment to establish knowledge that will be used to find increasingly precise adaptations, and the online white-box optimizer, which is the core element that calculates, classifies, and selects the most appropriate adaptations for the presented situation. The offline pre-processor has been enhanced with a new feature ranking module that allows for deeper insights into the system by identifying and ranking the features most critical to its functionality. Two algorithms, SHAP (SHapley Additive exPlanations) and permutation feature importance (FI) [12], are employed for this purpose.

These methods provide insights into the relative importance of input features on model predictions, thereby making the adaptation logic more transparent and interpretable. SHAP offers consistent, theoretically grounded feature importance scores based on cooperative game theory, while permutation feature importance assesses the effect of randomly shuffling each feature on model performance. This dual approach to feature ranking significantly enhances the system’s explainability, making it easier for engineers to understand the underlying adaptation decisions and fine-tune the system’s response mechanisms.

By integrating this extended feature-ranking module, XDAv2 achieves greater interpretability and reduces the computational cost of adaptation, as the system can focus on the most influential features, optimizing adaptation decisions more effectively. Additionally, the online white-box optimizer leverages the insights from feature importance to guide the search for adaptation options in a more targeted manner, improving both efficiency and goal satisfaction. XDAv2 is evaluated by an external module called *profiler* that collects time execution and memory allocation through empirical studies across three domains: robotics, autonomous flight, and autonomous driving. These domains differ substantially in their environments and feature sets, allowing for a broad and rigorous evaluation of XDAv2’s general applicability and adaptability across varied contexts.

In the three test domains mentioned above, three types of tests were conducted for each domain to tighten the constraints and evaluate the behavior of XDA v2 under more rigid conditions. In addition to these nine test types, empirical tests were also performed where the number of controllable features (which are those parameters that can be controlled and modified by the system) for each system was doubled to observe XDA v2’s performance with an increased workload. Therefore, a total of 18 tests were conducted: nine tests with progressively stricter constraints, and the same number of tests with the number of controllable features doubled.

This thesis is organized into four chapters:

- **Chapter 1 - State of the Art:** This chapter presents the foundational contributions of this thesis, reviewing the machine learning methods and algorithms that support the proposed approach.
- **Chapter 2 - Proposed Methods:** This chapter details the approach developed in this thesis, describing the methodology, including the XDAv2 architecture, components, and model-agnostic interpretability techniques.
- **Chapter 3 - Experimentation:** This chapter presents the testing and experimental evaluation of XDAv2, along with the analysis of results across the selected

domains.

- **Chapter 4 - Conclusions:** This final chapter discusses the overall findings and evaluations, summarizing the impact of XDAv2 on self-adaptive systems and outlining future research directions.



# 1 | State of the art

This section presents the key elements employed in this work to address the research questions. A general description of each element will be provided, along with its role within the overall framework of the study, highlighting its functions and applications. The elements presented are (in order of appearance within the document): Self-Adaptive Systems (Section 1.1 and subsections), Mutli-Objective Optimization (Section 1.2), Supervised Machine Learning (Section 1.3), Interpretable Machine Learning (Section 1.4), NSGA-III (Section 1.7), FITEST (Section 1.8), SHAP and Shapley Values (Section 1.5), Permutation Feature Importance (Section 1.6).

## 1.1. Self-Adaptive Systems

The concept of self-adaptive systems is generally understood as a system capable of adjusting its behavior in response to changes in its environment or within the system itself or as one that can autonomously adjust its behavior with minimal or no human interference [4]. These interpretations take the perspective of an external observer, focusing on the system's ability to handle external conditions, resources, workloads, and failures. In contrast, Garlan et al. and Andersson et al. consider self-adaptive systems from the engineer's perspective, emphasizing the use of closed feedback loops and the separation between "domain concerns" (the system's goals) and "adaptation concerns" (how the system achieves those goals under changing conditions) [4, 11].

A system is *self-adaptive* if it follows these two principles: [15]

- **External principle:** A self-adaptive system is a system that can handle changes and uncertainties in its environment, the system itself, and its goals autonomously (i.e., without or with minimal human interference).
- **Internal principle:** A self-adaptive system comprises two distinct parts: the first part interacts with the environment and is responsible for the domain concerns (i.e., concerns for which the system is built); the second part interacts with the first part (and monitors its environment) and is responsible for the adaptation concerns (i.e.,

concerns about the domain concerns).

An example of a self-adaptive system is the rescue robot, which is employed to manage critical situations during natural disasters and ensure the safe rescue of victims, either without or with very minimal human influence, thereby reducing potential risks. The operating environment for the rescue robot is rapidly changing, due to factors such as falling debris or fires.

Another system capable of autonomously adapting to critical scenarios is the UAV (Unmanned Aerial Vehicle). It can respond to a varying number of threats at different distances by adjusting its speed or activating/deactivating countermeasures accordingly.

Similarly, autonomous vehicles operate comparably. These systems enable self-driving within highly dynamic environments, such as roads with pedestrians and other vehicles. They can assess dangerous situations, like a pedestrian crossing the street, and respond accordingly to maintain safety.

In Figure 1.1 a model for self-adaptive systems is shown.

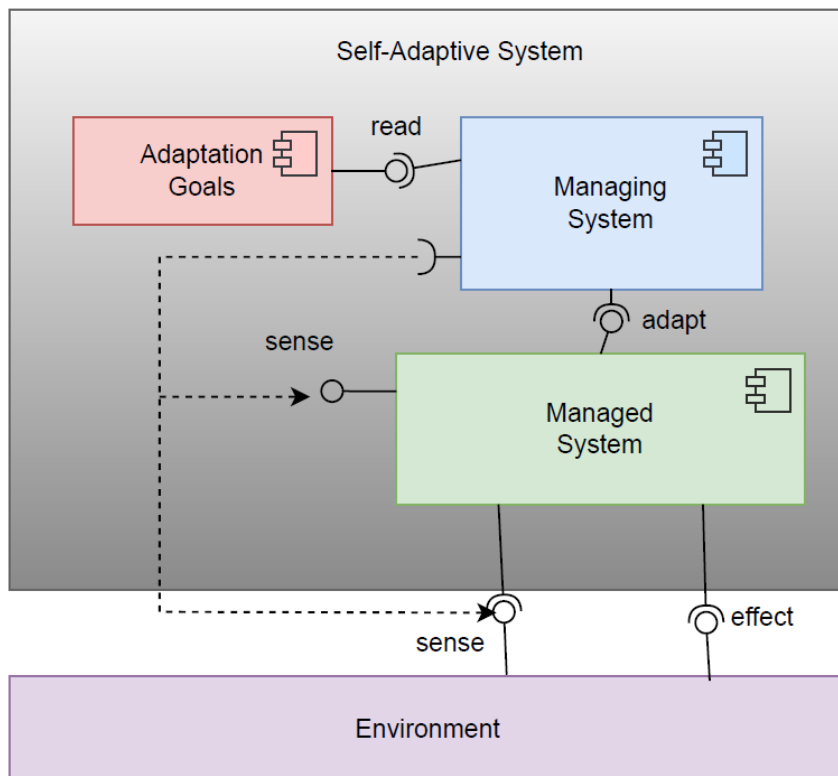


Figure 1.1: Self-adaptive systems model.

- **Environment** is the element from which the system collects information and upon

which the system's actions have an impact (e.g. site affected by an earthquake, a military area, or a street).

- **Managed System** is the system on which the application code is executed and which acts in the environment (rescue robot, UAV, autonomous vehicle).
- **Adaptation Goals** refer to the requirements that the system must fulfill during its operation. Four principal types of high-level adaptation goals can be identified: self-configuration (i.e., systems that automatically configure themselves), self-optimization (systems that continuously seek ways to improve their performance or reduce costs), self-healing (systems that detect, diagnose, and repair problems caused by bugs or failures), and self-protection (systems that defend themselves from malicious attacks or cascading failures).
- **Managing Systems** is responsible for overseeing the managed system using adaptation logic and strives to meet all imposed requirements, so it indicates to the managed systems how to act in the environment to satisfy adaptive goals. [15]

## 1.2. Multi-Objective Optimization (MOO)

One of the tasks of a self-adaptive system is precisely to meet certain requirements, such as maintaining a safe distance from the person to be rescued for the robot, avoiding direct collisions with threats for UAVs, or always keeping a safe distance from pedestrians for the autonomous vehicle. These are just a few examples; in fact, each system must simultaneously meet multiple requirements, which is why these systems need to be capable of modifying their behavior to find the best adaptation for a given situation.

Due to the complex nature of these problems, multi-objective optimization (MOO) has been adopted, a branch of optimization where the goal is to optimize several criteria simultaneously, rather than focusing on a single outcome.

Unlike single-objective optimization, where a clear optimal solution exists, multi-objective optimization often results in a set of solutions, known as the Pareto-optimal set. These solutions represent the best possible trade-offs between objectives; in other words, no solution in this set can improve one objective without worsening another. A solution is Pareto-optimal if there is no other solution that improves all objectives simultaneously. The collection of these solutions forms the Pareto front, which visually represents the trade-offs between conflicting objectives [5].

Multi-objective optimization is inherently challenging due to the conflicting nature of

objectives. Thus, the aim of MOO is not to find a single "best" solution but to identify a set of compromise solutions, allowing decision-makers to choose based on their preferences or priorities.

The complexity of these problems has driven the development of numerous algorithms and methods to effectively explore and solve multi-objective optimization problems. Over time, these methods have evolved from classical techniques like weighted sum approaches to more sophisticated strategies, such as evolutionary algorithms (NSGA-III [6], FITEST [1]).

### 1.3. Supervised Machine Learning

To accurately describe and model systems and their behaviors as effectively as possible, supervised machine learning is used. Supervised Machine Learning (ML) is a subset of machine learning where the goal is to train a model on labeled data to make predictions or decisions. In this paradigm, the algorithm learns from a dataset that contains both input features and the corresponding output labels, enabling the model to infer relationships between them. Once trained, the model can generalize from this training data to make predictions on unseen data.

Supervised machine learning follows a structured process that enables the development of predictive models from labeled datasets. This process can be broken down into several critical stages, each of which contributes to the overall success and reliability of the model. The key stages are *data collection*, *preprocessing*, *model selection*, and *evaluation*.

**Data Collection** The first and arguably most crucial step in any machine learning project is the collection of high-quality data. In supervised learning, the dataset must consist of both *input features* (also known as predictors or independent variables) and corresponding *output labels* (the target variable or dependent variable). These labeled examples allow the algorithm to learn the mapping from inputs to outputs.

Data can come from a variety of sources, including databases, sensors, web scraping, or pre-existing datasets. The quality and quantity of the data directly affect the model's performance, making it vital to ensure that the data is representative of the problem at hand.

In the case of the rescue robot, UAV, and autonomous vehicle, the training datasets are created to simulate logs during the operation of these systems. Each log represents a snapshot of the system's operating condition, which consists of the set of parameter

values that describe the system's behavior (independent and dependent variables) and whether the requirements are met or not under that condition.

**Data Preprocessing** Once data is collected, it rarely comes in a form suitable for immediate use. Preprocessing is the step where raw data is transformed into a format that can be fed into machine learning algorithms. This involves several tasks:

- **Feature Engineering:** This involves creating new features from existing data or selecting the most relevant features that contribute to better model performance (SHAP, permutation feature importance). Techniques like one-hot encoding for categorical variables, normalizing or standardizing numerical features, and dimensionality reduction (e.g., PCA) are often employed.
- **Data Splitting:** Typically, the dataset is split into a *training set* and a *test set*. The training set is used to train the model, while the test set is reserved for evaluating the model's generalization ability.

**Model Selection** The next step is to choose a suitable machine learning algorithm. The selection depends on the type of problem (classification or regression), the size and complexity of the data, and the desired performance criteria. Several algorithms are available for supervised learning, each with different strengths and weaknesses:

- **Linear Models** (e.g., Logistic Regression): These models are simple and interpretable, making them suitable for problems where the relationship between input and output is relatively straightforward.
- **Tree-Based Models** (e.g., C5.0 (Decision Trees), Random Forests, Gradient Boosting Machines, eXtreme Gradient Boosting tree): These models can handle complex, non-linear relationships and are relatively robust to noisy data. They also offer feature importance insights, aiding in model interpretability.
- **Neural Networks:** For complex tasks like image recognition or natural language processing, deep learning models, especially neural networks, have demonstrated state-of-the-art performance. However, they are more resource-intensive and harder to interpret.

**Model Evaluation** Once the model is trained, it must be evaluated to ensure it performs well not only on the training data but also on unseen data. This step involves testing the model on the *test set* (or validation set) that was held back during preprocessing. The metric used for the problem is: **ROC Curve and AUC**. The Receiver

Operating Characteristic (ROC) curve is a graphical plot used to assess the performance of a classification model, while the Area Under the Curve (AUC) provides a single scalar value summarizing its effectiveness [3].

## 1.4. Interpretable Machine Learning

Given the black-box nature of machine learning models (as discussed in the previous Section 1.3), **interpretable machine learning** has been adopted. This approach allows for the explanation and interpretation of important elements within certain black-box systems, making them clearer and more transparent. Among these, features from feature engineering were selected, such as SHAP, permutation feature importance, and partial dependence plots, which illustrate the effects of specific features on the behavior of the studied system. For example, additional insights are provided regarding the behavior of the robot after modifying its speed, the behavior of a UAV when flying in formation, or the changes in a vehicle's speed upon detecting a pedestrian.

### 1.4.1. Approaches to Interpretable Machine Learning

Two primary approaches to achieving interpretability in machine learning models are *intrinsic interpretability* and *post-hoc interpretability*.

**Intrinsic Interpretability** Intrinsic interpretability is inherent in simpler models, such as linear regression or decision trees, where the model structure allows for straightforward reasoning. These models are often referred to as white-box models, as the logic behind their decisions can be easily observed. While these models tend to be less complex, they are often limited in their ability to capture highly non-linear patterns in data, which may reduce their predictive performance in certain contexts.

**Post-hoc Interpretability** In contrast, post-hoc interpretability focuses on explaining the behavior of more complex, black-box models, such as deep neural networks or ensemble methods, after they have been trained. These models are typically more accurate but are also opaque, making it difficult to understand their decision-making processes directly. Post-hoc methods aim to bridge this gap, offering tools and techniques to demystify these complex systems. This thesis includes some of these techniques:

- **SHAP (Shapley Additive Explanations)**: This technique uses cooperative game theory to fairly attribute contributions to each feature in a model's prediction, providing a unified measure of feature importance across various models [12].

- **Permutation Feature Importance:** By randomly permuting feature values and measuring the change in model accuracy, this method identifies the most impactful features in the model's decisions [12].
- **Partial Dependence Plots (PDPs):** These plots visualize the relationship between a feature and the predicted outcome, holding other features constant, to show how changes in one feature influence predictions [12].

In this work, the decision was made to operate under the second option, namely post-hoc interpretability. The machine learning models adopted (as discussed in ??) are predominantly very complex, and due to this complexity, key information is gathered immediately after their training to enable more effective operation during their use.

