A Probabilistic Model Checking Approach to Self-adapting Machine Learning Systems

Maria Casimiro[1,2](✉), David Garlan[1], Javier Cámara[3], Luís Rodrigues[2], and Paolo Romano[2]

[1] Institute for Software Research, Carnegie Mellon University, Pittsburgh, PA, USA
[2] INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Lisboa, Portugal
maria.casimiro@tecnico.ulisboa.pt
[3] ITIS Software, Universidad de Málaga, Málaga, Spain

Abstract. Machine Learning (ML) is increasingly used in domains such as cyber-physical systems and enterprise systems. These systems typically operate in non-static environments, prone to unpredictable changes that can adversely impact the accuracy of the ML models, which are usually in the critical path of the systems. Mispredictions of ML components can thus affect other components in the system, and ultimately impact overall system utility in non-trivial ways. From this perspective, self-adaptation techniques appear as a natural solution to reason about how to react to environment changes via adaptation tactics that can potentially improve the quality of ML models (e.g., model retrain), and ultimately maximize system utility. However, adapting ML components is non-trivial, since adaptation tactics have costs and it may not be clear in a given context whether the benefits of ML adaptation outweigh its costs. In this paper, we present a formal probabilistic framework, based on model checking, that incorporates the essential governing factors for reasoning at an architectural level about adapting ML classifiers in a system context. The proposed framework can be used in a self-adaptive system to create adaptation strategies that maximize rewards of a multi-dimensional utility space. Resorting to a running example from the enterprise systems domain, we show how the proposed framework can be employed to determine the gains achievable via ML adaptation and to find the boundary that renders adaptation worthwhile.

Keywords: Machine-learning based systems · Self-adaptation · Probabilistic model checking · Architectural framework

1 Introduction

Machine learning (ML) is present in most systems we deal with nowadays and is not a trend that will vanish in the years to come. Like all other components in a

Support for this research was provided by Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program under Grant SFRH/BD/150643/2020 and via projects with references POCI-01–0247-FEDER-045915, POCI-01–0247-FEDER-045907, and UIDB/50021/2020.

A. Cerone et al. (Eds.): SEFM 2021 Workshops, LNCS 13230, pp. 317–332, 2022.
https://doi.org/10.1007/978-3-031-12429-7_23

system, ML components can fail or simply produce erroneous outputs for specific inputs [8,14,15]. This problem is exacerbated by the fact that the environments in which the ML components operate may be different from those that the component may have been trained on [31]. When such a situation occurs, the system is likely to suffer from a problem known as data-set shift [29]. Since ML components rely on input data to learn representations of the environment, data-set shift can cause accuracy degradations which ultimately affect system utility. Hence an important requirement for systems with ML components is to be able to engineer those systems in such a way as to be able to adapt the ML components when it is both possible and beneficial to do so.

Current approaches to architecture-based self-adaptive systems provide a useful starting point. Following the well-known MAPE-K pattern [17], a system is monitored to produce updated architectural models at run time, which can then be used to determine whether and how a system might be improved through the application of one or more tactics. In support of this approach, there are a variety of tactics that might be brought to bear on ML-based systems including model retraining [32], various incremental model adjustments [20], data unlearning [5], and transfer learning [16,27] techniques.

However, deciding both whether and how to adapt an ML-based system is non-trivial. In particular, typically there are costs as well as benefits for applying tactics. Determining whether the benefits of improving an ML component outweigh its costs involves considerations of timing, resources, expected impact on overall system utility, the anticipated environment of the system, and the horizon over which the benefits will be accrued. Moreover, in practice there is often considerable uncertainty involved in all of these factors.

This paper proposes a probabilistic framework based on model checking to reason, in a principled way, about the cost/benefits trade-offs associated with adapting ML components of ML-based systems. The key idea at the basis of the proposed approach is to decouple the problems of **i)** modelling the impact of adaptation on the ML model's quality (e.g., expected accuracy improvement after retrain) and **ii)** estimating the impact of ML predictions on system utility. The former is tackled by incorporating in the framework the key elements that capture relevant dynamics of ML models (e.g., the expected dependency between improvement of model's quality and availability of new training data). The latter is solved by expressing inter-component dependencies via an architectural model, enabling automatic-reasoning via model checking techniques.

We resort to a running example from the enterprise systems domain to showcase how to instantiate the proposed framework via the PRISM model checker. Finally, we present preliminary results that show how system utility can be improved through the adaptation of ML components.

The remainder of this document is organized as follows: Sect. 2 motivates the need for the proposed framework and highlights existing challenges; Sect. 3 presents the proposed framework and Sect. 4 shows how it can be applied. Section 5 evaluates the framework, Sect. 6 overviews related work, Sect. 7 discusses existing limitations and future work and Sect. 8 concludes the paper.

