Towards a Framework for Adapting Machine Learning Components

Maria Casimiro
INESC-ID, IST, Universidade de Lisboa
ISR, Carnegie Mellon University
maria.casimiro@tecnico.ulisboa.pt

Paolo Romano
INESC-ID, IST
Universidade de Lisboa
romano@inesc-id.pt

David Garlan
Institute for Software Research
Carnegie Mellon University
garlan@cs.cmu.edu

Luís Rodrigues
INESC-ID, IST
Universidade de Lisboa
ler@tecnico.ulisboa.pt

***Abstract*—Machine Learning (ML) models are now commonly used as components in systems. As any other component, ML components can produce erroneous outputs that may penalize system utility. In this context, self-adaptive systems emerge as a natural approach to cope with ML mispredictions, through the execution of adaptation tactics such as *model retraining*. To synthesize an adaptation strategy, the self-adaptation manager needs to reason about the cost-benefit tradeoffs of the applicable tactics, which is a non-trivial task for tactics such as *model retraining*, whose benefits are both context- and data-dependent.**

To address this challenge, this paper proposes a probabilistic modeling framework that supports automated reasoning about the cost/benefit tradeoffs associated with improving ML components of ML-based systems. The key idea of the proposed approach is to decouple the problems of (i) estimating the expected performance improvement after retrain and (ii) estimating the impact of ML improved predictions on overall system utility.

We demonstrate the application of the proposed framework by using it to self-adapt a state-of-the-art ML-based fraud-detection system, which we evaluate using a publicly-available, real fraud detection dataset. We show that by predicting system utility stemming from retraining a ML component, the probabilistic model checker can generate adaptation strategies that are significantly closer to the optimal, as compared against baselines such as periodic retraining, or reactive retraining.

***Index Terms*—self-adaptation, machine learning, model retrain, fraud detection system**

I. Introduction

The widespread use of Machine Learning (ML) models for a variety of tasks spanning multiple domains (e.g., enterprise and cyber-physical systems) raises concerns regarding the impact of the quality of the ML components on system performance. Indeed, the quality of a ML model in production is inherently affected by the training data used for its synthesis, and in particular by whether the statistical relations present in the training data also hold when the model is used in production. Further, different operational contexts may have different ML quality requirements. So if ML quality is acceptable but the context changes, higher quality decisions may be required, thus also triggering ML adaptation.

When deploying a ML model in the real world, typically under changing environments, the actual sample distribution may differ from the one under which the model was trained. These samples are known as out-of-distribution (OOD) samples [1] and can be caused for instance by concept drift (i.e., shifts of the input features) and co-variate shift (i.e., shift in the relationship between input feature and the target variable) [2]. OOD samples are thus a common cause of ML mispredictions [3], [4] and while these problems and how to detect their occurrence have been extensively studied by the ML literature [5]–[8], little research has addressed the problems of: **(i)** quantifying the expected impact of ML mispredictions on system utility – e.g., including penalties due to service level agreement (SLA) violations and costs related to training a ML model in the cloud; **(ii)** reasoning about what corrective actions to enact in order to maximize system utility in the face of ML mispredictions.

Self-adaptive systems [9], [10], which are systems capable of reacting to environment changes in order to maintain system utility at desired levels, emerge as a natural solution to cope with ML mispredictions. In particular, the use of formal reasoning mechanisms for synthesizing optimal adaptation strategies (i.e., sequences of adaptation tactics [11]) could ideally be applied to ML-based systems.

While previous work in the self-adaptive systems literature [12]–[14] has leveraged probabilistic model checking techniques to synthesize optimal adaptation strategies for non-ML-based systems, extending those frameworks to deal with ML-based systems is far from trivial. First, since probabilistic model checkers verify properties of a formal model of a system, formal models of ML-based systems need to capture the key dynamics of ML components in a compact but meaningful way. This calls for identifying the right abstraction level to represent such components, ensuring not only that their characteristic behaviors are modeled, but also that the formal abstraction is expressive, general, accurate, and that the model verification is tractable for usage in online adaptation of systems. Leveraging such an abstraction to represent ML components ideally would allow the model checker to reason about the impacts of mispredictions on system utility.

Second, a key requirement for self-adaptive systems is the quantification of the benefits and costs of applying different adaptation tactics. Understanding these tradeoffs allows a planner to select one tactic over another, or more generally one adaptation strategy over another. However, due to the

Support for this research was provided by ANI and 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.

context- and data-dependencies of ML adaptation tactics such as *model retrain*, estimating the costs and benefits of such tactics requires developing specified predictors. While a number of solutions have been recently proposed to estimate the cost/latency of (re)training ML models on different types of computational resources [15], [16], the problem of predicting the benefits on model accuracy deriving from retraining the model has not been addressed by the current literature.