#### 1.4.2. Partial Dependence Plot (PDP)

In the context of machine learning and statistical modeling, understanding the relationship between input features and a model's predictions is crucial, particularly when working with complex, nonlinear models such as decision trees, random forests, and gradient boosting machines. Since these models often function as "black boxes," one of the primary challenges is interpreting how individual features influence predictions. *Partial Dependence Plots* (PDPs) [12] provide an effective solution to this challenge by visualizing the marginal effect of one or more features on the predicted outcome, while averaging out the effects of all other features.

A PDP represents the expected prediction of a model as a function of a specific input variable or a set of input variables while holding all other variables constant. This helps isolate and interpret the contribution of each feature to the model's prediction. PDPs are particularly useful for understanding whether the relationship between a feature and the predicted outcome is linear, monotonic, or exhibits more complex behavior. They have been used to gain insights into the dependence of controllable features on the success of a specific requirement (e.g. robot speed for distance to the person). A PDP is created for each controllable feature of the system, illustrating how the categorization of the model's output as satisfied or not satisfied depends on that feature. Certain feature values can maximize or minimize the probability of meeting a requirement, and thus, these values are studied to consistently maximize this probability.

Mathematically, the partial dependence function for a set of features  $x_S$  can be expressed as:

$$\hat{f}_S(x_S) = \mathbb{E}_{X_C}[\hat{f}(x_S, X_C)] = \int \hat{f}(x_S, X_C) dP(X_C), \quad (1.1)$$

where  $x_S$  represents the features of interest,  $X_C$  denotes all other features, and  $\hat{f}(x_S, X_C)$  is the model's prediction for a given instance. The PDP then shows how the model's predictions change as  $x_S$  varies while averaging out the effects of the other features.

## 1.5. SHAP and Shapley values

SHAP (SHapley Additive exPlanations) [12] is a game-theoretic approach for explaining the output of any machine learning model. It links optimal credit allocation with local explanations, leveraging Shapley values [12] from cooperative game theory. These values represent how to fairly distribute the "payout" (or prediction) among the features contributing to the model's output, allowing for an interpretable feature ranking (refer to related papers for further details and citations). [12]

---

**Algorithm 1:** 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:  $\phi_j(x) = \frac{1}{M} \sum_{m=1}^M \phi_j^m$ 

```

---

**Definition 1.** The Shapley value of a feature value is its contribution to the payout, weighted and summed over all possible feature value combinations:

$$\phi_j(val) = \sum_{S \subseteq \{1, \dots, p\} \setminus \{j\}} \frac{|S|! (p - |S| - 1)!}{p!} (val(S \cup \{j\}) - val(S)) \quad (1.2)$$

where  $S$  is a subset of the features used in the model,  $x$  is the vector of feature values of the instance to be explained, and  $p$  is the number of features. [12]

**Definition 2.**  $val_x(S)$  is the prediction for feature values in set  $S$  that are marginalized over features that are not included in set  $S$ :

$$val_x(S) = \int \hat{f}(x_1, \dots, x_p) d\mathbb{P}_{x \notin S} - E_X(\hat{f}(X)) \quad (1.3)$$

SHAP has been used to classify the most important controllable features for the system's output. This approach allows for prioritizing modifications to these features first, followed by adjustments to the remaining features, as they exhibit a more subtle dependence on the model's output.

## 1.6. Permutation Feature Importance

The `sklearn.inspection` package provides an alternative approach to feature ranking through permutation feature importance [8, 14]. This method measures the contribution of each feature to the performance of a fitted model on a given dataset. It works by randomly shuffling the values of a single feature and measuring the resulting degradation in the model's performance. By disrupting the relationship between the feature and the target variable, this technique reveals how much the model relies on each feature.

---

**Algorithm 2:** 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  $FI$
- 

Permutation feature importance, like SHAP, has been used to classify the controllable features to obtain a more comprehensive understanding of their dependence on whether the requirements are met or not.

## 1.7. NSGA-III

NSGA-III (Non-dominated Sorting Genetic Algorithm III) [2, 6] is an advanced evolutionary algorithm designed for multi-objective optimization, especially effective when dealing with high-dimensional objective spaces. It extends the capabilities of NSGA-II [2] by introducing a reference-point-based mechanism that maintains diversity across the objectives.

**Initialization:** The algorithm begins by initializing a random population of individuals,  $P_0$ , of size  $N$ . Each individual represents a potential solution to the optimization problem, and the objective values are evaluated for each solution. Additionally, a set of reference points  $\mathbf{Z}$  is generated to ensure diversity in the solution space. These reference points are used later in the algorithm to guide the selection of solutions based on their proximity to the reference points in the objective space.

**Main Evolutionary Loop:** The algorithm proceeds through a series of generations (iterations), with  $G$  representing the maximum number of generations. At each generation, the following steps are executed:

- An offspring population  $Q_t$  is generated from the parent population  $P_t$  using genetic operators such as crossover and mutation. This process introduces variability into the population, allowing the algorithm to explore new areas of the solution space.
- The objective values of the offspring population are then evaluated.
- The parent and offspring populations are combined to form a larger population  $R_t$ , which contains both parent and offspring individuals. The size of  $R_t$  is  $2N$ .

**Non-Dominated Sorting:** To maintain the best solutions across generations, a key step in the algorithm is the non-dominated sorting of the combined population  $R_t$ . The `NonDominatedSort` function divides  $R_t$  into a series of non-dominated fronts,  $F_1, F_2, \dots, F_k$ . Solutions in the first front  $F_1$  are non-dominated with respect to the entire population, meaning that no other solution in  $R_t$  is strictly better in all objectives. Similarly, solutions in the second front  $F_2$  are only dominated by those in  $F_1$ , and so on for subsequent fronts.

**Population Selection:** The next step is to construct the population for the next generation,  $P_{t+1}$ , from the sorted fronts. The algorithm adds entire fronts  $F_1, F_2, \dots, F_i$  to  $P_{t+1}$  until adding the next front  $F_{i+1}$  would exceed the population size  $N$ . At this point, the algorithm proceeds to select individuals from the

remaining front,  $F_i$ , using a reference-point-based selection mechanism.

**Reference Point-Based Selection:** Since the front  $F_i$  cannot be fully added to  $P_{t+1}$  without exceeding the population size limit, the algorithm must carefully choose which solutions to include. This is where the set of reference points  $\mathbf{Z}$  comes into play. The solutions in  $F_i$  are normalized and then associated with the nearest reference points in  $\mathbf{Z}$ . The algorithm selects solutions that are close to under-represented reference points, thereby ensuring that the diversity of the population is preserved across all objectives.

**Termination:** The process of offspring generation, evaluation, sorting, and selection repeats for  $G$  generations. After the final generation, the algorithm returns the final non-dominated population,  $P_G$ , which represents a diverse set of high-quality solutions across all objectives.

## 1.8. FITEST

The Feature Interaction Testing (FITEST) [1] algorithm is an evolutionary algorithm designed to solve multi-objective optimization problems through the principles of genetic algorithms such as selection, crossover, and mutation. FITEST aims to generate high-quality test suites that cover a given set of objectives  $\Omega$ , representing safety and functional requirements in a feature-based control system. The algorithm seeks to cover as many objectives as possible by evolving a population of solutions over multiple generations, gradually improving their ability to satisfy the test objectives.

In the context of FITEST, a feature-based control system  $F$  is defined as a tuple  $(f_1, \dots, f_n, \text{IntC})$ , where  $f_1, \dots, f_n$  represent individual features and IntC is an integration component. The set of test objectives,  $\Omega = \{\Omega_{1,1}, \dots, \Omega_{k,n}\}$ , defines the system's branches and safety requirements, with  $k$  representing the number of branches in the integration component and  $n$  the number of safety requirements for each branch. The goal of the FITEST algorithm is to find a test suite that covers as many objectives  $\Omega_{i,j}$  as possible, with "coverage" defined as satisfying the requirement to at least 80%.

**Initialization:** FITEST starts by generating an initial population,  $P$ , using an adaptive random population generation method. The population size corresponds to the number of objectives  $|\Omega|$ . The initial population represents potential test suites, and each solution in  $P$  is evaluated by calculating its objective values relative to  $\Omega$ . The algorithm identifies which objectives are covered by the initial population using the GET-COVERED-OBJECTIVE function, which returns the subset of objectives

$\Omega_c$  that are already covered and the corresponding test cases  $T_c$ . These test cases are added to the archive,  $A$ , which stores all test cases that successfully cover at least one objective.

**Main Evolutionary Loop:** The algorithm proceeds through an iterative loop, where a new offspring population  $Q$  is generated by recombining the solutions from the current population  $P$  using crossover and mutation operations. Once the offspring population is generated, it undergoes a correction step to ensure that the new solutions conform to the problem's constraints and objectives. The objectives of the new population  $Q$  are then recalculated, and the set of newly covered objectives  $\Omega_c$  is identified. At each iteration, the archive  $A$  is updated with any new test cases that cover additional objectives. The set of uncovered objectives  $\Omega$  is also updated by removing the objectives that have already been covered.

**Environmental Selection and Population Update:** Once the new objectives are evaluated, the algorithm performs environmental selection, which determines the next generation's population,  $F_0$ . This selection step is crucial for maintaining diversity in the population while ensuring that the solutions continue to improve in terms of objective coverage. The environmental selection combines the current population  $P$  and the offspring population  $Q$  to create a new population  $P$  for the next iteration. The selected population  $P$  contains the best individuals that have shown promising coverage of the remaining objectives.

**Termination Condition:** The algorithm continues to iterate through the main evolutionary loop until a stopping condition is met. This condition can be based on a maximum number of generations, a threshold for objective coverage, or the inability to make further progress in covering new objectives. Once the stop condition is satisfied, the algorithm terminates and returns the archive  $A$ , which contains the test cases that provide the best coverage of the test objectives  $\Omega$ .

FITEST is used as a comparison algorithm alongside NSGA-III. The goal is to compare the performance of the proposed adaptation method against FITEST and NSGA3, which serve as a baseline. By doing so, it is possible to evaluate the robustness and effectiveness of XDA in terms of system adaptation, performance optimization, and success probability.

## 1.9. Faiss

To find the Pareto front mentioned earlier (Section 1.2), the kNN algorithm is used to obtain multiple solutions, which can then be classified and from which the best option

can be selected for meeting the requirements. To speed up the search for the  $k$  nearest neighbors in the dataset and make it more efficient, Meta's Faiss library has been adopted. Faiss [7] is a library designed for efficient similarity search and clustering of large sets of dense vectors. It is particularly useful for applications involving high-dimensional data. Faiss is employed to speed up the  $k$ -nearest neighbors (kNN) search, enhancing the performance of adaptation algorithms that rely on proximity-based feature analysis.

Given a set of vectors  $\{x_1, \dots, x_n\}$  in dimension  $d$ , Faiss builds a data structure in RAM. After the structure is constructed, when given a new vector  $x$  in dimension  $d$ , it performs the operation:  $i = \arg \min_i \|x - x_i\|$ , where  $\|\cdot\|$  represents the Euclidean distance (L2).



# 2 | Proposed methods

## 2.1. Problem Statement

It is well-established that autonomous systems must meet their objectives at runtime, meaning they must be able to make real-time decisions without interruptions, patching, or redeployment. Indeed, every autonomous system operates within a working environment that generates information impacting the functioning and behavior of the self-adaptive system. The external environment can be analyzed to extract valuable insights about what affects the objectives that need to be met and what can be disregarded due to its minimal influence on those objectives.

In this context, the environment is described by a tuple of values representing the *Observable Features* (OF), which are elements that neither the autonomous system nor the designer can control, such as weather conditions, time of day, road conditions, etc. On the other hand, the State of the autonomous system is represented by another tuple that captures all the elements the system can control, known as the *Controllable Features* (CF), to meet the system's requirements. At any given time  $t$ , the combination of Controllable Features, Observable Features, and the system requirements forms what is known as the system's *operating condition* at that specific time.

Due to the multi-objective optimization nature of the problem (i.e., the search for tuples of values that satisfy more than one requirement), two generative algorithms, considered standards in this field, have been selected: NSGA-III and FITEST.

The former is a generative algorithm specifically focused on solving multi-objective optimization problems, while the latter is a test generator designed to produce ideal tests for a given system. Its purpose is not strictly to search for targeted adaptations but rather to find tests that cover one or more objectives (where, in the case of FITEST, the objectives determine how far candidate tests are from detecting undesired interactions).

The control loop's decision-making process is significantly affected by the black-box nature of embedded neural networks (NNs). These systems often operate with limited

transparency, making it difficult for engineers and system administrators to interpret their behavior. This lack of clarity poses a critical challenge when trying to understand or debug the control loop, especially in scenarios where safety is paramount, like in our rescue robot example or UAV. Moreover, adaptation decisions in this context are usually the result of optimization processes designed to maximize the chance of meeting specific objectives. However, optimizing such black-box models in large search spaces can be highly resource-intensive. The challenge stems from the fact that the internal workings of NNs are not directly observable or understandable, forcing iterative approaches, such as evolutionary algorithms, to explore vast semantic spaces while balancing multiple objectives. As a result, achieving a compromise between optimization efficiency and cost becomes a pressing issue, exacerbated by the "curse of dimensionality" which makes runtime adaptation decisions even more expensive.

## 2.2. Explanation-Driven Approach

Negri, Nicolosi, et al. [13] introduced a novel approach to address this issue, called eXplanation Driven self-adaptation (XDA). This method collects the system's operating condition at a given time  $t$ , analyzes which requirements are satisfied and which are not, and then searches for the ideal combination of Controllable Features that, together with the Observable Features at that moment, satisfies the maximum possible number of requirements.

In this thesis, XDA is extended to make the system even more interpretable through feature ranking. This extension, called XDAv2, provides a classification of the controllable features of the monitored system, offering a prioritized view of which features have the most influence on system behavior. This ranking of features helps system administrators and engineers focus on the most impactful elements when adapting the system, further improving decision-making efficiency.

XDAv2 is integrated in the MAPE-K Loop, which is embedded in a self-adaptive system. Periodically the *monitor* collects samples about the operating current condition (i.e. an assignment of controllable features  $CF = \hat{CF}$  and observable features  $OF = \hat{OF}$ ) of the managed system, then sends them to the *analyze* component. With the help of the set of pre-trained classifiers  $M = \{M_1, \dots, M_n\}$  the *analyze* uses each classifier  $M_i$  to predict the probability of satisfaction of the corresponding requirement  $R_i$ , where the class label of  $R_i$  is *true* if it is satisfied else if unsatisfied *false*. If there is at least one requirement whose predicted class label is *false* the *plan* is triggered by *analyze* and searches one or more new assignments to CFs that increase the probability of satisfying all the requirements. Such

assignments represent alternative adaptation options among which the final adaptation decision can be made according to a given policy (e.g., highest class probability estimate).

XDAv2 shares most of its components with XDA, as it includes both the online white optimizer and the offline pre-processor. As shown in the Figure 2.2, XDA has been extended with the SHAP algorithm and permutation feature importance, in terms of the offline pre-processor, as well as with a profiler.

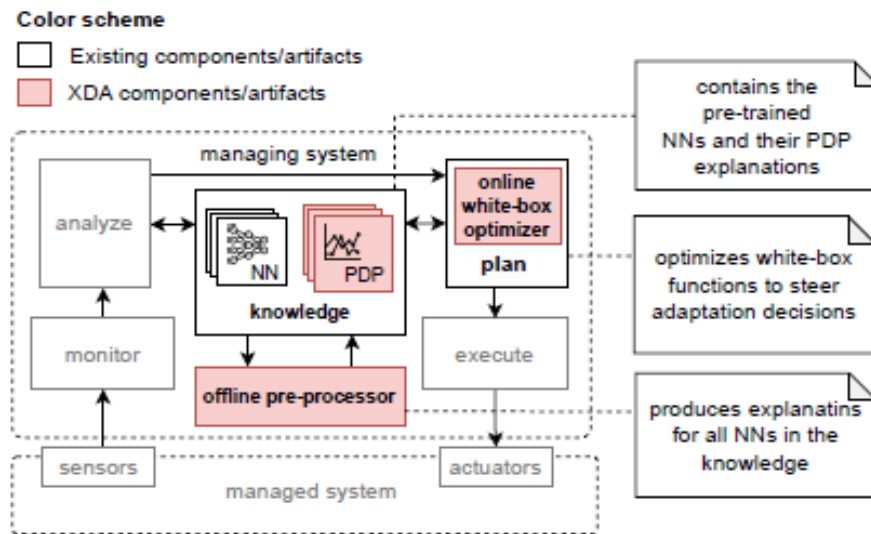


Figure 2.1: MAPE-K Loop with XDA

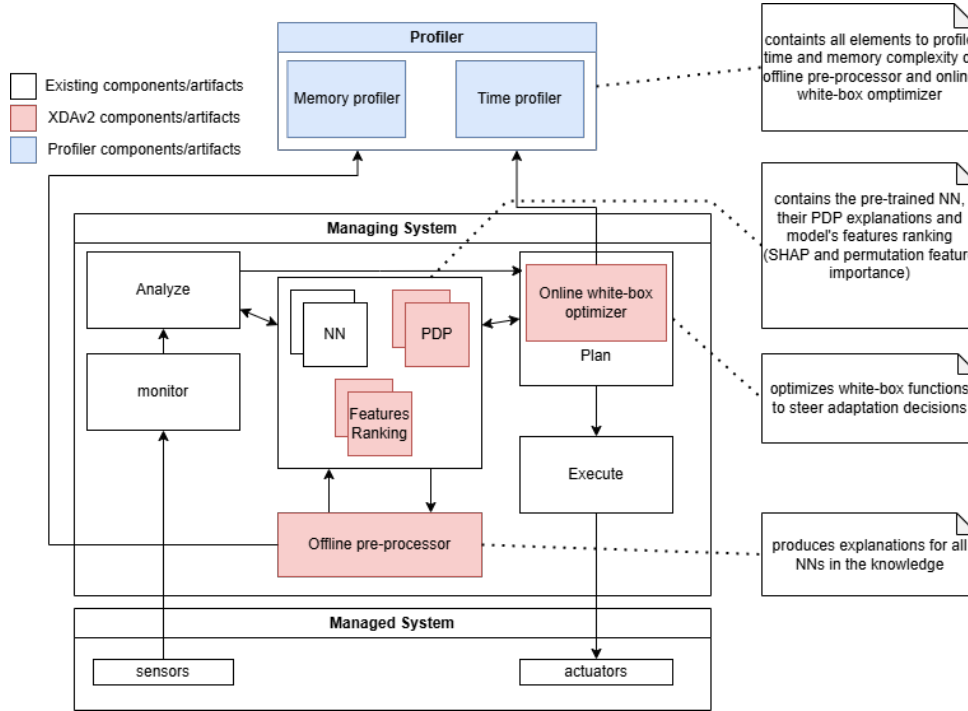


Figure 2.2: MAPE-K Loop with XDAv2

XDA is divided into two parts and is extended with a third part:

- **Offline Pre-Processor** is executed before the deployment and is used to create and train the set of classifier  $M = \{M_1, \dots, M_n\}$  and to build *PDPs*. It is expanded in the new version with two feature ranking methods (*SHAP* and *Permutation Feature Importance*) that explain the impact of a certain feature  $CF_i$  on all set of classifier  $M$ . This information is collected to make the system more explainable and to make the Online White-box Optimizer more efficient (Section 2.2.1).
- **Online White-box Optimizer** runs with the other component of the MAPE-K control loop after deployment. It reads *PDPs* to search for possible adaptation options (Section 2.2.2).

### 2.2.1. Offline Pre-processor

After the creation and the training of the set of classifiers  $M$ , the offline pre-processor creates a set of partial dependence plots *PDPs* that provide an advanced understanding between a given feature and the target prediction of the black-box classifiers created earlier. For each feature  $CF_i$  and each requirement  $M_j$ , a partial dependence plot ( $PDP_{ij}$ ) [12] is created (Algorithm 3), representing the dependence of satisfaction of the studied requirement concerning the range of values that the variable can assume in its domain.

A score function is used to classify each  $CF_i$  based on all models (line 10). The score function for each sample is essentially the product of all individual conditional expectation curves across all models:

$$score(j, s, x) = \prod_i PDP_{i,j}(s, x) \quad (2.1)$$

The output of Algorithm 3 are the set of PDP and the set  $SPDP$ , i.e. all score functions.

---

**Algorithm 3:** 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   $SPDP \leftarrow \text{updateSet}(SPDP, score_j)$ 
12 end