2 Motivation

In this work we focus on self-adaptation of ML-based systems which are composed of both ML and non-ML components. For example, fraud detection systems rely on ML models to output the likelihood of a transaction being fraudulent and on rule-based models (non-ML component) to decide whether to accept/block/review a transaction based on the ML's output [2]. Similarly, cloud configuration recommenders rely on ML models to select the platform (non-ML component such as a virtual machine) on which users should deploy their jobs. These recommenders are typically guided by user-defined objective functions such as minimizing execution time [1,6].

There are two key requirements associated with reasoning about self-adaptation of ML components. First, it is necessary to understand if and how ML predictions affect overall system utility. Second, it is necessary to estimate the costs and benefits of the available adaptation tactics. Let us now discuss the key challenges associated with each requirement.

i) Impact of Machine Learning Predictions. A key problem that needs to be addressed to enable automatic reasoning on the dynamics of ML-based systems is determining to what extent incorrect predictions will impact overall system utility. In fact, this is not only application but also context dependent. For example, in cloud configuration recommenders, when the relative difference in job execution speed between the available cloud configurations is low, ML mispredictions have little impact on system utility [6]. Similarly, in a fraud detection system, the impact of mispredictions is different in periods with higher volumes of transactions, in which it is critical to maximize accepted transactions, while accurately detecting fraud [2].
ii) Estimating Costs and Benefits of Adaptation Tactics. Predicting the time/cost and benefits of ML adaptation tactics is far from trivial. This prediction is strongly influenced both by the type of models and their settings (hyper-parameters and execution infrastructure), and by the input data employed in the adaptation process. For instance, in the case of a tactic that triggers the retrain of an ML model, the benefits of tactic execution are dependent on the data available for the process – data more representative of the current environment contributes to higher benefits. Differently, if the adaptation tactic consists of querying a human (human-in-the-loop tactic), the benefits are now dependent on human expertise. Their execution latency and economic cost are also likely to be different and affected by factors that are inherently tactic dependent, e.g., the retraining time is affected by the amount of available training data, whereas the latency of a human-in-the-loop tactic may depend on the complexity of the problem the human is required to solve.

In this work, we argue that by leveraging formal methods we can instantiate the problem of reasoning about the need for adaptation at a general architectural level. The framework we propose allows us to abstract away from system-specific issues and instead instantiate the decision of whether to adapt ML components

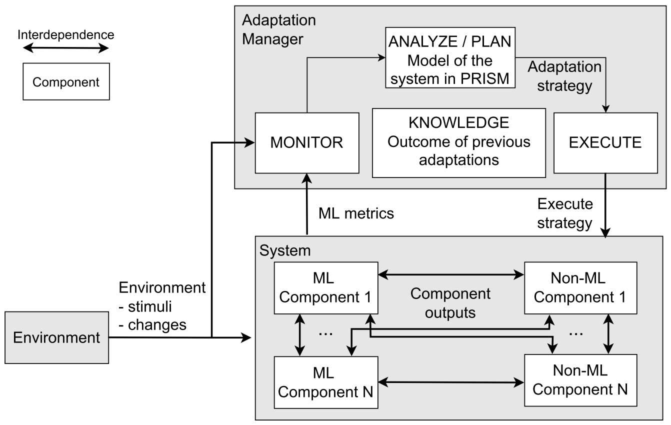


Fig. 1. Framework modules and inter-dependencies.

as a general decision that relies on the key factors of ML-based systems. The next section introduces our formal framework.

3 Framework for Self-adaptive ML-Based Systems

This section describes a generic framework that can be used to derive formal models of self-adaptive ML systems. The resulting models can then be utilized to enable automatic reasoning via probabilistic model checking tools such as PRISM [19]. In fact, the use of such tools in the self-adaptive systems (SAS) domain is not new [4,25,26]. Conceptually, the proposed framework can be regarded as a specialization of the frameworks already proposed in this field, which targets a specific class of "managed" systems: systems containing ML-based components. As such, in the following sections our focus will be on how to capture the most relevant dynamics of ML-components via abstract and generic models that can be easily extended and customized to specific use cases.

3.1 Architectural Overview

As in typical frameworks for SAS, our framework requires specifying the behavior of the following modules: environment, system, and the adaptation manager (Fig. 1). Next, we discuss how each of these modules is modelled in the proposed framework. For the ML component, we further describe its internal state, how it evolves, and the methods exposed by its interface to allow for inter-component interactions.

Environment. In the environment component, it is necessary to consider the types of stimuli to which the system responds/reacts. Additionally, since our goal is to reason about the impact of environment changes, these must also be modelled. Examples of environment stimuli are for instance the transactions that a fraud detection system has to classify or the jobs received by a scheduler.

Adaptation Manager. As in typical SAS, the adaptation manager contains a repository of adaptation tactics. However, and differently from previous work in this domain, we consider adaptation tactics that directly actuate over the ML component [7], such as retraining the ML component. Each adaptation tactic is specified by: (i) a precondition that triggers its execution and which generally depends on the state of the system and the environment; (ii) the effects on the targeted components. We model adaptation tactics as a tuple composed of tactic cost and tactic latency. This division allows us to study the impact of the different dimensions of tactics on overall system utility. For example, consider a retrain tactic. It has a latency associated, since retraining a model takes a non-negligible amount of time. If that tactic is executed in cloud environments it also has a monetary cost, which depends both on its latency, and on the underlying cloud platform selected for the execution. By leveraging model checking tools such as PRISM [19] we can thus explore alternative adaptation policies with the objective of identifying the one that, for example, maximizes system utility.