This paper proposes a probabilistic framework based on model checking to reason, in a principled way, about the cost/benefit tradeoffs associated with adapting ML components of ML-based systems. The proposed approach is based on the insight that this is achievable by decoupling the problems of **(i)** modeling the impact of an adaptation tactic on the ML model's performance and **(ii)** estimating the impact of ML (mis)predictions on system utility. We show that the former can be effectively tackled by relying on blackbox predictors that leverage historical data of previous retraining processes. The latter problem is solved by expressing inter-component dependencies via an architectural model, which enables automated reasoning via model checking.

To validate the proposed framework, we apply it to a fraud detection use-case and implement a prototype of a self-adaptive credit-card fraud detection system. Specifically, we use the proposed framework to automate the decision of when to retrain a state of the art ML model for fraud detection [17] and evaluate it using a public data set [18], accounting for the impact of SLA violations as well as model retrain cost and latency on system utility. We demonstrate that by leveraging the predicted benefits of retraining a ML component, a self-adaptation manager can generate adaptation strategies that are closer to the optimal one when compared against baselines such as periodic retrains, or reactive retrains (triggered upon an SLA violation).

II. Probabilistic Model Checking

Probabilistic model checking is a set of methods for reasoning about and analyzing systems that exhibit probabilistic and uncertain behavior. Probabilistic model checking techniques have been extensively used in the self-adaptive systems literature [12]–[14] to synthesize optimal adaptation strategies. To generate these strategies, it is necessary to instantiate a formal model of the system under adaptation, and to specify an adaptation goal in the form of a property (written as a temporal logic formula) which the model checker can verify for optimality. Additionally, these techniques are a natural fit for planning the need for adaptation in self-adaptive systems since they support proactive adaptation schemes such as *look-ahead* [14]. This consists of having the model checker, via the formal model of the system, simulate the different possible future states to synthesize the adaptation strategy (sequence of adaptation tactics to execute) that maximizes system utility.

This work leverages the PRISM model checker [19], which is a probabilistic model checker commonly used in the literature [14], [20]. We define the formal models as Markov Decision Processes (MDPs), which allow to model systems' dynamics through a set of states, whose transitions are either probabilistic or partially controlled by an actor and which model the evolution of the state of the system in discrete timesteps. At each timestep, a set of adaptation tactics is available for adaptation and the model checker has to select one to execute. Not adapting is considered a possible tactic available for adaptation that has no impact on the system and which we refer to as *NOP*. The choice between adaptation tactics corresponds to a nondeterministic choice, and is guided by the optimization of a user-defined property. This property encodes the goal of the adaptation process and generally corresponds to maximizing system utility. To specify these properties, PRISM relies on probabilistic reward computation-tree logic (PRCTL) [21]. By specifying reward-based properties as a function of state-specific constraints of the system, PRISM generates optimal strategies that comply with these constraints.

III. Framework for ML Adaptation

This section introduces the proposed framework for reasoning about adaptation of ML-based systems. We start by discussing the assumptions and design goals underlying the framework and its requirements for ensuring the design goals. Then, we focus on its novel aspects, namely: **(i)** how to formally model ML components in order to reason about the impacts of ML mispredictions on system utility; **(ii)** how to predict the costs/benefits of different adaptation tactics and how to integrate these predictions with the formal model.

A. Design Goals and Assumptions

The proposed framework targets systems composed of ML and non-ML components and is designed to automate the analysis of the tradeoffs associated with adapting (e.g., retraining) a ML component at a given moment with the goal of maximizing system utility. Our design aims to ensure the following key properties: **(i) generic** – designed to be applicable to different types of offline supervised ML models (e.g., neural networks, random forests); **(ii) tractable** – designed to be usable by a probabilistic model checker like PRISM, which requires identifying an adequate level of abstraction to model ML components in order to enable systematic analysis via model checking; **(iii) expressive** – designed to capture the general and key dynamics of ML models; **(iv) extensible** – designed to be easily extended to incorporate additional adaptation tactics (e.g., transfer learning, unlearning), as discussed by [22], and customized to capture application specific dynamics.