```

---

The offline pre-process was then extended by adding a feature ranking functionality. Using two algorithms (SHAP and permutation feature importance [12]) a ranking is generated for the set of controllable features  $CF$ . Both algorithms apply different methods to test and evaluate the importance of individual features concerning the output of the machine learning model being used. In this case, the ranking of controllable features  $CFs$  is based on their relationship with the output of the classifier models  $M_j$ .

SHAP (as described in Section 1.5) was applied in this component by calculating the Shapley values for each feature, averaging them for each classifier  $M_j$  (Algorithm 4), and finally summing all averages to obtain a general overview of the importance of a single feature across all requirements  $M$ . Since a feature  $CF_i$  is more important for the model's output when its SHAP values are higher, the set of controllable features indices  $CFI$  is

then reordered in descending order based on the SHAP values of each feature.

---

**Algorithm 4:** SHAP Features Classifier
 

---

**Input:** Set of classifiers  $M = \{M_1, \dots, M_n\}$ , Data matrix  $X$ , Set of controllable features indices  $CFI = \{CFI_1, \dots, CFI_m\}$

**Output:** Sorted  $CFI$  by cumulative importance

```

1 Initialize cumulative_importance  $\leftarrow$  array of zeros of size  $m$ 
2 foreach classifier  $M_i$  in  $M$  do
3    $Scores_{SHAP} \leftarrow \text{shapClassifier}(M_i, X, CFI)$ 
4   for  $j \leftarrow 1$  to length of  $Scores_{SHAP}$  do
5      $cumulative\_importance[j] \leftarrow cumulative\_importance[j] + Scores_{SHAP}[j]$ 
6   end
7 end
8  $CFI \leftarrow CFI$  sorted in descending order of importance
9 return  $CFI$ 

```

---

The following images provide an example of SHAP feature ranking. They use bar charts to represent the importance of each feature (both controllable and observable) to the output of each requirement model. For simplicity, observable features are excluded from selection and modification, as they are not useful in the search for adaptation. Since they are observable, they cannot be modified by the system and the ranking focuses on controllable features, those that can be modified by the system. The higher the mean of the SHAP values, the more important the associated feature is for the model's output.

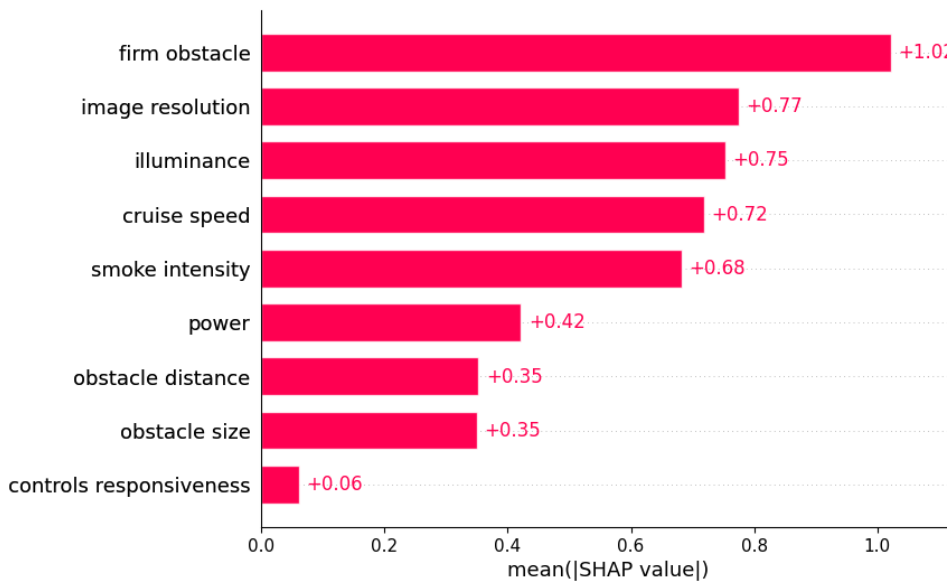


Figure 2.3: SHAP features ranking requirement 1 rescue robot

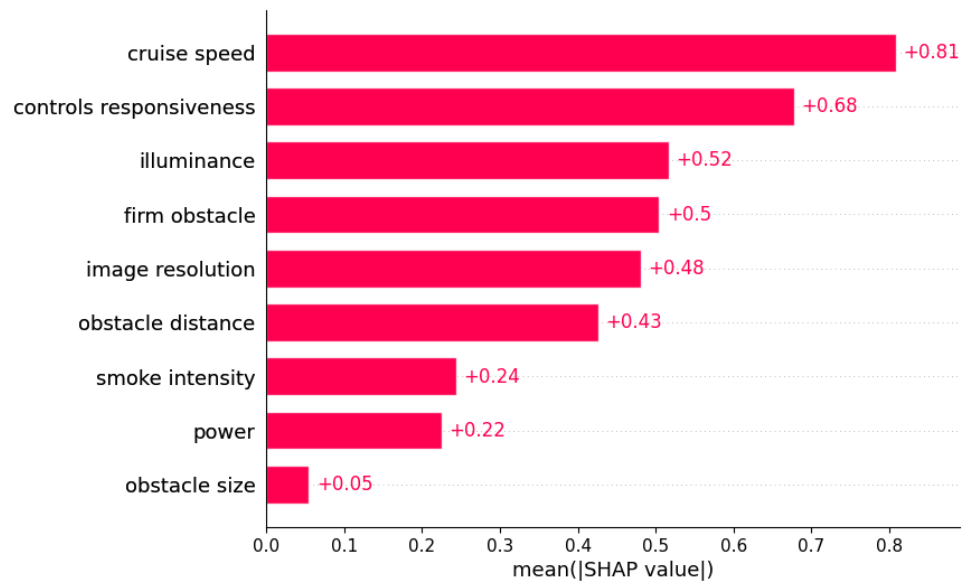


Figure 2.4: SHAP features ranking requirement 2 rescue robot

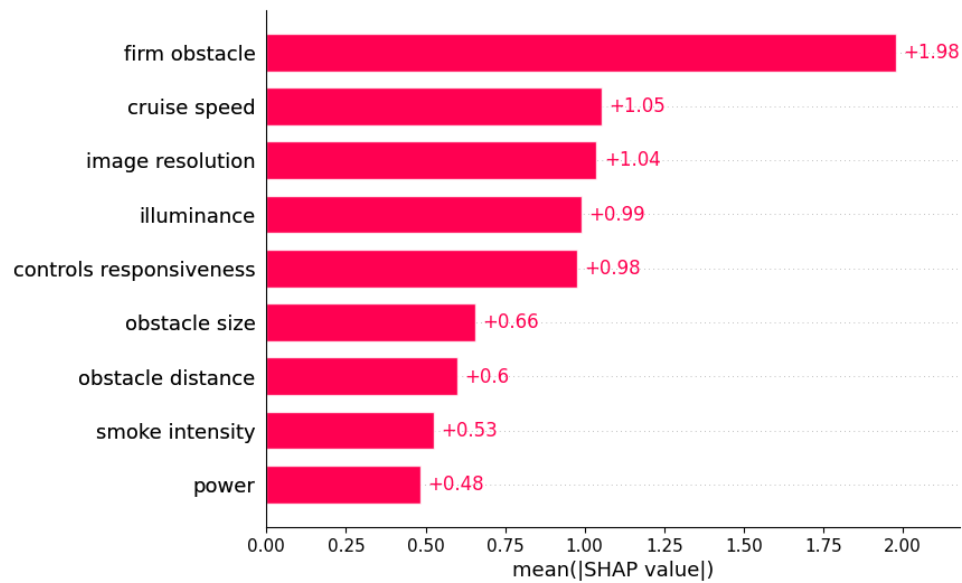


Figure 2.5: SHAP features ranking requirement 3 rescue robot

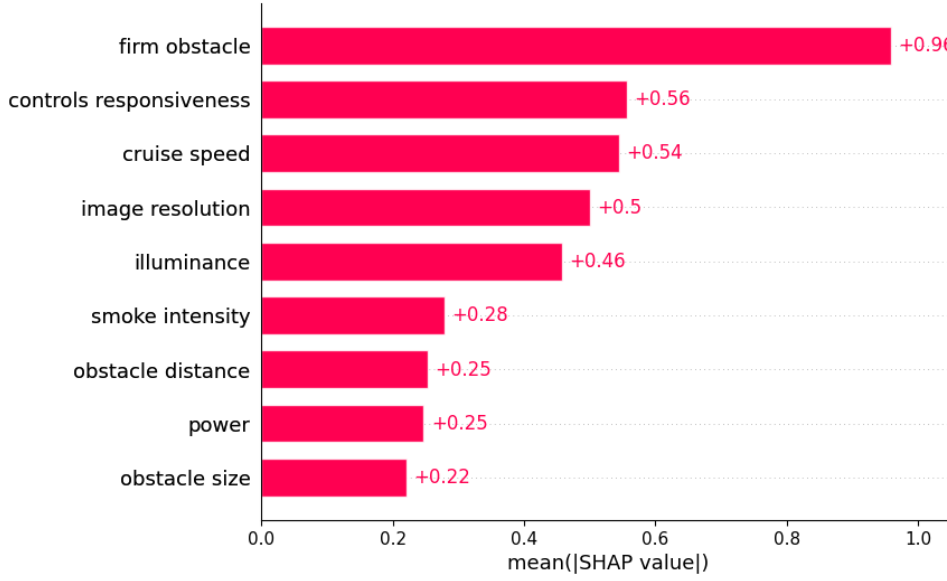


Figure 2.6: SHAP features ranking requirement 4 rescue robot

Permutation feature importance (as described in Section 1.6) operates similarly. The value associated with a CF is calculated by summing the averages for each requirement and for each feature (Algorithm 5), after which the set of controllable feature indices is reordered accordingly.