System. The key novelty of our framework is that it enables reasoning about the adaptation of ML-based systems, which we abstractly define as systems that comprise two types of components: ML-based and non-ML based components (Fig. 1). An ML-based component is used to encapsulate an ML-model that can be queried and updated (e.g., retrained). Non-ML based components are used to encapsulate the remaining functional components of the system being managed/adapted. Our framework is agnostic to the modelling of the application-dependent dynamics of non-ML based components, which can be achieved by resorting to conventional techniques already proposed in the SAS literature [4,25,26]. However, we require non-ML components to interact with ML components in two ways: i) pre-processing data to act as input to the ML component or using the ML component's outputs to perform some function ii) affecting system utility by adding negative/positive rewards upon completion of a task. For example, in an ML-based scheduling system, once a job completes its execution on the selected cloud platform (a non-ML based component), it triggers the accrual of a reward (e.g., dependent on the execution time of the job) on system utility.

As for modelling the ML components, our design aims to ensure the following key properties: **(i) generic** – designed to be applicable to offline and online learning, supervised, unsupervised and semi-supervised models, different types of ML models (e.g., neural networks, random forests); **(ii) tractable** – designed to be usable by a probabilistic model checker like PRISM, having a high level of abstraction to aid systematic analysis via model checking; **(iii) expressive** – designed to capture key dynamics of ML models that are general across ML models; **(iv) extensible** – designed to be easily extended to incorporate additional adaptation tactics and customized to capture application specific dynamics. The following section introduces the proposed modelling approach for ML components.

3.2 Machine Learning Components

We consider ML components that solve classification problems, i.e., whose possible outputs are defined over a discrete domain. We argue that this assumption is not particularly restrictive given that any regression problem (in which the ML model's output domain is continuous) can be approximated via a classification problem by discretizing the output domain. We abstractly formalize the behavior of an ML based component by specifying (i) its state, (ii) the set of events that change its state, (iii) the logic governing how the internal state evolves due to each possible event.

Machine Learning Component State. The state of an ML component is characterized by two elements: a confusion matrix and the set of new data (knowledge) which encodes information regarding the data accumulated so far by the ML component. The confusion matrix is a tabular way to represent the quality of a classifier. We opt for using the confusion matrix to abstract over the internal dynamics of the specific model being used while still capturing the quality of its predictions. More in detail, it is a matrix $n \times n$ where n represents the number of output classes. In cell (i, j) the confusion matrix maintains the probability for an input of class i to be classified by the model as belonging to class j. In fact, due to how it is constructed, it provides access to metrics such as false positives/negatives, which in turn constitute the basis to compute alternative metrics, like f-score or recall, that can be of interest for specific applications/domains (e.g., relevant for fraud detection systems [2]). Our framework supports the specification of multiple confusion matrices, which can be of interest when there are several types of input to an ML model (e.g., if a scheduler receives different job types, some easier to classify than others, this could be modelled by associating a different confusion matrix to each job type).

The second element of the ML component's state represents the knowledge maintained by the ML component. This is characterized by the data which the model has already used for training purposes, and the new data that is continually gathered during operation and that represents the current state of the environment the system is operating in. However, the representation of this knowledge is application dependent. While in simpler use-cases it may be enough to simply maintain a counter for the inputs that arrive in the system, in more complex scenarios the data gathered during execution may encode important information to characterize the environment (e.g., measures of dataset shift [29]). This knowledge can be used by the adaptation tactics when these are executed over the ML component.

Machine Learning Component Interface. In order for the other components in the system to interact with the ML component, we propose a general interface that enables this interaction. Generally, any ML component that supports adaptation requires three key methods: *query*, *update_knowledge*, and *retrain*. Clearly, new methods can be added to this interface to tailor the framework to specific application requirements. For example, to capture manipulations of the dataset (e.g., sub-sampling certain types of inputs) or to support further adaptation tactics (e.g., unlearning).

As the name suggests, *query* is used to solicit a prediction from the ML component. The internal behavior of the ML component when issuing predictions is abstracted away by reasoning only over the likelihoods specified in the confusion matrix. More precisely, the output event produced by executing a query is a probabilistic event, which can assume any of the possible output classes of the classifier, with the probability given by the classifier's confusion matrix.

The method *update_knowledge* should be called when the system has reacted to an event and thus there is new data to be accounted for. The framework can keep track either of the pair $\langle$*ML input, ML prediction*$\rangle$ or of the triple $\langle$*ML input, ML prediction, real output*$\rangle$. The selection of either option is domain dependent. For example, in the fraud detection domain, knowing the actual value of a transaction (legitimate or fraudulent) is not always possible [28]. The pair is thus required when this is the case. Finally, *retrain* corresponds to retraining the ML model resorting to the data stored in the ML component's state.