To realize the proposed framework it is assumed that:

- **A1** There are fluctuations of the ML model's quality over time – ML techniques are inherently approximate (they can mispredict even in absence of changes); or the system may be operating under changing environments which lead to ML mispredictions (data shift);
- **A2** The adaptation tactics can have non-negligible costs and latencies. Considering the case of an adaptation tactic such as model *retrain*, the costs could be quantified in terms of energy consumption or as the economical cost

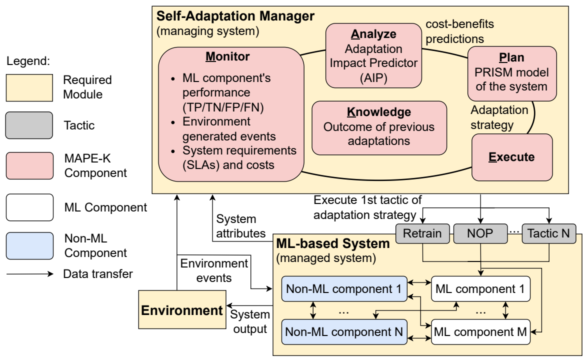

Legend:
Required Module
Tactic
MAPE-K Component
ML Component
Non-ML Component
Data transfer
Self-Adaptation Manager (managing system)
Monitor
ML component's performance (TP/TN/FP/FN)
Environment generated events
System requirements (SLAs) and costs
Analyze Adaptation Impact Predictor (AIP)
cost-benefits predictions
Plan PRISM model of the system
Knowledge Outcome of previous adaptations
Adaptation strategy
Execute
Execute 1st tactic of adaptation strategy
System attributes
Retrain
NOP
Tactic N
ML-based System (managed system)
Environment events
Non-ML component 1
ML component 1
Non-ML component N
ML component M
Environment
System output

Fig. 1: Architecture of the self-adaptation framework.

incurred by provisioning the virtual machines used for retrain (in the case of cloud deployments);

A3 Time is discretized into fixed-sized intervals. At each time interval, the framework synthesizes the optimal adaptation strategy to use in the following time interval(s). The most appropriate granularity of time discretization is inherently application dependent and should be chosen by taking into consideration that it will affect the rate at which adaptations are performed.

B. Self-Adaptation Manager's Architecture

Similarly to previous work [12]–[14], the proposed framework leverages a self-adaptation manager that adopts a MAPE-K [23] architecture, as illustrated in Figure 1. The following paragraphs briefly describe each module of the architecture.

Environment. Generates events which constitute the inputs to the system, and hence to its ML component(s). These events may cause ML mispredictions and a decrease in system utility.

ML-based system. Implements the domain specific tasks, for which it relies on at least one ML component and may rely on several other components, both ML and non-ML.

Self-adaptation manager. Provides the required functionalities for ML adaptation. The novelty of the proposed framework with respect to existing self-adaptation managers lies in the operation of the Analyze and Plan components: **Analyze** – contains the cost/benefit predictors, which leverage historical data of previous adaptations and of their impact on the ML component's quality (e.g., accuracy) in order to estimate its future performance in case an adaptation is or is not executed; **Plan** – comprises the adaptation planner, which relies on a formal model of the system being adapted and on a probabilistic model checker to synthesize the adaptation strategy that maximizes system utility.

The self-adaptation manager should thus abide by the following key requirements:

R1 Provide the means to predict the effects of adapting and not adapting the model on its future accuracy;

R2 Include a way to characterize in a compact but meaningful way the error of a ML component;

R3 Be able to determine the impact of ML mispredictions on overall system utility.

The following sections describe the Analyze and Plan components, explaining how these requirements are met.

C. Formally Modeling ML Components

This section details how we formally model the ML components to capture their error in a compact but meaningful way (realizing R2), and its impact on system utility (realizing R3).

ML component definition. Depending on the domain of operation of a system, the most appropriate type of ML component varies. For instance, while in cyber-physical systems it is common to see reinforcement learning ML components [24], in the context of fraud detection [25], [26] or medical diagnosis systems [27] offline trained ML components (e.g., decision trees or neural networks) are more common. We focus our analysis on offline trained ML components and specifically on ML classifiers. (Note that it is possible to transform a regressor into a classifier by discretizing the target domain, although this implies introducing an intrinsic prediction error due to the chosen discretization granularity.)

ML component state. Since the goal of the proposed framework is to model ML components, and in particular the impact of their mispredictions on system utility, we require a way to evaluate their classification performance. Classification models are typically evaluated based on a popular construct known as confusion matrix [28], which provides a statistical characterization of the model's quality by describing the distribution of its misclassification errors. For a classification problem with N classes, the confusion matrix normalized by rows $\mathcal{C}$ (the rows represent the actual sample class and sum to 1) contains, in each cell (i, j), the ratio of samples of class i (ground truth) that have been classified as being of class j (prediction). For the simpler case of binary classification problems, the confusion matrix is reduced to a 2×2 matrix where each cell specifies the following: True Positives Rate (TPR) – percentage of examples of the positive class that the model classified as such; True Negatives Rate (TNR) – percentage of examples of the negative class that the model classified as such; False Positives Rate (FPR) – percentage of examples of the negative class that the model classified as positive; False Negatives Rate (FNR) – percentage of examples of the positive class that the model classified as negative. This representation allows for extracting further error metrics such as the model's accuracy, and f1-score.