---

**Algorithm 5:** Permutation Importance Classifier

---

**Input:** Set of classifiers  $M = \{M_1, \dots, M_n\}$ , Data matrix  $X$ , Labels  $Y$ , Set of controllable features indices  $CFI = \{CFI_1, \dots, CFI_m\}$

**Output:** Sorted  $CFI$  by cumulative permutation importance

```

1 Initialize cumulative_importance  $\leftarrow$  array of zeros of size  $m$ 
2 foreach classifier  $M_i$  in  $M$  do
3    $feature\_indices \leftarrow$  permutation_importance_classifier( $M_i, X, Y, CFI$ )
4   for  $j \leftarrow 1$  to length of feature_indices do
5      $cumulative\_importance[j] \leftarrow$ 
6      $cumulative\_importance[j] + feature\_indices[j]$ 
7   end
8 end
9  $CFI \leftarrow CFI$  sorted in descending order of importance
10 return  $CFI$ 

```

---

### 2.2.2. Online White-box Optimizer

When the operating condition of the managed system is described by a set of controllable and observable features that lead to one or more negative evaluations of the requirements—i.e., the class label for one or more requirements is set to false—the online white-box optimizer is triggered by the analyze component. Once triggered, the optimizer records the current operating condition  $(\hat{CF}, \hat{OF})$  and, using the information collected by the offline pre-processor (PDP, SPDP, and feature ranking), it generates a new set  $(\hat{CF}^*)$  such that  $(\hat{CF}^*, \hat{OF})$  yields a true class label with a probability higher than a given threshold  $P$  for all requirements.

As shown in the schema, the algorithm takes as input the set of controllable features  $(CF)$ , the set of observable features  $(OF)$ , the reordered feature indices ranked by importance (provided by the ranking algorithms,  $CFI$ ), and the current operating condition represented by  $(\hat{CF}, \hat{OF})$ .

The first step of the algorithm is to find  $k$  neighbors within the training dataset  $D$ . This is done to check whether there are data points in the dataset that are similar to the current operating condition. The algorithm used to find the neighbors is Faiss [7], which measures the Euclidean distance (L2) between data points in the semantic space. The neighbors  $(\hat{CF}', \hat{CF}')$  are then modified using the guidance of the score functions to improve the satisfaction probability of each requirement.

The modification of the controllable features is divided into two steps: the *climbing step* and the *descending step*.

The *climbing step* makes coarse-grained adjustments to the controllable features, taking into account the score functions to maximize the probability of satisfying all requirements. The *descending step* refines the results of the climbing step by reversing the adjustment process, slowly decreasing the satisfaction probability at a low descent rate ( $\Delta$ ). This fine-tuning improves the values of the controllable features while keeping the satisfaction probability of all requirements above the threshold  $P$ .

**Climbing step** For each neighbor (line 4), the algorithm iterates through all the controllable features (lines 8-14), and by evaluating the score functions, it identifies the feature  $CF_h$  that maximizes the value of  $y$  (where  $y$  is the output of the score function). The value that maximizes  $y$  is then stored in  $CF_h$  (line 15), and the next closest neighbor is identified (line 18). At this point,  $CF_h$  can be excluded, and the same procedure is applied to the remaining unprocessed controllable features until all of them have been evaluated. Finally, their assignment to the controllable features  $\hat{CF}''$  becomes a possible

adaptation option, incorporating the entire operating condition  $(\hat{CF}'', \hat{OF})$  into the set  $S$  (line 20).

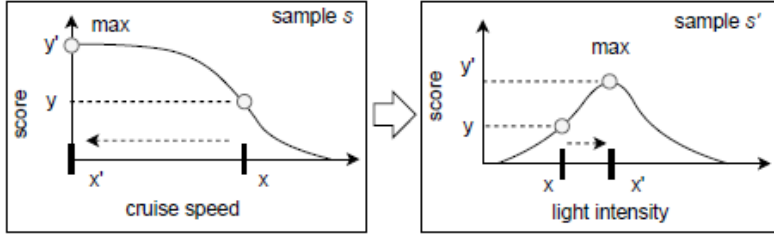


Figure 2.7: climbing step. [13]

**Descending step** The descending step is used to improve further the initial adaptation options found in the previous step by slowly adjusting the values of the controllable features while maintaining the probability of requirement satisfaction above a certain threshold  $P$  for all requirements. Domain knowledge is used to determine ideal values for each controllable feature. The descending step considers one adaptation option at a time and iterates through all options sequentially (line 24), evaluating the controllable features based on the ranking provided by the SHAP and permutation feature importance algorithms (Algorithm 4, Algorithm 5) executed in the offline pre-processor at deployment time. The descent direction is determined at the beginning and indicates whether the ideal value of the controllable feature should be the minimum (-1) or maximum (1) of its domain. For instance, if  $CF_j < I_j$  and the descent direction is left-to-right (1), the procedure will tend to increase the value of  $CF_j$ . Conversely, if  $CF_j > I_j$  and the descent direction is right-to-left, the procedure will decrease the value of  $CF_j$  (line 29). The controllable features  $CF$  are adjusted by a fixed constant value multiplied by the descent direction. This process continues until no further modifications can be made, or the satisfaction probability of at least one requirement falls below the threshold  $P$  (line 31).

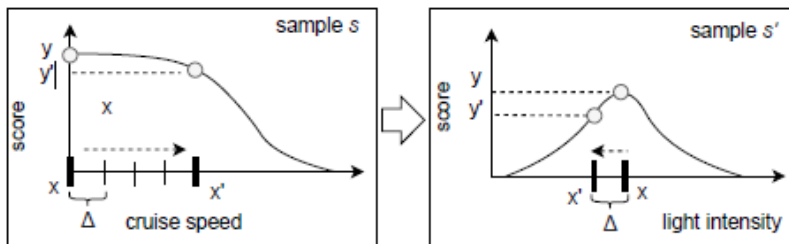


Figure 2.8: descending step. [13]

**Algorithm 6:** Online white-box optimization with improved features selection

**Input:** Set of classifiers  $\{M_1, \dots, M_n\}$  associated with requirements, probability threshold  $P$ , set of controllable features  $CF = \{CF_1, \dots, CF_m\}$  sorted in order of importance (SHAP or FI) and corresponding values  $\hat{CF} = \{\hat{CF}_1, \dots, \hat{CF}_m\}$ , set of observable features  $OF = \{OF_1, \dots, OF_l\}$  and corresponding values  $\hat{OF} = \{\hat{OF}_1, \dots, \hat{OF}_l\}$ , set of ideal values for  $CF$   $I = \{I_1, \dots, I_m\}$ , set of summary PDPs  $SPDP = \{SPDP_1, \dots, SPDP_m\}$ , training set  $D$  of the classifiers in  $M$ , number of neighbors  $k$ , descent step  $\Delta$ .

**Output:** Set of possible adaptations  $\{\hat{CF}^*\} = \{\hat{CF}_1^*, \dots, \hat{CF}_m^*\}$

```

1 Set of possible adaptations  $S \leftarrow \emptyset$ 
2 Set of neighbors  $U \leftarrow \text{neighborhood}(\hat{CF}, \hat{OF}, D, k)$ 
3 /* First step: climbing */
4 foreach  $(\hat{CF}', \hat{OF}') \in U$  do
5    $y \leftarrow 0$ 
6   Set of changed CFs  $A \leftarrow \emptyset$ 
7   while  $|A| < m$  do
8     for  $j \leftarrow 1$  to  $m$  do
9        $(x', y') \leftarrow \max(SPDP_j, \hat{CF}', \hat{OF}')$ 
10      if  $j \notin A$  and  $y' > y$  then
11         $x \leftarrow x'$ 
12         $y \leftarrow y'$ 
13         $h \leftarrow j$ 
14      end
15       $\hat{CF}'_h \leftarrow x$ 
16       $\hat{CF}''_h \leftarrow x$ 
17       $A \leftarrow \text{updateSet}(A, h)$ 
18       $\{(\hat{CF}', \hat{OF}')\} \leftarrow \text{neighborhood}(\hat{CF}', \hat{OF}', D, 1)$ 
19    end
20     $S \leftarrow \text{updateSet}(S, (\hat{CF}'', \hat{OF}'))$ 
21  end
22  $A \leftarrow \emptyset$ 
23 /* Second step: descending step */
24 /* CF sorted in order of importance (SHAP or FI) */
25  $featureIndices \leftarrow \{j \mid j \in CF \wedge j \notin A\}$ 
26 foreach  $(\hat{CF}^*, \hat{OF}) \in S$  do
27   while  $|A| < m$  do
28     foreach  $j \in featureIndices$  do
29        $\hat{CF}^*_j \leftarrow \text{move}(\hat{CF}^*_j, I_h, \Delta)$ 
30     end
31     if  $\exists M_i : P_{M_i}(y = true \mid (\hat{CF}^*, \hat{OF})) < P$  then
32        $A \leftarrow \text{updateSet}(A, j)$ 
33     end
34   end
35 end

```

### 2.2.3. Profiling

The *profiler* is executed to collect data about time execution and memory allocation of both offline pre-processor and online white-box optimizer. A Python package called `cProfile` [9] was used to profile the main function (`findAdaptation`), along with its internal functions, to identify which part of the code represented a performance bottleneck. At the end of its execution, `cProfile` provides the total and individual execution times, as well as the number of calls for each function within `findAdaptation`. The function was identified as a performance bottleneck, particularly in scenarios where it was invoked a significant number of times. To address this inefficiency, the function was modified to allow simultaneous updates to multiple features within a single invocation. The revised function, rather than handling one feature per call, constructs a list of all features that are eligible for modification (i.e., not present in the `excludedFeatures` or `tempExcludedFeatures` lists). It then iterates over this list and applies the corresponding adjustments in a single loop.

Finally, the Python package `tracemalloc` [10] was used to profile the memory consumption of each algorithm within the main adaptation search function. It tracked the peak memory usage for each test, which was then compared and visualized in a graph.

# 3 | Experimentation

Three types of evaluation subjects were selected, all of which involve a self-adaptive system managing specific situations. The first subject is a simulated rescue robot (*RR*) that navigates a designated area (enhanced with various obstacles) to identify injured individuals while maintaining compliance with its requirements. The second describes an unmanned aerial vehicle (*UAV*) that flies within a certain area containing threats, prioritizing safety as its primary goal. The third case involves an autonomous vehicle (*AV*) encountering a pedestrian crossing the road during its journey.

Every system aforementioned can be modeled following Section 1.1 in fact all systems comprehend one environment, one managing system, one managed system, and some adaptation goals to satisfy. The goal of all systems is to modify the main components so that, at runtime, they can evaluate and choose the best adaptations based on a given situation.

For example, the rescue robot might struggle when communication is poor (low bandwidth) or faces many obstacles. Similarly, the UAV could encounter adverse weather conditions or a high number of threats. Lastly, the autonomous vehicle may come across a pedestrian crossing the road during its journey.

Here, the three instances used throughout this work are listed with general descriptions.

## Rescue Robot

**Environment:** Dynamic disaster areas with debris and unpredictable conditions.

**Managing System:** Control system responsible for navigating obstacles and coordinating robot actions.

**Managed System:** Robot sensors and actuators, including locomotion and environment detection.

**Adaptive Goals:** Ensure safe navigation and effective search and rescue in uncertain conditions.

### UAV (Unmanned Aerial Vehicle)

**Environment:** Outdoor environments, often in complex terrains or urban areas.

**Managing System:** Ground control systems or autonomous flight controllers.

**Managed System:** Onboard navigation, GPS, and communication systems.

**Adaptive Goals:** Maintain stable flight and accomplish surveillance.

### Autonomous Driving

**Environment:** Public roads, interacting with other vehicles and pedestrians.

**Managing System:** Autonomous driving software, handling decision-making and route planning.

**Managed System:** Sensors and actuators controlling the vehicle's movement.

**Adaptive Goals:** Safely reach the destination while avoiding obstacles.

For each of the aforementioned systems, a model was created to translate each scenario into a set of tuples that would describe, as precisely as possible, the operating conditions of the system and the satisfaction of adaptive goals within the environment, this made it possible to modify the behavior of the managed system within the environment to achieve the fulfillment of all adaptive goals.

Every component is modeled to obtain a valid abstraction of the system:

- **Environment:** a tuple of *Observable Features (OFs)* representing the variables to which the managed system must adapt.
- **Managed System:** a tuple of *Controllable Features (CFs)* representing the modifiable tools of the managed system (e.g., for the robot: cruise speed, camera quality; for the UAV: whether to fly in formation or not; for the autonomous vehicle: the vehicle's speed).
- **Adaptive Goals:** *requirements (Rs)* that must be satisfied in the external environment (e.g., the probability that the rescue robot does not misclassify an obstacle must be greater than 0.85).
- **Managing System:** the method used to calculate the best assignment of the controllable features of the managed system to achieve the optimal behavior in a given environment.

- **Adaptation:** A tuple of values corresponding to the various controllable features ( $\hat{C}\hat{F}^r$ ).

Each test scenario (i.e. rescue robot, UAV, autonomous driving) is divided into three parts, respectively no label, v2 and v3 (e.g. UAV, UAVv2, UAVv3), representing three different test instances. Each instance can be characterized by a dataset and its associated constraints. Initially, the constraints are created to represent the requirements that the system must meet, followed by the generation of a dataset created artificially that contains 5000 samples representing the system’s operating condition logs. It is important to emphasize that the division of variations into three parts is because the constraints become progressively stricter with each part, starting from the first.

The following tables describe used systems as a set of controllable features , which represent the parameters that can be controlled by the system (such as cruise speed for the robot, formation for the UAV, and speed for the autonomous vehicle), and observable features, which represent parameters belonging to the external environment along with their respective domains and types.

Variable	Dimensions	Type	Domain
cruise speed	CF	continuous	[0, 5] m/s
camera quality	CF	categorical	{low, mid, high}
control responsiveness	CF	continuous	[10, 50] Mbit/s
illuminance	CF	continuous	[40, 120000] lux
power level	OF	integer	[0, 100] %
smoke intensity	OF	categorical	{none, thin, thick}
obstacle size	OF	continuous	[0, 2] m <sup>3</sup>
obstacle distance	OF	continuous	[0, 10] m
firm obstacle	OF	Boolean	Yes/No

Table 3.1: Features Rescue Robot

Variable	Dimensions	Type	Domain
formation	CF	Boolean	Yes/No
flying speed	CF	continuous	[5, 50]
countermeasure	CF	Boolean	Yes/No
weather	OF	integer	[1, 4]
day time	OF	integer	[0, 23]
threat range	OF	continuous	[1000, 40000] m
number of threats	OF	integer	[1, 100]

Table 3.2: Features UAV

Variable	Dimensions	Type	Domain
car speed	CF	continuous	[5, 50] km/h
position x pedestrian	OF	continuous	[0, 10]
position y pedestrian	OF	continuous	[0, 10]
orientation	OF	integer	[-30, 30] degree
weather	OF	integer	[0, 2]
road shape	OF	integer	[0, 2]

Table 3.3: Features Autonomous Drive

These evaluations allowed us to address the following questions:

**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?

The method used is many-objective optimization, where system configurations are selected to increase the likelihood of satisfying the system requirements concerning a given threshold, by adjusting the controllable features toward their ideal values. The fitness for each controllable feature is defined as the normalized distance between the actual point (given by the selected adaptation option) and the ideal one. In addition to objectives, constraints are set in the form  $P_{M_i}(y = true | x) > P$ , where  $P$  is the minimum probability of satisfying the specific requirement  $R_i$ . The probability of satisfaction is calculated through the probability estimate expectation set by the corresponding classifier  $M_i$ .