Any method has an associated cost and latency that directly impact system utility. These are captured by the framework as follows. The latency is used to determine after how many units of time the effects of each method are applied to the component (and to the system). The cost of executing each method (when the method has a cost) is discounted from the system's utility.

Machine Learning Component State Evolution. When any of the previously described interface methods is triggered, the state of the ML component can be altered. Since *query* consists only of asking the ML model for a prediction, it does not alter the component's state. *update_knowledge* changes the knowledge of the ML component by adding instances to that set. Finally, *retrain* changes both elements of the state: the confusion matrix and the knowledge are updated to reflect the execution of the adaptation tactic. In the case of a retrain adaptation tactic the data used for executing the tactic is updated in the knowledge such that it is no longer considered new data. At the same time, the confusion matrix is updated to reflect the current performance of the model after having been retrained. In fact, we propose a simple model that aims to capture the improvements to the confusion matrix given by the execution of a retrain adaptation tactic. The rationale behind the proposed model is that the larger the number of new samples seen since the last training (i.e., new data), the larger should be the expected reduction in the misclassification rate. Specifically, the confusion matrix should be updated as follows. The diagonal is incremented by a factor $\delta = (100 - cell_{ii}) * new\_data * impact\_factor$ that is proportional to the model's loss and to the amount of new data (e.g., number of new samples). The *impact_factor* allows for flexibility in different types of retrain (e.g., when the hyper-parameters of the model are also updated, the benefits may be higher). The remaining cells in the same row should then be updated as $cell_{ij} = cell_{ij} - \delta cell_{ij}/(1 - cell_{ii})$. The non-diagonal cells are thus reduced proportionally to δ while ensuring that no cell gets a value lower than 0,

Table 1. Visualization of an update to a confusion matrix. We assume *new_data* = 6 and *impactFactor*= 0.1 which yields $\delta = (100 - 95) \times 6 \times 0.1 = 3$. P1 and P2 stand for Platform 1 and Platform 2, respectively.

Pred. \ Real	P1	P2
P1	95	5
P2	5	95

(a) Initial confusion matrix.

P \ R	P1	P2
P1	95 + 3	$5 - \frac{3\times5}{(100-95)}$
P2	$5 - \frac{3\times5}{(100-95)}$	95 + 3

(b) Confusion matrix update.

P \ R	P1	P2
P1	98	2
P2	2	98

(c) Final confusion matrix.

and that the total reduction on non-diagonal cells is equal to δ. Table 1 provides an example of a confusion matrix being updated.

Dealing with Uncertainty. As shown by recent work in SAS, capturing uncertainty and including it when reasoning about adaptation contributes to improved decision making [4,25,26]. Uncertainty can affect a range of components, including non-ML components (e.g., the execution time of a job on a specific cloud platform is unknown), ML components (e.g., in the fraud domain, as there is no real time access to real labels of transactions, we cannot measure the *current* model's performance, but at most estimate it [28]), and adaptation tactics (e.g., with 90% probability retrain is expected to reduce the misclassification rate, however in the remaining 10% of the cases the misclassification rate remains the same). In the proposed framework, uncertainty regarding a specific component or event can be naturally integrated by defining the affected state of the component/event via discrete distributions built leveraging historical data. Uncertainty can thus be conveniently captured by expressing the outcome of an uncertain action (or state) via a probabilistic event.

4 Model Checking the Need for Adaptation

This section exemplifies, based on a running example, how to leverage the proposed framework to reason about whether to adapt ML components. By introducing in the formal model non-deterministic choices between the tactics available for execution, a model-checking based approach can be used to determine, at any time, which adaptation tactic to enact in order to maximize system utility. We implement our framework using the PRISM [19] tool, which allows to model non-deterministic phenomena via probabilistic methods such as Markov Decision Processes (MDPs) and generate optimal strategies for reward-based properties. PRISM has been extensively used by the literature on Self-Adaptive Systems (SAS) to reason about adaptation trade-offs [9,21,25,26].

We start by introducing a simple use-case, which was selected to exemplify the framework. The following sections then describe the modules of the PRISM model in more detail. Due to lack of space, we do not provide details about the implementation of the framework and of the running example in PRISM. However, we are working on a technical report that includes these details and will make it available in the near future.

Running Example. Consider a system that receives jobs and has to select a platform for them to execute. As it is often the case in practice, we assume that the execution time of a job on a given platform depends on the job's characteristics, i.e., it may execute faster on a platform than on another [1,6]. Each time a job completes, the system receives a fixed reward. As such, in a given period, the system will strive to complete as many jobs as possible by selecting the platform that can execute each incoming job in the shortest amount of time, so as to accrue the maximum benefits possible. For example, the system could receive different data analytic jobs with diverse characteristics (e.g. neural network (NN) training, data stream processing) [1,6]. The system then relies on an ML component to decide the best platform for a specific job to execute in. For instance, the training of a neural network can be offloaded to GPUs or CPUs. While both platforms allow the system to complete its task (i.e., execute the job) one platform may be more efficient (lower latency) than the other, thus allowing the system to complete more jobs in a given horizon. We are interested in scenarios in which the type of job generated by the environment is altered, for example due to data-set shift [29], thus leading the ML model to lose accuracy [18].