Several baselines were employed to contextualize the results and enable the ability to answer the questions. The first is NSGA-III, a mainstream many-objective metaheuristic optimizing search algorithm. The second is FITEST, an ideal search algorithm for test generation aimed at multi-objective optimization. Additionally, the standard form of XDA was used, along with a very simple algorithm that merely randomizes the controllable features within their domains without any further consideration.

All experiments have been executed on a commodity hardware laptop running Windows 11 Home version 23H2 equipped with an Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz-1.50 GHz (4 cores) and 8GB RAM.

### 3.1. Results

**RQ1** The process begins with the selection of 200 assignments to the variables in their semantic space, with each assignment representing an operational condition that leads to the violation of one or more requirements. Two metrics were used to compare the different solution search methodologies and to answer RQ1.

**Metric 1 (Satisfaction likelihood):** Given a requirement  $R_i$ , the current value of observable features  $\hat{O}F$ , and an adaptation decision  $\hat{C}F^*$ , the satisfaction likelihood is the probability estimation for the class label “true” as provided by the corresponding black-box classifier  $M_i$  under the operating condition  $(\hat{C}F^*, \hat{O}F)$ , that is,

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

**Metric 2 (Success rate):** Given a requirement  $R_i$ , and a set of operating conditions  $X$ , the success rate is defined as the number of elements  $(\hat{C}F^*, \hat{O}F) \in X$  such that the simulation of the subject under the condition  $(\hat{C}F^*, \hat{O}F)$  fulfills  $R_i$ , divided by the total number of elements in  $X$ .

The success rate was chosen as it reflects the outcome of empirical observations from actual simulations rather than relying solely on model predictions. The use of XDA and XDAv2 requires defining the size  $k$  of the neighbors. A small  $k$  reduces both memory and time costs but also decreases effectiveness, as it limits the exploration space and thus results in fewer solutions to evaluate. So it was therefore decided to set  $k$  equal to 10 because it is a value that expresses the balance between the two requests mentioned above. Exclusively for XDAv2, the Faiss library was introduced for the K-nearest neighbors search algorithm, replacing the KNeighborsClassifier from the sklearn library. This decision was made because Faiss performs efficient similarity search and clustering of large sets of dense

vectors, reducing both memory and time costs.

As previously mentioned, different multi-objective optimization algorithms were employed, and their parameters were fine-tuned to ensure that all search methods were on a similar level. For NSGA-III, the population size was set equal to the number of reference directions. As a termination condition, a convergence tolerance of  $10^{-6}$  in the constraints was defined, along with a maximum limit of 100 generations.

For FITEST, three tunable parameters were used: the population size was set equal to the population size of NSGA-III to maintain the baseline, the number of objectives, which corresponds to the number of requirements to be satisfied for that specific system, and the satisfaction probability threshold for the requirements. For all algorithms (XDA, NSGA-III, FITEST, and Random) I adopted the same probability threshold  $P$  equal to 0.8 for all requirements  $R_i$ .

From the set of tests performed, the most significant ones were selected, where the information can best convey the details. The chosen tests involve the rescue robot, the UAV, and autonomous driving, using standard constraints for each instance.

For Metric 1, the triplet rescue robot, UAV, and autonomous driving were selected using basic constraints, ensuring a solid foundation for testing the various algorithms.

First, the graphs representing the satisfaction likelihood of each model concerning the requirements were presented. For the rescue robot XDAv2 (SHAP and FI), the average confidence in two out of four requirements is the best among all algorithms, while for the remaining two it is comparable (Figure 3.1). For the UAV, the average confidence is comparable across all requirements when compared to other algorithms (Figure 3.2). Finally, the graph for autonomous driving was included, where the behavior of XDAv2 mirrors that of the UAV case, with the average confidence being comparable to the other algorithms for each requirement (Figure 3.3).

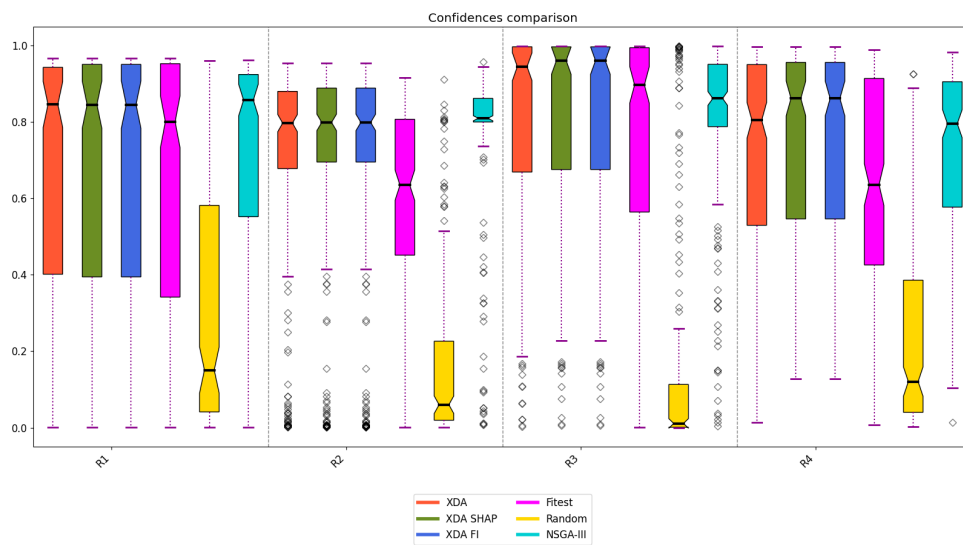


Figure 3.1: Rescue robot confidences comparison

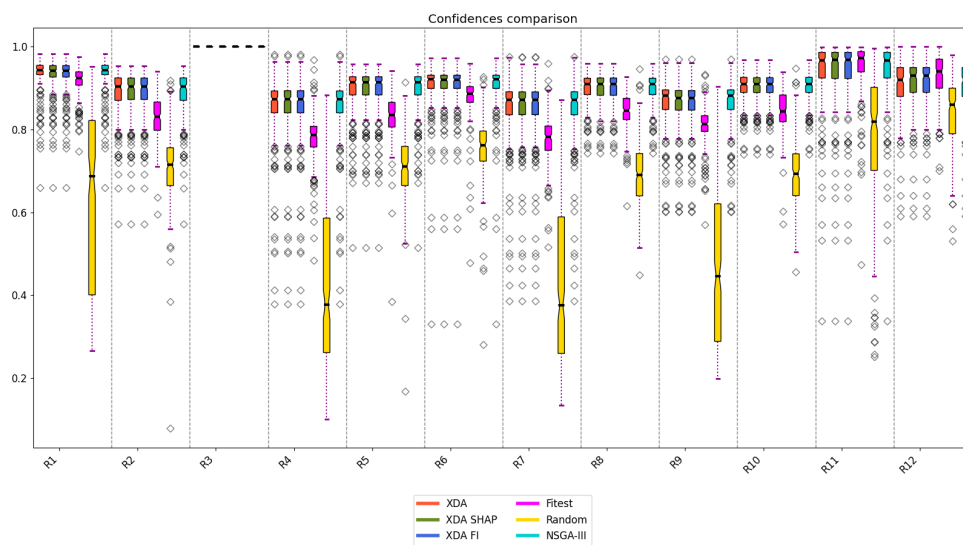


Figure 3.2: UAV confidences comparison

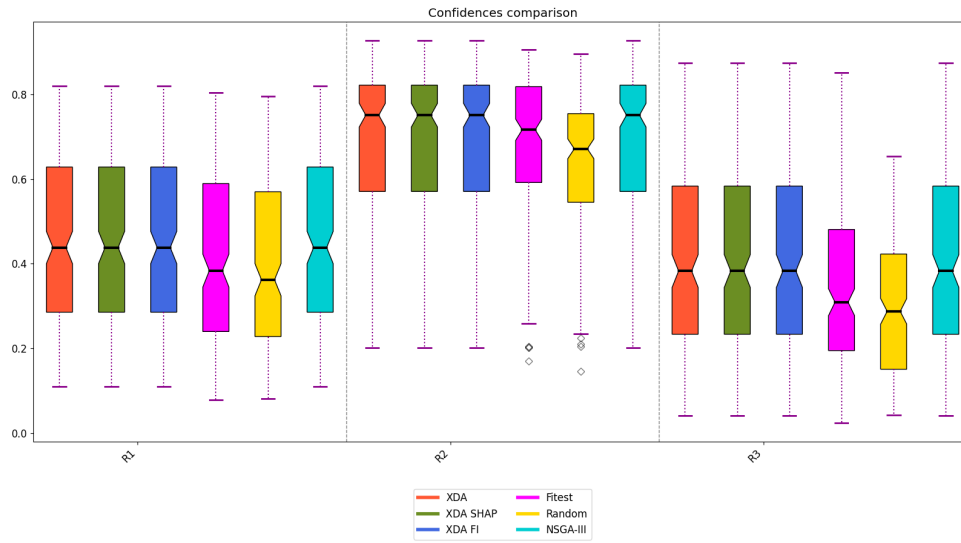


Figure 3.3: Autonomous driving confidences comparison

Next, the graphs show the overall success rate of the various algorithms across different cases (rescue robot, UAV, and autonomous driving). XDAv2 consistently has a success rate that is equal to or greater than FITEST and NSGA-III, while being equal to or lower than XDA base in every test instance (Figure 3.4, Figure 3.5, Figure 3.6).

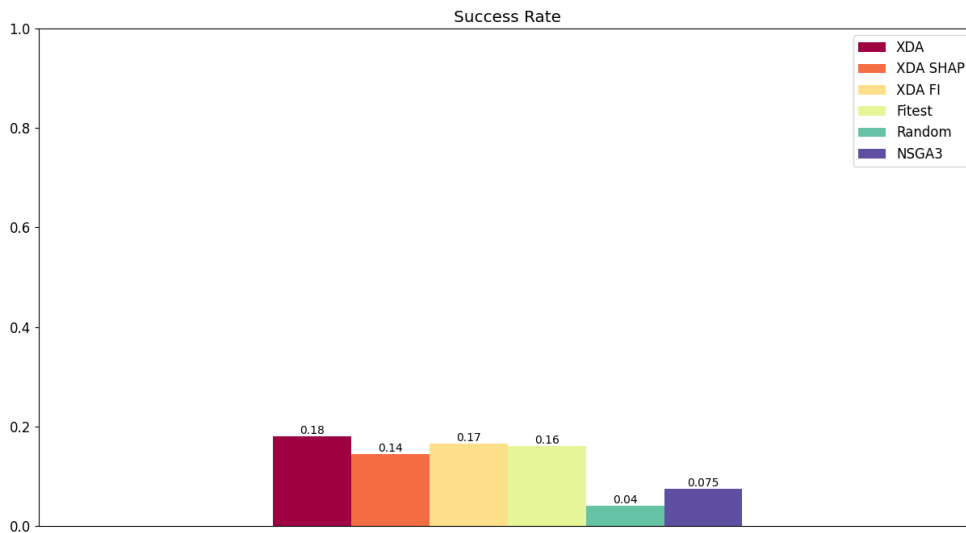


Figure 3.4: Rescue robot success rate

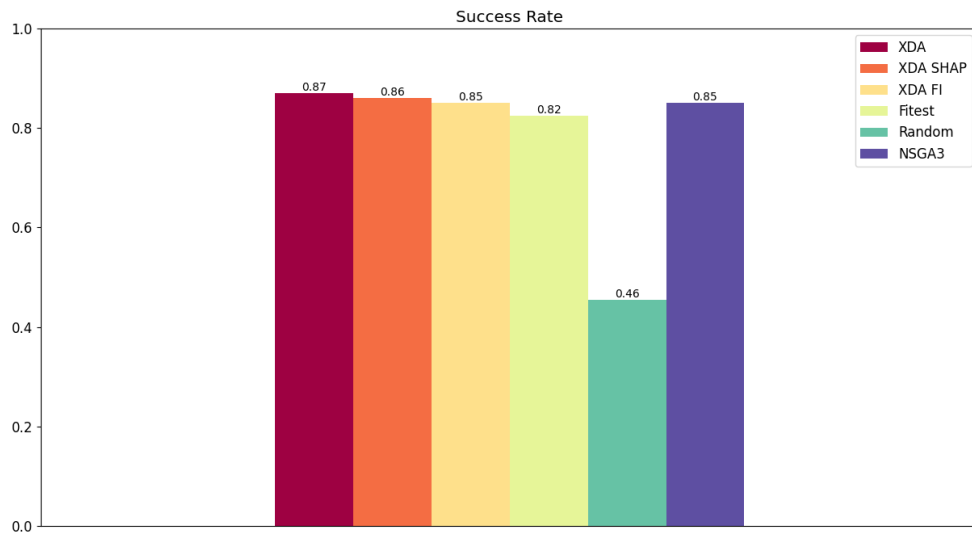


Figure 3.5: UAV success rate

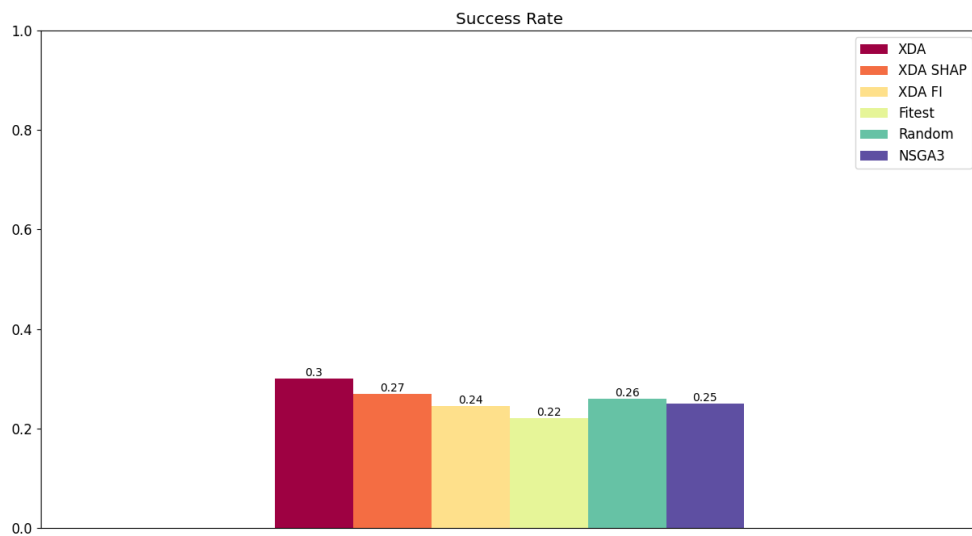


Figure 3.6: Autonomous driving success rate

**RQ1 summary.** Considering the satisfaction likelihood, XDAv2 produces better results compared to the baseline. However, in terms of success rate, there is no clear statistical difference between XDAv2 and the baseline methods.

**RQ2** To address RQ2, the execution time cost and memory cost of the online white-box optimizer were measured utilizing *profiler* for all 200 tests and then compared with the execution times of XDA, NSGA-III, Fitest, and Random.

To collect the execution time of each test, the Python *time* package was used. It records the start and end time of a test and calculates the total execution time utilizing

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

In this case, specific tests were also selected. Two types of test instances were compared: the first using the actual number of controllable features of the system, and the second using twice the number of controllable features as the first, where some observable features were chosen and made controllable (in Table 3.4, Table 3.6, Table 3.5 can be found domain description of systems).

Variable	Dimensions	Type	Domain
cruise speed	CF	continuous	[0, 5] m/s
camera quality	CF	categorical	{low, mid, high}
control responsiveness	CF	continuous	[10, 50] Mbit/s
illuminance	CF	continuous	[40, 120000] lux
power level	CF	integer	[0, 100] %
smoke intensity	CF	categorical	{none, thin, thick}
obstacle size	CF	continuous	[0, 2] m <sup>3</sup>
obstacle distance	CF	continuous	[0, 10] m
firm obstacle	OF	Boolean	Yes/No

Table 3.4: Features Rescue Robot Double

Variable	Dimensions	Type	Domain
car speed	CF	continuous	[5, 50] km/h
position x pedestrian	CF	continuous	[0, 10]
position y pedestrian	OF	continuous	[0, 10]
orientation	OF	integer	[-30, 30] degree
weather	OF	integer	[0, 2]
road shape	OF	integer	[0, 2]

Table 3.5: Features Autonomous Drive Double

Variable	Dimensions	Type	Domain
formation	CF	Boolean	Yes/No
flying speed	CF	continuous	[5, 50]
countermeasure	CF	Boolean	Yes/No
weather	CF	integer	[1, 4]
day time	CF	integer	[0, 23]
threat range	CF	continuous	[1000, 40000] m
number of threats	OF	integer	[1, 100]

Table 3.6: Features UAV Double

This was done to understand the effect of a higher workload when more features need to be modified. In the standard case, an initial speed-up of XDAv2 (SHAP or FI) compared to the base XDA can already be observed, as well as a significant speed-up compared to the FITEST and NSGA-III algorithms. When the workload increases, as in the cases of robot double and UAV double, the speed-up of XDAv2 (SHAP or FI) becomes even more pronounced, both compared to the base XDA and the FITEST and NSGA-III algorithms.

In the rescue robot double test instance, it was not possible to collect data for NSGA-III because its execution required excessive RAM usage, around 18.9 GB, which exceeded the 8 GB available on the system used for testing. Therefore, NSGA-III will not appear in the rescue robot double charts, and in the tables, its data will be marked as N.A. (not available).

The average execution time across 200 tests performed by each algorithm was used. Below are tables displaying the speed-ups in percentage for every instance of tests, followed by graphs representing box plots of the execution times.

The Table 3.7, Table 3.8 and Table 3.9 presents the average execution times for each algorithm, allowing for a direct comparison of raw performance. This data provides insight into the absolute execution times.

Algorithm	Mean Time Execution (s)	
	Rescue Robot	Rescue Robot Double
XDA	0.2781	2.5442
XDAv2 SHAP	0.0091	0.0349
XDAv2 FI	0.0092	0.0341
Fitest	6.8322	164.2751
NSGA-III	11.2145	N.A.
Random	0.0016	0.0017

Table 3.7: Mean Time Execution for Rescue Robot and Rescue Robot Double

Algorithm	Mean Time Execution (s)	
	UAV	UAV Double
XDA	0.2926	44.7612
XDAv2 SHAP	0.07615	0.2786
XDAv2 FI	0.0727	0.2788
Fitest	134.7402	646.0472
NSGA-III	0.2926	172.3393
Random	0.0233	0.0224

Table 3.8: Mean Time Execution for UAV and UAV Double

Algorithm	Mean Time Execution (s)	
	Autonomous Driving	Autonomous Driving Double
XDA	0.0034	0.0061
XDAv2 SHAP	0.0018	0.0022
XDAv2 FI	0.0018	0.0022
Fitest	0.3223	1.0567
NSGA	0.0034	0.1589
Random	0.0006	0.0005

Table 3.9: Mean Time Execution for Autonomous Driving and Autonomous Driving Double

In all three cases, XDAv2 does not exceed the order of  $10^{-2}$  seconds, while XDA averages around  $10^{-1}$  seconds. Fitest and NSGA-III, on the other hand, are in the order of seconds for the test instances with the standard number of controllable features (4 for the rescue robot, 3 for the UAV, and 1 for autonomous driving). In the case where the controllable features are doubled (8 for the rescue robot, 6 for the UAV, and 2 for autonomous driving), XDAv2 still maintains an average that does not exceed the order of  $10^{-1}$  seconds. In contrast, XDA is always under the minute while Fitest and NSGA-III reach times on the order of minutes.

The Table 3.10 shows the percentage speedups achieved by applying XDAv2 SHAP and FI compared to the execution time of a baseline version. The percentage speedup measures the increase in efficiency, calculated as the percentage reduction in execution time relative to the reference version. Higher values indicate a more significant performance improvement, negative values indicate a worsening of performance.

Algorithm	XDAv2 SHAP Speed-Up (%)	XDAv2 FI Speed-Up (%)
XDA	2953.33%	2938.43%
Fitest	74900.70%	74534.57%
NSGA-III	123007.02%	122406.05%
Random	-82.35%	-82.44%
<b>Robot Double</b>		
XDA	7194.45%	7353.18%
Fitest	470885.34%	481134.06%
NSGA-III	N.A.	N.A.
Random	-95.24%	-95.13%

Table 3.10: Percentage speed-up rescue robot

Algorithm	XDAv2 SHAP Speed-Up (%)	XDAv2 FI Speed-Up (%)
XDA	284.30%	302.04%
Fitest	176838.29%	185007.01%
NSGA-III	284.30%	302.04%
Random	-69.31%	-67.89%
<b>UAV Double</b>		
XDA	15961.63%	15951.35%
Fitest	231720.51%	231572.12%
NSGA-III	61740.35%	61700.76%
Random	-91.93%	-91.93%

Table 3.11: Percentage speed-up UAV

Algorithm	XDAv2 SHAP Speed-Up (%)	XDAv2 FI Speed-Up (%)
XDA	86.94%	90.78%
Fitest	17868.99%	18237.64%
NSGA-III	86.94%	90.78%
Random	-69.25%	-68.62%
<b>Drive Double</b>		
XDA	173.20%	171.38%
Fitest	47388.46%	47073.35%
NSGA-III	7042.06%	6994.67%
Random	-76.51%	-76.67%

Table 3.12: Percentage speed-up autonomous driving.

Observing the results, there is a clear trend in the speed-up of XDAv2 SHAP and FI, which increases with the number of controllable features and the number of requirements. For instance, using XDAv2 SHAP for comparison, in the case of the UAV and UAV double examples (with 3 controllable features and 12 requirements, and 6 controllable features and 12 requirements, respectively), the speed-up over XDA increases from 284.30% to 15,961.63%. Similarly, for the rescue robot and rescue robot double cases (with 4 controllable features and 4 requirements, and 8 controllable features and 4 requirements, respectively), the speed-up over XDA rises from 2,953.33% to 7,194.45%.

To visualize the distribution of execution times for each of the 200 tests per algorithm, a box plot was used, as it allows both the distribution of samples and their mean distribution to be observed. For each test instance (e.g., rescue robot, rescue robot double, UAV, etc.), each algorithm is represented by a box, except for the rescue robot double instance, which does not include any NSGA-III data due to excessive RAM usage on the system used for testing.

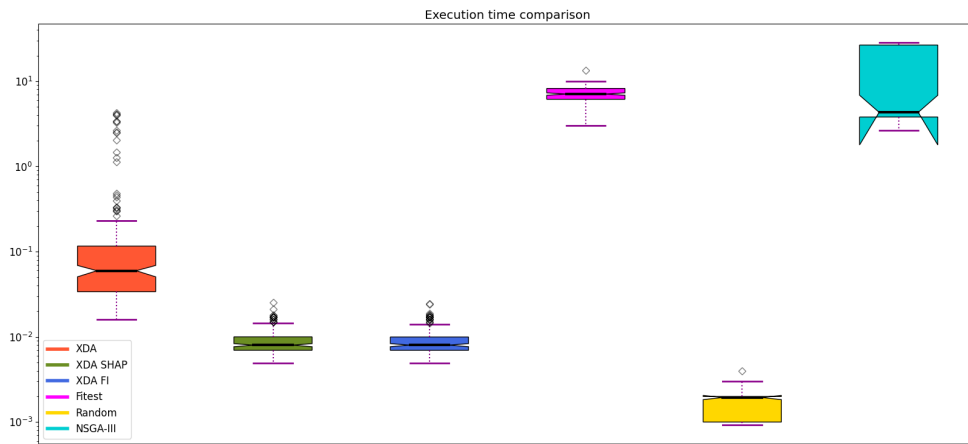


Figure 3.7: Rescue robot time comparison

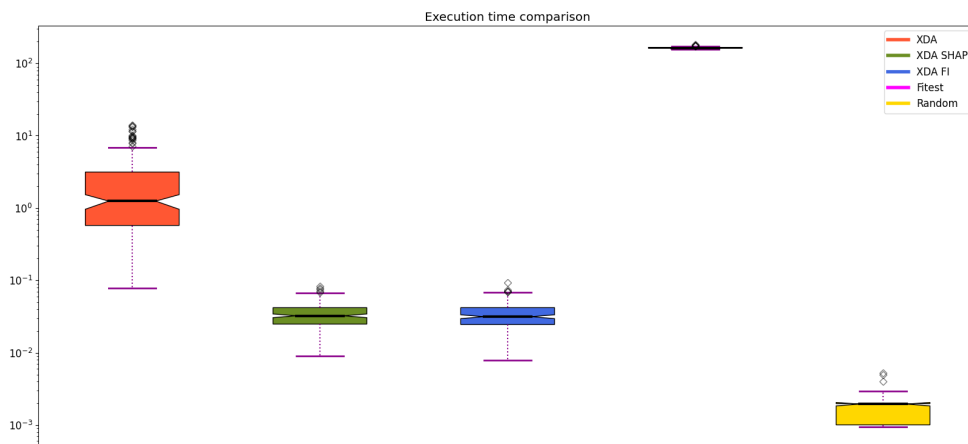


Figure 3.8: Rescue robot double time comparison

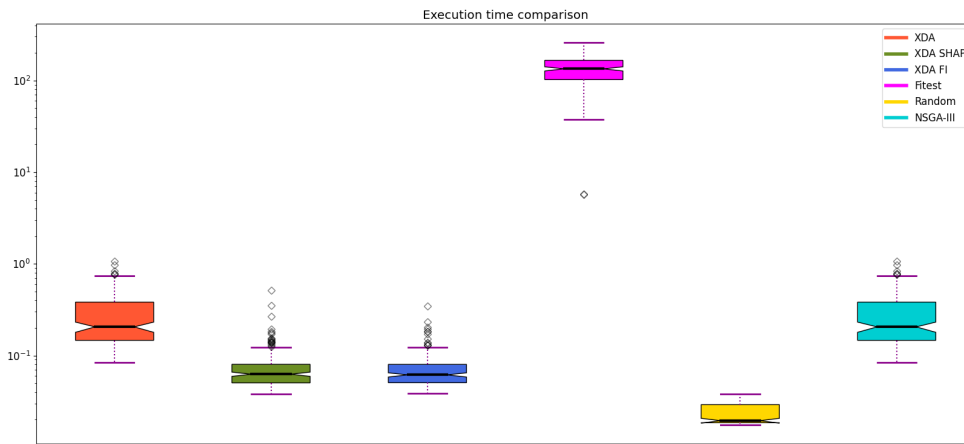


Figure 3.9: UAV time comparison

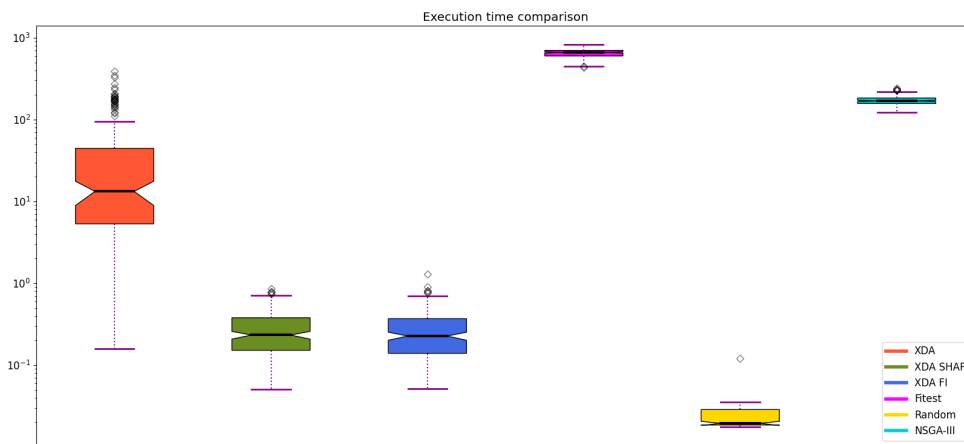


Figure 3.10: UAV double time comparison

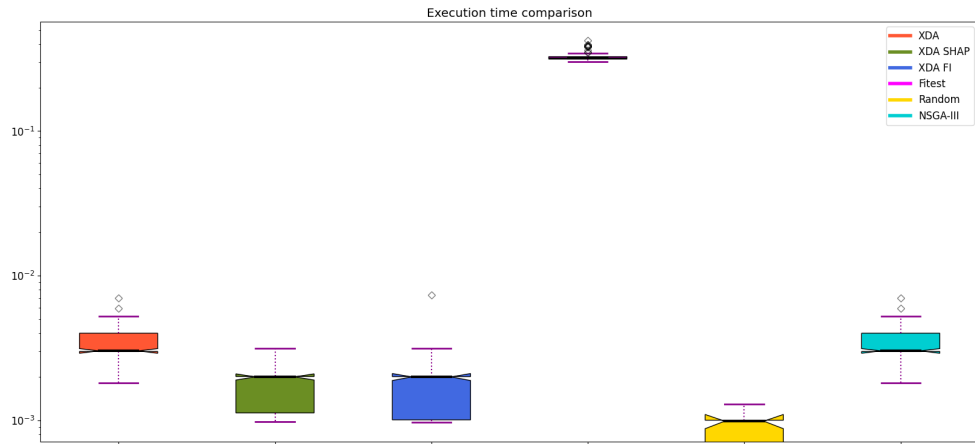


Figure 3.11: Autonomous driving time comparison

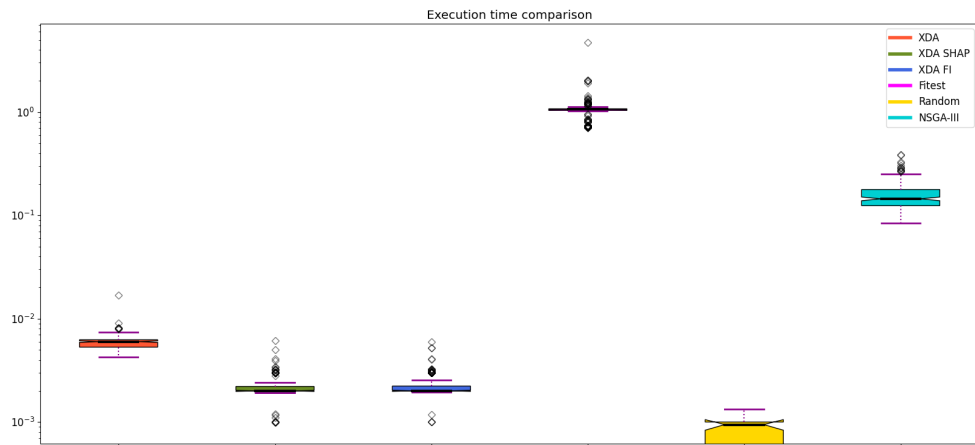


Figure 3.12: Autonomous driving double time comparison

Images from Figure 3.7 to Figure 3.12 are consistent with the data presented for speed-up; in fact, when observing the mean distribution, the one for XDAv2 SHAP and FI is the lowest among all algorithms except for the Random algorithm.