Machine Learning Adaptation. To equip the system with adaptation capabilities, so that it can deal with environment changes and accuracy fluctuations of the ML-based predictor, for instance due to unknown jobs (e.g., unseen NN topology), we consider that each time a new job arrives, the adaptation manager can decide between simply querying the current ML model (i.e., no adaptation/tactic *nop*) or adapting it (tactic *retrain*), in order to increase its accuracy and maximize the likelihood of executing the job in the preferred platform. The retrain adaptation tactic consists of incorporating additional training data in a new version of the model [32]. However, the execution of this tactic requires a non-negligible time interval for its effects to manifest themselves in the system, and has a monetary cost (e.g., if retrain is performed in the cloud) [1,6]. As such, overall system utility is defined as the sum of the benefits of completing jobs minus the cost of executing a tactic (tactic *nop* has no cost).

4.1 Modelling the Components of the Framework

Environment. We model the environment as generating two types of job (J1 and J2) according to probability *pJob1* (or *pJob2*=1-*pJob1*). Although it could be trivially extended to generate more job types, having only two is enough for the purpose of reasoning about whether to adapt the ML component.

Adaptation Manager. The adaptation manager is responsible for triggering adaptations. In the running example, two tactics are available to be executed: *nop* (no operation) and *retrain*. Whenever the system receives a new job generated by the environment, the pre-condition for the tactics' execution becomes true. At

Table 2. Discretized distribution of job latencies for both job types and for each platform and corresponding likelihoods. Each cell in each matrix has the probability of the corresponding PRISM transition. That is, in Table 2a with probability 18% the predicted latency for a job is 3 on platform 1 (P1) and 8 on platform 2 (P2). If, for this situation, the ML component is very accurate, it will select platform 1 to deploy the job, since P1 has the lowest latency. The difference between job types lies in the accuracy of the ML model for each.

(a) Job latency depends on the platform in which it is executed. Thus, in this case, ML accuracy has an impact on system utility.

P1 \ **P2**	lat.	6	8	10
lat.	prob.	20%	30%	50%
3	60%	12%	18%	30%
5	30%	6%	9%	15%
7	10%	2%	3%	5%

(b) The diagonal accounts for more than half of the probability, thus it is more likely that the latency of a job will be the same regardless of the platform, which means that ML accuracy should have no impact on system utility.

P1 \ **P2**	lat.	3	5	7
lat.	prob.	15%	70%	15%
3	15%	2.25%	10.5%	2.25%
5	70%	10.50%	49.0%	10.50%
7	15%	2.25%	10.5%	2.25%

this point, the model checker, when asked to synthesize optimal policies, decides whether to adapt or do nothing and triggers the corresponding tactic in the ML component. The latency of the tactic is accounted for by the ML component during tactic execution. The tactic's cost is subtracted from the system's rewards when the job completes its execution.

Non-Machine-Learning Component. The non-ML component, which is responsible for simulating the execution of the jobs has two possible platforms at its disposal. When the environment generates a new job, and after the adaptation manager has selected the adaptation tactic to execute, the executor is ready to deploy the job on the selected platform. To simulate the execution, it needs to know the latency of the job. As there is intrinsic uncertainty in determining job execution latency, we assume the existence of historical data which can be used to construct distributions of possible execution latencies. These distributions can be discretized (as shown in Table 2) and fed to PRISM so that model checking is feasible and this uncertainty is explicitly modeled. Whenever a job completes, the utility of the system is updated by adding the job completion reward and subtracting the tactic execution cost.

Machine Learning Component. In this use-case, since there are two possible execution platforms and two input job types, we define two binary confusion matrices: one for each type of job. As for the knowledge element of the state of the ML component, in order to model this aspect, we count the inputs of each type that are received and use this count as a proxy for the amount of information encoded in these new inputs. This count is increased whenever an input arrives and reset whenever the tactic is executed. This information then contributes to the impact of the retrain adaptation tactic on the ML component. The increase in model's accuracy is computed as described in Sect. 3.2.

4.2 Collecting Rewards

Since the system receives a fixed reward whenever it completes a job, its goal is to maximize the number of jobs completed in a given time period, while simultaneously minimizing the costs spent on retraining. This requires seeking an adequate trade-off between investing time and resources to retrain the model and reasoning about the expected impact of the current accuracy on system utility. Since model checking tools require the specification of properties in order to compute optimal policies, we verify with PRISM a property that maximizes system utility when the state "end" is reached, i.e., when the time period expires.