For memory usage, the Python tracemalloc package was employed, which records the peak of allocated memory for each test.

**Metric 4:** Peak memory allocated by the algorithm during its adaptation search (megabyte).

The following tables (Table 3.13, Table 3.14, Table 3.15) present a comparative analysis of memory allocation percentages across all algorithms under all test instances, comparing memory allocation of XDAv2 SHAP and FI with all other algorithms. The memory allocation performance is evaluated in various environments: the rescue robot, UAV, and autonomous driving scenarios. This analysis aims to highlight the efficiency and computational overhead each algorithm incurs under these conditions. Negative values indicate that XDAv2 has a smaller memory allocation and positive values indicate a bigger memory allocation relative to each algorithm (XDA, Fitest, Random, NSGA-III).

<b>Algorithm</b>	<b>XDAv2 SHAP</b>	<b>XDAv2 FI</b>
XDA	-89.01%	-89.01%
Fitest	-99.23%	-99.23%
Random	112.83%	112.77%
NSGA-III	-99.59%	-99.59%
<b>Robot Double</b>		
XDA	9.73%	9.73%
Fitest	-99.74%	-99.74%
Random	3028.28%	3028.17%

Table 3.13: Percentage comparison of allocated memory rescue robot

<b>Algorithm</b>	<b>XDAv2 SHAP</b>	<b>XDAv2 FI</b>
XDA	-27.1%	-26.62%
Fitest	-97.93%	-97.92%
Random	12.98%	13.72%
NSGA-III	-89.08%	-89.01%
<b>UAV Double</b>		
XDA	-96.23%	-96.24%
Fitest	-99.40%	-99.40%
Random	37.73%	37.55%
NSGA-III	-99.98%	-99.98%

Table 3.14: Percentage comparison of allocated memory UAV

<b>Algorithm</b>	<b>XDAv2 SHAP</b>	<b>XDAv2 FI</b>
XDA	-76.72%	-76.72%
Fittest	-99.10%	-99.10%
Random	323.10%	323.07%
NSGA-III	-62.71%	-62.71%
<b>Drive Double</b>		
XDA	-81.26%	-81.26%
Fittest	-98.85%	-98.85%
Random	699.73%	699.73%
NSGA-III	-81.81%	-81.81%

Table 3.15: Percentage comparison of allocated memory autonomous driving

Overall, XDAv2 SHAP and XDAv2 FI generally show strong memory efficiency, with negative percentages across most scenarios. This trend demonstrates their ability to manage memory more effectively compared to other algorithms. For instance, in the rescue robot scenario, both XDAv2 methods achieve nearly 89% less memory allocation relative to other approaches, and this efficiency is consistent in the UAV and autonomous driving cases as well. Only in rescue robot double there is a bigger memory allocation for XDAv2 SHAP and FI more precisely of 9.73%. Fittest and NSGA-III also achieve substantial memory savings, though NSGA-III is not feasible in the "Robot Double" scenario due to excessive memory demands that exceed system capacity. The random algorithm is the most efficient one because it simply allocates one array and modifies all values of controllable features randomly.

To clarify the results presented in the tables above, graphs have been included to show the peak memory allocated by each algorithm across each of the 200 tests performed, with each algorithm represented by a separate line (Figure 3.13, Figure 3.15, Figure 3.17).

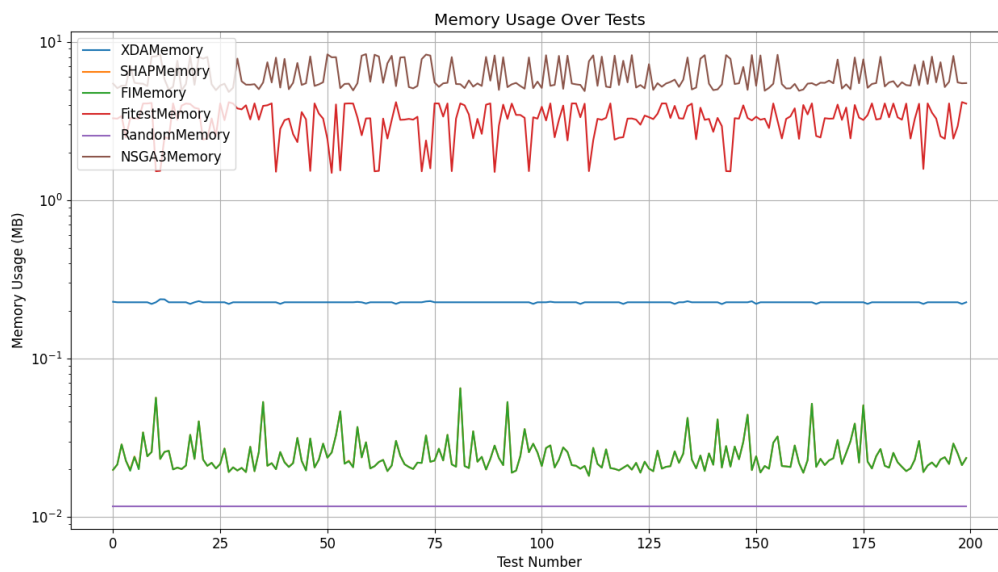


Figure 3.13: Rescue robot peak memory per test



Figure 3.14: Rescue robot double peak memory per test



Figure 3.15: UAV peak memory per test



Figure 3.16: UAV peak memory per test



Figure 3.17: Autonomous driving peak memory per test

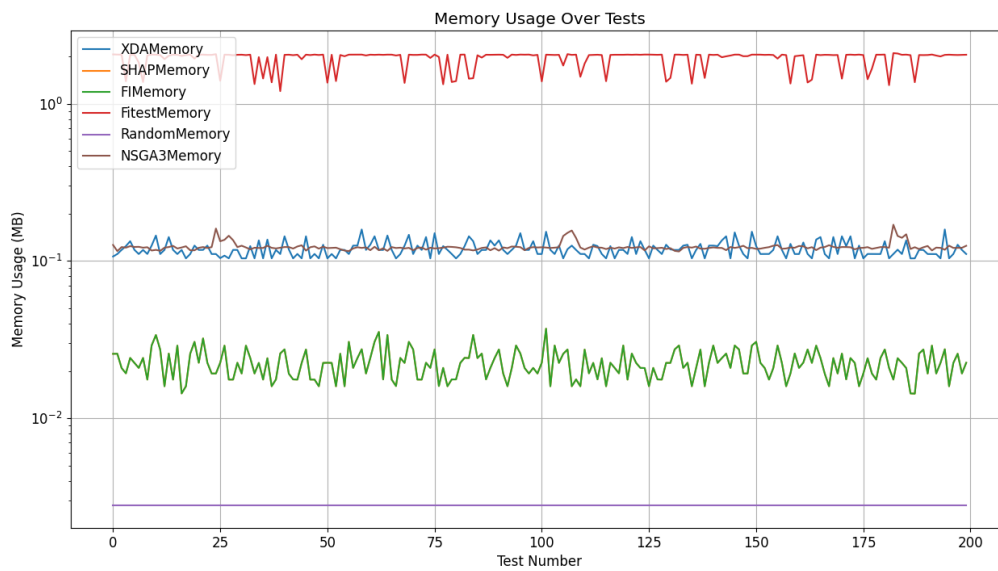


Figure 3.18: Autonomous driving double peak memory per test

**RQ2 summary.** The speedup of XDAv2 compared to the baselines reaches up to 18776.81% over XDA, 169933.72% over FITEST, and 26096.41% over NSGA-III. In terms of memory usage, the average cost of XDAv2 showed a reduction of 60.1% compared to XDA, 99.04% compared to FITEST, and 86.63% compared to NSGA-III.

**RQ3** After profiling the online white-box optimizer, the next step was to profile the offline pre-processor, which builds the knowledge base by creating the PDPs and SPDPs, as well as performing feature ranking (using SHAP and permutation feature importance). This profiling was carried out to observe the memory and time costs during the system deployment, allowing for an assessment of whether this phase is resource-intensive. To obtain sufficient data and reinforce the analysis, it was decided to construct PDP and SPDP 20 times, thus generating multiple cases for the same profiling, both in terms of execution time and peak memory usage.

In the offline phase, immediately after model construction for the requirements and their training, the models are passed to a function that constructs a PDP and an SPDP for each requirement and each controllable feature (CF). For each construction, both execution time and peak memory allocation were profiled.

**Metric 5:** Execution time of creation of PDPs and SPDPs for every machine learning model (seconds).

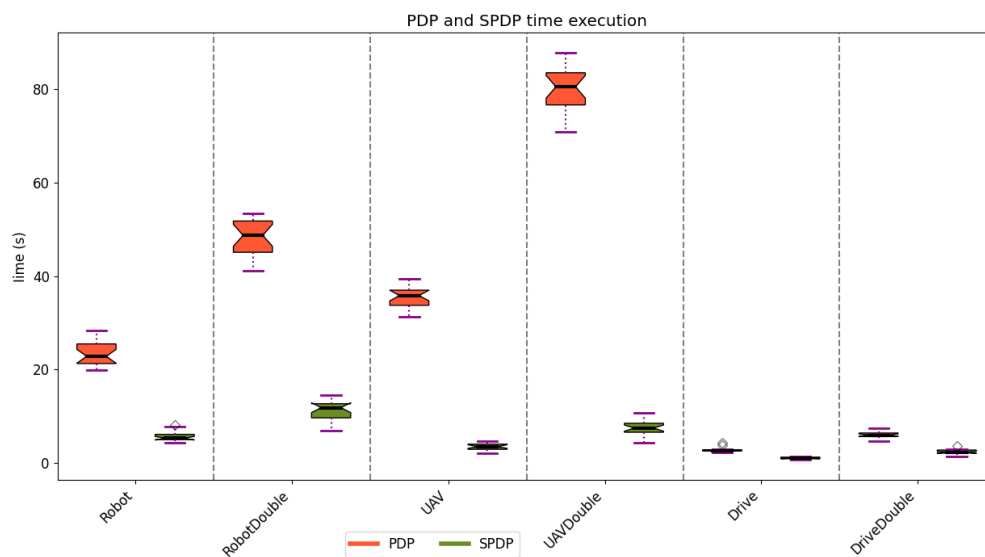


Figure 3.19: PDP and SPDP execution time

From Figure 3.19, it can be observed that the greater the number of features and requirements (e.g., UAVDouble), the longer the execution time. This can lead to a slowdown during system deployment, especially when using a highly complex system. It is important to note that this phase is one of the most critical, as it reveals how feature variations correspond to the fulfillment of the requirements.

**Metric 6:** Peak memory allocated by the creation of PDPs and SPDPs for every machine learning model (megabyte).

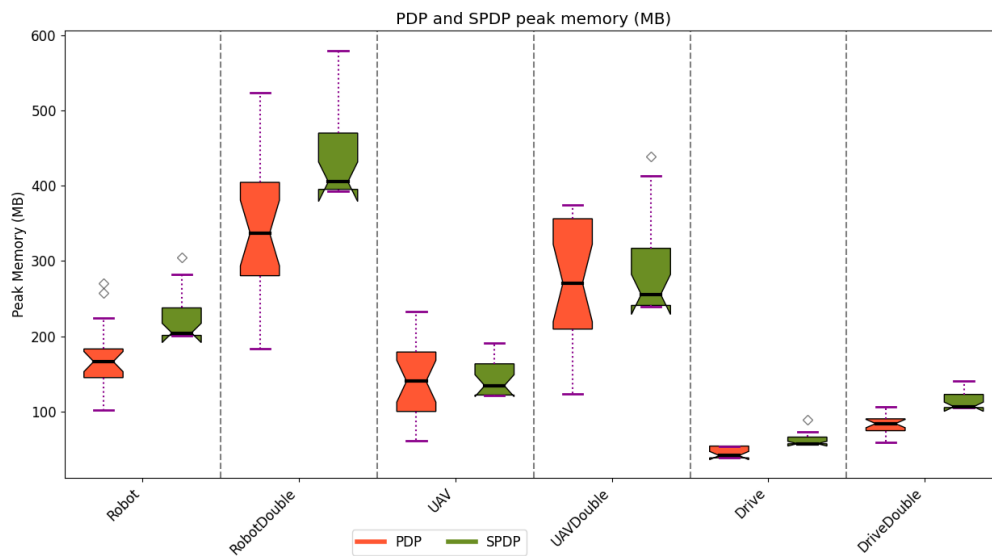


Figure 3.20: PDP and SPDP peak memory usage

After the PDP and SPDP construction, the feature ranking phase begins, using the controllable features (CFs) and the models representing the requirements. For each requirement, both SHAP and permutation feature importance perform the ranking of each CF, and in this phase, execution times and peak memory allocations were recorded for each requirement.

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

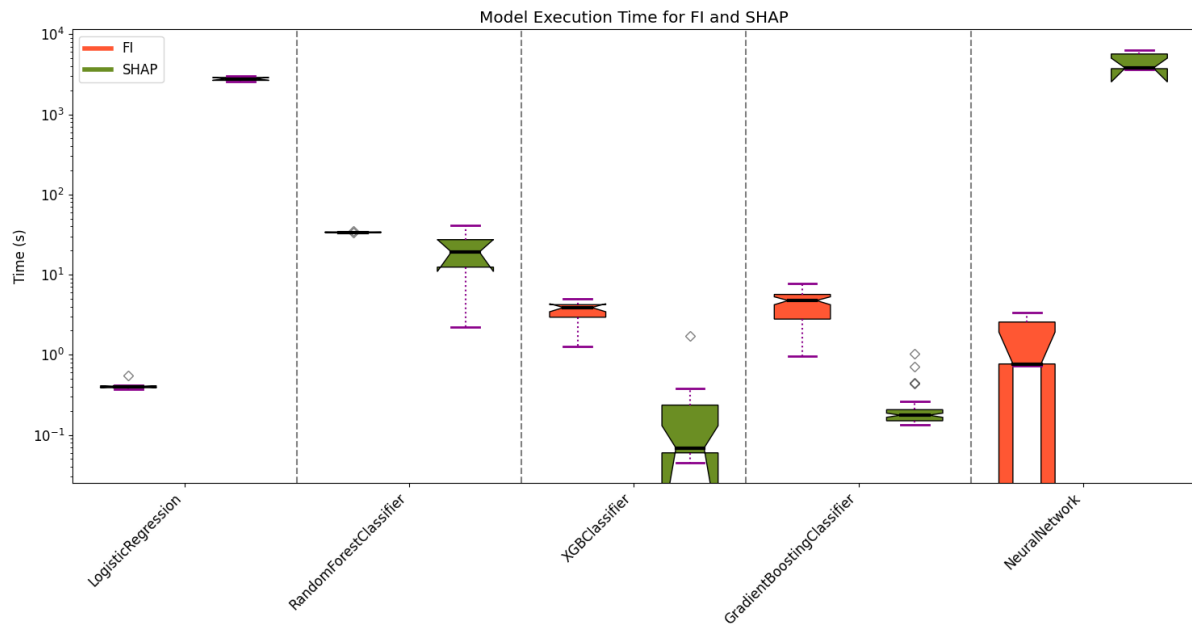


Figure 3.21: SHAP and FI execution time

From Figure 3.21, it can be observed that SHAP has a significantly higher execution time compared to permutation feature importance for the Logistic Regression and Neural Network models, while it remains comparable for the other models.

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

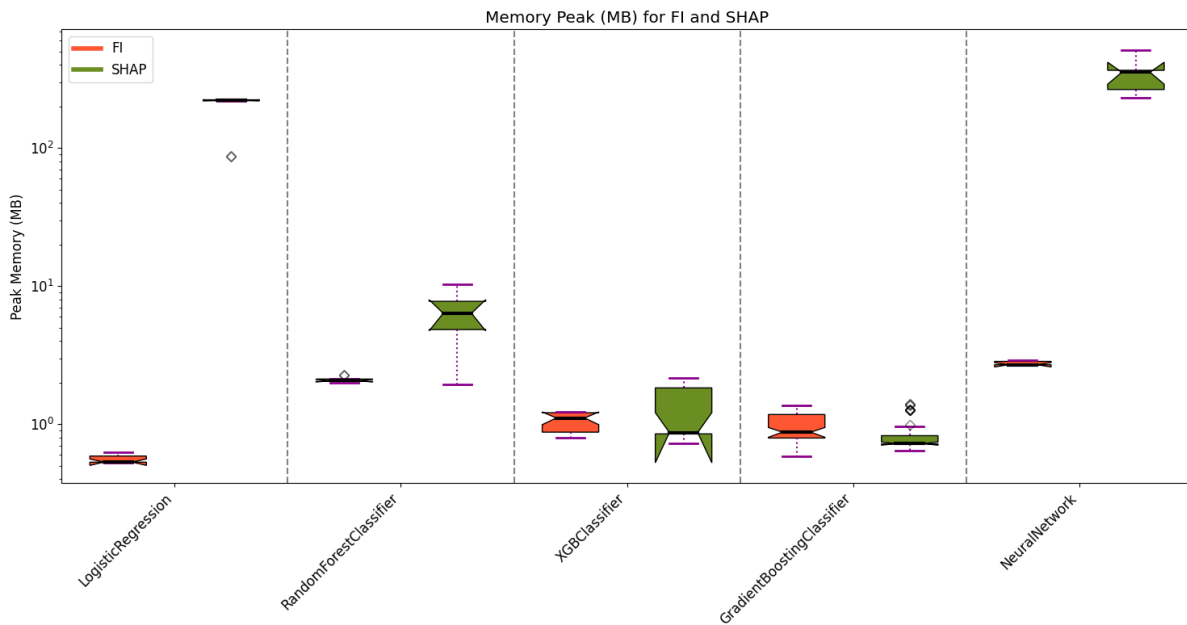


Figure 3.22: SHAP and FI peak memory allocation

The memory usage also reflects the previous results, with a noticeable peak for SHAP in the Logistic Regression and Neural Network models, while it remains comparable for the other models.

**RQ3 summary.** As the number of controllable features and requirements increases, both execution time and memory allocation also increase, due to the construction of PDPs and SPDPs as well as the feature ranking algorithms, SHAP and permutation feature importance. Performance degradation is observed only for SHAP when analyzing Logistic Regression and Neural Network models.

## 4 | Conclusions

This thesis presented XDAv2 an approach which extends XDA and embeds model-agnostic interpretable machine learning within a MAPE-K control loop. This approach is based on two key components: an offline pre-processor and an online white-box optimizer designed to enhance the interpretability and explainability of the system model. The XDAv2 extension also includes an additional module to the offline pre-processor, feature ranking, which uses SHAP and permutation feature importance algorithms to classify the most significant features impacting model outputs that represent system requirements. Finally, to gather data on execution times and memory usage of the offline pre-processor and the online white-box optimizer, a *profiler* was added to the self-adaptive system.

To obtain data for verifying and evaluating results, 18 types of empirical experiments were conducted, each consisting of 200 tests across three experimental scenarios (rescue robot, UAV, and autonomous driving). Subsequently, constraints were modified in each scenario to make them progressively more stringent across three levels (no label, v2, v3). For each level, a parallel test was conducted, characterized by doubling the controllable features.

The tests demonstrated that XDAv2 is comparably effective relative to XDA and more effective than the selected baseline methods (i.e., FITEST, NSGA-III). The cost of the offline pre-processor is negligible, except SHAP, which causes significant slowdowns for some models. The speedup of XDAv2 compared to the baselines reaches up to 18,776.81% over XDA, 169,933.72% over FITEST, and 26,096.41% over NSGA-III.. In terms of memory usage, the average cost showed a reduction of 60.1% compared to XDA, 99.04% compared to FITEST, and 86.63% compared to NSGA-III.

The strengths of XDAv2, namely, its temporal speed-up and memory efficiency—could play a significant role, given that average execution times do not exceed  $10^{-1}$  seconds (or  $10^{-2}$  seconds when considering only tests with the deployment-level number of controllable features) and that memory savings reach approximately half that of XDA and around 100% compared to FITEST and NSGA-III. Even when using increasingly complex models with higher numbers of controllable features and requirements, XDAv2 demonstrated

scalability, maintaining a less pronounced increase in execution time compared to the baseline. For the rescue robot scenario, execution time increased by 9.15% for XDA but only by 3.84% for XDAv2 relative to the base model with deployment-level controllable features; for the UAV scenario, execution time increased by 152.98% for XDA and by only 3.66% for XDAv2; and for autonomous driving, XDA showed a 1.79% increase compared to just 1.22% for XDAv2.

At the time of deployment, the steps contained within the offline pre-processor must be executed, which can result in a significant overhead as the number of controllable features and requirements increases, i.e., when the analyzed system becomes more complex.

## Bibliography

- [1] R. B. Abdesslem, A. Panichella, S. Nejati, L. C. Briand, and T. 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. ISBN 9781450359375. doi: 10.1145/3238147.3238192. URL <https://doi.org/10.1145/3238147.3238192>.
- [2] J. Blank and K. Deb. pymoo: Multi-objective optimization in python. *IEEE Access*, 8:89497–89509, 2020.
- [3] A. P. Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern Recognition*, 30(7):1145–1159, 1997. ISSN 0031-3203. doi: [https://doi.org/10.1016/S0031-3203\(96\)00142-2](https://doi.org/10.1016/S0031-3203(96)00142-2). URL <https://www.sciencedirect.com/science/article/pii/S0031320396001422>.
- [4] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Di Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle. *Software Engineering for Self-Adaptive Systems: A Research Roadmap*, pages 1–26. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-02161-9. doi: 10.1007/978-3-642-02161-9\_1. URL [https://doi.org/10.1007/978-3-642-02161-9\\_1](https://doi.org/10.1007/978-3-642-02161-9_1).
- [5] K. Deb. *Multiobjective Optimization Using Evolutionary Algorithms*. Wiley, New York. 01 2001.
- [6] K. Deb and H. Jain. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: Solving problems with box constraints. *IEEE Transactions on Evolutionary Computation*, 18(4):577–601, 2014. doi: 10.1109/TEVC.2013.2281535.
- [7] M. Douze, A. Guzhva, C. Deng, J. Johnson, G. Szilvasy, P.-E. Mazaré, M. Lomeli,

- L. Hosseini, and H. Jégou. The faiss library, 2024. URL <https://arxiv.org/abs/2401.08281>.
- [8] A. Fisher, C. Rudin, and F. Dominici. All models are wrong, but many are useful: Learning a variable’s importance by studying an entire class of prediction models simultaneously, 2019. URL <https://arxiv.org/abs/1801.01489>.
- [9] P. S. Foundation. *The Python Standard Library: cProfile*, 2024. URL <https://docs.python.org/3/library/profile.html#module-cProfile>. Accessed: 2024-09-19.
- [10] P. S. Foundation. *The Python Standard Library: tracemalloc*, 2024. URL <https://docs.python.org/3/library/tracemalloc.html>. Accessed: 2024-09-19.
- [11] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10): 46–54, 2004. doi: 10.1109/MC.2004.175.
- [12] C. Molnar. *Interpretable Machine Learning*. BOOKDOWN, 2024.
- [13] F. R. Negri, N. Nicolosi, M. Camilli, and R. 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. ISBN 9798400705854. doi: 10.1145/3643915.3644085. URL <https://doi.org/10.1145/3643915.3644085>.
- [14] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [15] D. 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. doi: 10.1109/FAS-W.2018.00012.

## List of Figures

1.1	Self-adaptive systems model. . . . .	6
2.1	MAPE-K Loop with XDA . . . . .	21
2.2	MAPE-K Loop with XDAv2 . . . . .	22
2.3	SHAP features ranking requirement 1 rescue robot . . . . .	24
2.4	SHAP features ranking requirement 2 rescue robot . . . . .	25
2.5	SHAP features ranking requirement 3 rescue robot . . . . .	25
2.6	SHAP features ranking requirement 4 rescue robot . . . . .	26
2.7	climbing step. [13] . . . . .	28
2.8	descending step. [13] . . . . .	28
3.1	Rescue robot confidences comparison . . . . .	37
3.2	UAV confidences comparison . . . . .	37
3.3	Autonomous driving confidences comparison . . . . .	38
3.4	Rescue robot success rate . . . . .	38
3.5	UAV success rate . . . . .	39
3.6	Autonomous driving success rate . . . . .	39
3.7	Rescue robot time comparison . . . . .	45
3.8	Rescue robot double time comparison . . . . .	45
3.9	UAV time comparison . . . . .	46
3.10	UAV double time comparison . . . . .	46
3.11	Autonomous driving time comparison . . . . .	47
3.12	Autonomous driving double time comparison . . . . .	47
3.13	Rescue robot peak memory per test . . . . .	50
3.14	Rescue robot double peak memory per test . . . . .	50
3.15	UAV peak memory per test . . . . .	51
3.16	UAV peak memory per test . . . . .	51
3.17	Autonomous driving peak memory per test . . . . .	52
3.18	Autonomous driving double peak memory per test . . . . .	52
3.19	PDP and SPDP execution time . . . . .	53

3.20 PDP and SPDP peak memory usage . . . . .	54
3.21 SHAP and FI execution time . . . . .	55
3.22 SHAP and FI peak memory allocation . . . . .	56

## List of Tables

3.1	Features Rescue Robot . . . . .	33
3.2	Features UAV . . . . .	34
3.3	Features Autonomous Drive . . . . .	34
3.4	Features Rescue Robot Double . . . . .	40
3.5	Features Autonomous Drive Double . . . . .	40
3.6	Features UAV Double . . . . .	41
3.7	Mean Time Execution for Rescue Robot and Rescue Robot Double . . . .	42
3.8	Mean Time Execution for UAV and UAV Double . . . . .	42
3.9	Mean Time Execution for Autonomous Driving and Autonomous Driving Double . . . . .	42
3.10	Percentage speed-up rescue robot . . . . .	43
3.11	Percentage speed-up UAV . . . . .	43
3.12	Percentage speed-up autonomous driving. . . . .	44
3.13	Percentage comparison of allocated memory rescue robot . . . . .	48
3.14	Percentage comparison of allocated memory UAV . . . . .	48
3.15	Percentage comparison of allocated memory autonomous driving . . . . .	49



## List Of Symbols

<b>NSGA-III</b>	
<b>Symbol</b>	<b>Description</b>
$N$	Population Size
$G$	Number of Generations
$\mathbf{Z}$	Reference Points
$P_0$	Initial Population
$Q_t$	Offspring at time $t$
$P_t$	Population at time $t$
$R_t$	Combined Population at time $t$
$F_k$	Non-Dominated Fronts
$P_G$	Final Non-Dominated Population

<b>FITEST</b>	
<b>Symbol</b>	<b>Description</b>
$\Omega$	Set of objectives
$A$	Archive
$F$	Feature Based Control System
$f_n$	Individual Feature
IntC	Integration Component
$P$	Current population
$T_c$	Test Cases
$Q$	Offspring population

<b>XDA v2</b>	
<b>Symbol</b>	<b>Description</b>
$M_n$	Classifier associated with requirement $n$
$n$	Number of requirements
$P$	Probability of satisfaction threshold
$CF_m$	Controllable Feature $m$
$m$	Number of Controllable Features
$OF_l$	Observable Features $l$
$l$	Number of Observable Features
$I_m$	Ideal values of controllable feature $m$
$PDP_m$	Partial Dependence Plot of controllable feature $m$
$SPDP_m$	Summary Partial Dependence Plot of controllable feature $m$
$k$	Number of neighbours
$\Delta$	Descent step

## Acknowledgements

This work represents the final step of a journey that has allowed me to challenge myself and find the strength to know who I am truly. It has helped me answer some questions while opening up new ones, brought me face-to-face with darkness while allowing me to discover the light. It introduced me to incredible new people while leading me to lose others, gave me the opportunity to visit places I had only dreamed of, and allowed me to take part in activities that made me feel part of something larger. It taught me that the most important battle is the one within ourselves. Above all, it showed me that without all of this, I would not be Paolo.

I would like to express my gratitude to my advisors, Matteo Camilli and Raffaella Mirandola, for guiding me and dedicating their time to me over these months; to my parents, Fiorella and Massimo, for giving without me ever needing to ask and for being by my side since the day I was born; and to my sister, Irene, for always believing in me and supporting me unconditionally.

A heartfelt thank you to my lifelong friends—Sara, Diego, Ludovico, Jacopo, Lorenzo, Diego, Matteo, Mattia, Sennosuke, Elia, Andrea, Silvia, Alessandro, Jan, Ginevra, Massimo, Alice, Francesca, Gaia, Giorgio, Stefania—who have accompanied me at every stage of this journey, never stopped believing in me and my abilities, and have been a light in moments when I found myself in the depths.

The Politecnico di Milano has made it possible for me to realize my dreams and to dream even bigger.

"Beautiful things,  
Beautiful people,  
This is what it's all about.  
If you think the world looks good,  
Guess what,  
It does."