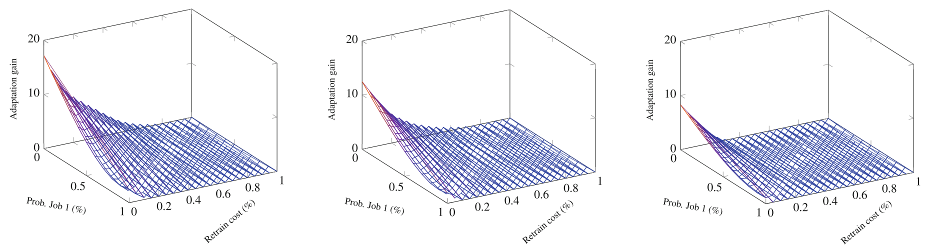
5 Results

In this section, we evaluate whether the proposed framework can reason about the trade-offs of ML component adaptation. Specifically, to understand whether adding self-adaptation functionalities to ML-based systems translates into increased benefits to the system, we investigate two research questions:

RQ1 – What are the estimated utility gains achievable through ML adaptation?
RQ2 – Under what conditions does the framework determine that ML adaptation improves overall system utility?

Experimental settings. In our experiments we varied the following parameters: **(i)** the retrain cost; **(ii)** the retrain latency (1, 5, 10); and **(iii)** the probability that the environment generates each type of job (from 0 to 1 with 0.1 increments). Throughout all experiments we set to 100% the probability of the environment generating a new job. Since we are interested in modelling environment changes, we assume that the ML model has better knowledge for one type of job than for the other, which is assumed to be the environment change. Thus, the ML model has an accuracy of 95% for jobs of type 1 and an accuracy of 50% for jobs of type 2 (these correspond to symmetric confusion matrices), and the impact factor is set to 0.1 for both types of jobs. Job latencies and uncertainty in each platform are set according to Table 2.

Results. Figure 2 shows, for an execution context in which ML accuracy affects system utility, the utility gains achievable due to ML adaptation. The difference between plots corresponds to the latency of execution of the retrain tactic. We can see that, regardless of how this parameter is set, adapting the ML component improves system utility in specific areas of the space. Determining the boundary that divides the areas of the space in which it is worth/not worth adapting is thus a critical aspect. Our framework is capable of determining this boundary, which we show in the following paragraphs.



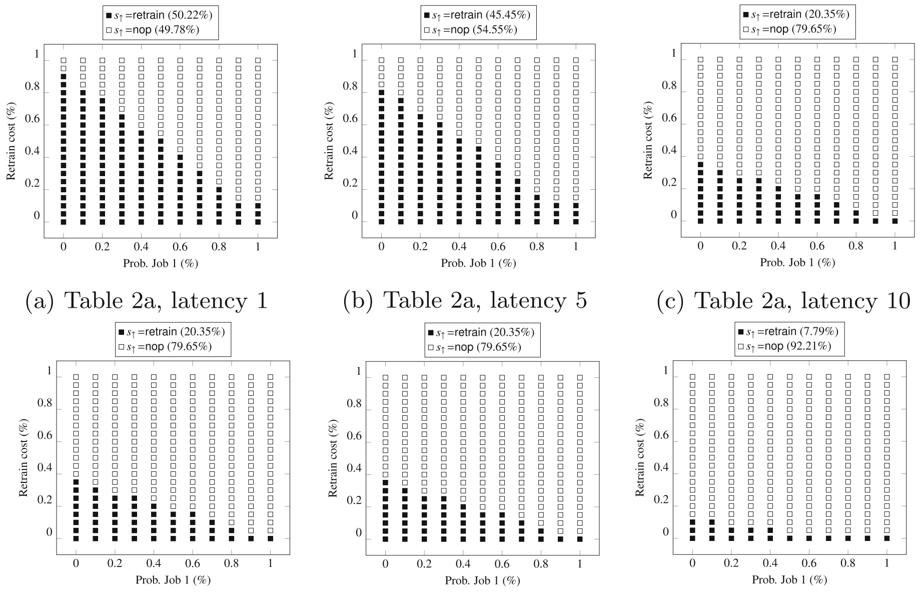
(a) Latency 1 (b) Latency 5 (c) Latency 10

Fig. 2. Utility gains achievable due to ML adaptation when the system operates in an execution context in which ML accuracy impacts system utility.

The plots in Fig. 3 represent, for different execution contexts and tactic latency, the conditions of the environment in which adapting the ML model improves overall system utility. We now focus on Fig. 3a and analyze the impact of the retrain cost and job probability variables. As expected, for low values of retrain cost, the framework tells us that adaptation is always worth it, regardless of the environment stimuli. However, as the cost starts to increase, adaptation is no longer the optimal action when the environment is more likely to generate jobs of type 1. This is due to the fact that, since the ML model has a good knowledge for jobs of type 1, the costs of adaptation outweigh its benefits. Differently, when the environment generates more jobs of type 2 (prob. job 1 $<$ 50%), the tolerated cost of adaptation tactics increases. As tactic latency increases (Figs. 3b and 3c), we see that in order for adaptation to be worth it, its cost must also be lower than in scenarios with lower latency.

The difference between the figures in the top row (Figs. 3a, 3b,3c) and the figures in the bottom row (Figs. 3d, 3e,3f) is the execution context of the system, that is, the latencies of the jobs in each platform and their probabilities (which are set according to Table 2). For the bottom row it is more likely that a job has the same latency regardless of the platform (Table 2b). In such a situation, having an inaccurate ML model has little impact on system utility. The comparison between top and bottom rows demonstrates this effect: for the bottom row plots, adaptation pays off only in very few scenarios and only when tactic cost is low. In fact, we see that when latency is high (Fig. 3f), the cost of retrain has to be close to zero for adaptation to provide benefits.

Overall, our preliminary results confirm that adaptation of ML components in systems improves overall system utility (RQ1). Our framework also shows that, in this concrete example, if the costs of adaptation are too high, adaptation pays off if: **(i)** the environment starts generating more jobs that are less known (probability of job 1 lower than 0.5), or **(ii)** the tactic has a low latency. The framework further shows that in execution contexts in which ML accuracy is not expected to impact system utility, ML adaptation rarely pays off (RQ2).



(a) Table 2a, latency 1 (b) Table 2a, latency 5 (c) Table 2a, latency 10

(d) Table 2b, latency 1 (e) Table 2b, latency 5 (f) Table 2b, latency 10

Fig. 3. Areas of the space in which ML adaptation improves overall system utility. Black squares correspond to configurations in which it is worth adapting. White squares correspond to configurations in which having the option to adapt does not improve utility. The top row corresponds to the execution context of Table 2a and the bottom row to the execution context of Table 2b.

6 Related Work

Existing frameworks for reasoning about self-adaptation are mostly focused on dealing with uncertainty [4, 25, 26] and on analyzing human involvement and the benefits of providing explanations [9, 21–23]. Differently, the framework we propose is concerned with analyzing the trade-offs of adapting an ML component of a system when it has a negative impact on system utility.

In the literature on self-adaptive systems ML has been leveraged by recent works to improve different stages of the MAPE-K loop, such as the Plan and the Analysis stages [13]. Works focusing on collective SAS have also researched the most common learning strategies employed in these systems, finding that reinforcement learning is the most common [10, 12]. Differently from both these lines of work, our framework is focused on single-agent systems that rely on ML in order to work as expected. Our framework, which leverages self-adaptation to improve AI in systems, is aligned with the vision of Bureš [3], that argues for a new self-adaptation paradigm in which self-adaptation and AI benefit and enable one-another. This is also aligned with the vision for continual AutoML [11], which advocates for architectures that can manage ML systems and adapt them in the face of adversities (such as uncertainty, data-set shift, outliers).

The literature on continual/lifelong learning addresses problems which self-adaptive systems also face [24,30]. Specifically, this branch of literature focuses on open-world problems (i.e., unexpected changes can occur) and on how to learn new tasks from past tasks. In fact, as discussed in our prior work, the techniques developed in this field can be thought of as tactics of self-adaptive systems to adapt ML models [7]. Instead, in this work, we present an actual model-checking based framework to reason about whether the benefits of executing an adaptation tactic outweigh its costs and improve overall system utility.

7 Threats to Validity and Future Work

Our framework relies on the assumption that we have access to historical data, however this might not be the case for all systems. Specifically, for new deployments of systems (e.g., new client of a fraud detection company) there is an initial period during which data must be collected. Nonetheless, we believe this assumption is acceptable since this data collection period is also required to have an initial training set with which to train the ML component.

Further, in this work we adopt a pragmatic strategy for modelling the benefits of a retrain tactic, which corresponds to instantiating a parametric model that defines the benefits of retrain as being proportional to the model's loss and to the amount of new data gathered since the previous execution of the tactic. However, this corresponds to a simple model of the benefits of adaptation. In fact, not only is it possible that this relationship is not linear, but also there are likely other factors that influence the benefits of a tactic and that should be considered. Examples of such factors are the latency of a tactic and the context of the system upon the tactic's execution. Taking the example of the retrain tactic, intuitively one would expect the benefits provided by such a tactic to increase as its execution latency also increases, i.e., when the system is retrained for a longer period. However, it is also expected that the achievable accuracy will plateau at some point, thus not corresponding to a linear relationship. As future work, we plan to study how the benefits of different adaptation tactics vary based on parameters such as tactic execution latency, application and application context, to extend our current model of adaptation benefits to account for these factors.

To address possible scalability issues and generalize the proposed framework to more job types, one can change the PRISM model to reason about aggregated metrics instead of job specific metrics. This can be achieved for example by evaluating the system in terms of correctly scheduled jobs per time interval (user-defined). This would require the system to monitor the expected number of jobs generated per time interval and the confusion matrix to model the probabilities of a job being correctly or incorrectly scheduled (independently of their type).

Finally, since in the presented use case we considered a single adaptation tactic (model re-train), we plan to conduct the aforementioned study on several adaptation tactics. This corresponds to extending the repertoire of tactics considered by the framework and also entails the extension of the framework to account for dynamic environments.

8 Conclusion

In this paper we proposed a framework to reason about the need to adapt ML components of ML-based systems. Resorting to a running example, we showed how to instantiate the framework in a practical setting and how system utility can be improved through the adaptation of ML components. We further demonstrated how the adaptation decision boundary is affected by environment changes and execution context. As next steps, we plan to investigate more fine-grained approaches to modelling the effects of retraining ML models, as well as to extend our framework to consider additional adaptation tactics (e.g., unlearning and human-in-the-loop).

References

1. Alipourfard, O., et al.: Cherrypick: adaptively unearthing the best cloud configurations for big data analytics. In: Proceedings of NSDI (2017)
2. Aparício, D., et al.: Arms: Automated rules management system for fraud detection. arXiv preprint. arXiv:2002.06075 (2020)
3. Bureš, T.: Self-adaptation 2.0. In: Proceedings of SEAMS (2021)
4. Cámara, J., et al.: Reasoning about sensing uncertainty and its reduction in decision-making for self-adaptation. Sci. Comput. Program. **167**, 51–69 (2018)
5. Cao, Y., Yang, J.: Towards making systems forget with machine unlearning. In: Proceedings of IEEE S& P (2015)
6. Casimiro, M., et al.: Lynceus: cost-efficient tuning and provisioning of data analytic jobs. In: Proceedings of ICDCS (2020)
7. Casimiro, M., et al.: Self-adaptation for machine learning based systems. In: Proceedings of SAML. LNCS, Springer (2021)
8. Cito, J., Dillig, I., Kim, S., Murali, V., Chandra, S.: Explaining mispredictions of machine learning models using rule induction. In: Proceedings of ESEC/FSE (2021)
9. Cámara, J., Moreno, G., Garlan, D.: Reasoning about human participation in self-adaptive systems. In: Proceedings of SEAMS (2015)
10. D'Angelo, M., et al.: On learning in collective self-adaptive systems: state of practice and a 3d framework. In: Proceedings of SEAMS (2019)
11. Diethe, T., et al.: Continual learning in practice. Presented at the NeurIPS 2018 Workshop on Continual Learning (2019)
12. D'Angelo, M., et al.: Learning to learn in collective adaptive systems: mining design patterns for data-driven reasoning. In: Proceedings of ACSOS-C (2020)
13. Gheibi, O., Weyns, D., Quin, F.: Applying machine learning in self-adaptive systems: A systematic literature review. arXiv preprint. arXiv:2103.04112 (2021)
14. Gu, T., et al.: Badnets: evaluating backdooring attacks on deep neural networks. IEEE Access **7**, 47230–47244 (2019)
15. Huang, L., et al.: Adversarial machine learning. In: Proceedings of AISec (2011)
16. Jamshidi, P., et al.: Transfer learning for improving model predictions in highly configurable software. In: Proceedings of SEAMS (2017)
17. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003)
18. Koh, P.W., et al.: Wilds: A benchmark of in-the-wild distribution shifts. In: Proceedings of ICML. PMLR (2021)

19. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47
20. Li, J., Hu, M.: Continuous model adaptation using online meta-learning for smart grid application. IEEE Trans. Neural Netw. Learn. Syst. **32**(8), 3633–3642 (2021)
21. Li, N., Adepu, S., Kang, E., Garlan, D.: Explanations for human-on-the-loop: a probabilistic model checking approach. In: Proceedings of SEAMS (2020)
22. Li, N., Cámara, J., Garlan, D., Schmerl, B.: Reasoning about when to provide explanation for human-in-the-loop self-adaptive systems. In: Proceedings of ACSOS (2020)
23. Li, N., Cámara, J., Garlan, D., Schmerl, B., Jin, Z.: Hey! preparing humans to do tasks in self-adaptive systems. In: Proceedings of SEAMS (2021)
24. Liu, B.: Learning on the job: online lifelong and continual learning. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 34 (2020)
25. Moreno, G.A., et al.: Proactive self-adaptation under uncertainty: a probabilistic model checking approach. In: Proceedings of ESEC/FSE (2015)
26. Moreno, G.A., et al.: Uncertainty reduction in self-adaptive systems. In: Proceedings of SEAMS (2018)
27. Pan, S.J., Yang, Q.: A survey on transfer learning. IEEE TKDE **22**(10), 1345–1359 (2009)
28. Pinto, F., et al.: Automatic model monitoring for data streams. arXiv preprint. arXiv:1908.04240 (2019)
29. Quionero-Candela, J., et al.: Dataset Shift in Machine Learning. The MIT Press, Cambridge (2009)
30. Silver, D.L., Yang, Q., Li, L.: Lifelong machine learning systems: beyond learning algorithms. In: 2013 AAAI spring symposium series (2013)
31. Varshney, K.R., Alemzadeh, H.: On the safety of machine learning: cyber-physical systems, decision sciences, and data products. Big Data **5**(3), 246–255 (2017)
32. Wu, Y., Dobriban, E., Davidson, S.: DeltaGrad: rapid retraining of machine learning models. In: Proceedings of ICML (2020)