The row-normalized confusion matrix has the following relevant properties: **(i) generic** – can be computed for different ML models (e.g., random-forest or neural network); **(ii) tractable** – is compact and abstract enough to be encoded into a formal model; **(iii) expressive** – captures the predictive performance of the ML model and allows for computing several error metrics; **(iv) extensible** – can be used to model the impacts of executing different adaptation tactics [22] (e.g., retrain, nop), by updating its cells. These properties make it a natural fit to model ML components and hence realize R2.

Depending on the predictive models used by the model checker to estimate the evolution of the confusion matrix (or the cost of executing an adaptation tactic) the state of the ML component can be extended with additional variables, e.g., that describe the expected data shifts on the input or output.

ML Component Interface. Since the framework aims to adapt offline-trained ML components, we define the base interface as being composed of the methods *query* and *retrain*. As the name suggests, this method models the execution of a retrain procedure of the ML component by triggering an update of its row-normalized confusion matrix. The techniques employed to predict how the confusion matrix of a ML component evolves as a result of a retrain procedure are described Section III-D. The *query* method models the process of asking the ML component for predictions for a set of inputs. Specifically, this method should abstract over the concrete input/output values of the samples and of the predictions, requiring only the total number of inputs for the ML component and the expected distribution of (real) output classes **O** (given by the probability p_i for an input to be of class i, for all classes $i \in [1, N]$). The method returns a (non-normalized) confusion matrix $\mathcal{C}^*$, that reports in position (i, j) the (absolute) number of inputs of class i that are classified as of class j by the model. $\mathcal{C}^*$ can be simply computed by multiplying each row i of the normalized confusion matrix $\mathcal{C}$ by p_i. The interface can be extended to account for more adaptation tactics which allow to tailor the framework to specific adaptation scenarios.

Dealing with uncertainty. As shown by recent work [14], [29], [30], capturing uncertainty and including it when reasoning about adaptation contributes to improved decision making. To capture uncertainty, we leverage the probabilistic framework proposed by Moreno et al. [11] which allows to account for different sources of uncertainty in the system (e.g. uncertainty on the effects of an adaptation tactic or on the input class distribution) and which generates memoryless strategies (strategies that depend only on the current state of the system). This framework accounts for uncertainty by modeling the source of uncertainty as a probabilistic tree which is approximated via the Extended Pearson-Tukey (EP-T) [31] three-point approximation. The current state of the source of uncertainty is represented by the root node of the probability tree and the child nodes are its possible realizations.

D. Predicting the Effects of Adapting (or not)

A key requirement of our framework is the ability to predict the costs and benefits of executing adaptation tactics on the ML components (requirement R1). For this purpose, the proposed framework associates with each adaptation tactic a dedicated component, which we call Adaptation Impact Predictor (AIP). The AIP is in charge of predicting: **(i)** the adaptation tactic's *cost*, that is charged to the system utility; **(ii)** the impact of the adaptation on the future *quality* of the ML component. For each ML component, we also include an adaptation tactic corresponding to performing no changes to the ML component (*NOP*). While the AIP for tactic *NOP* always predicts zero costs (this tactic inherently has no cost), its model quality predictor captures the evolution of the model's performance if no action is taken, e.g., the possible degradation of accuracy of the ML component due to data shifts. Overall, this approach allows the model checker to quantify the impact of different adaptation tactics on system utility and reason about their cost/benefits tradeoffs.

We focus on the problem of how to predict the future evolution of the performance of the ML component and describe, in the next section, how we tackle the problem of implementing AIPs for the retrain and NOP tactic for generic ML components. Indeed, for adaptation tactics such as *retrain* the problem of estimating their costs has been investigated in the system's community. The literature has shown that data-driven approaches [15] based on observing previous retraining procedures, possibly mixed with white-box methods [16], can generate accurate predictive models of the retrain cost.

Predicting future quality of ML components. Given the reliance on a row-normalized confusion matrix $\mathcal{C}$ to characterize the performance of ML components, predicting their performance evolution requires estimating how $\mathcal{C}$ will evolve in the future, e.g., due to shifts affecting the quality of the current model or as a consequence of retraining the model to incorporate newly available data.

The proposed method abstracts over the specific adaptation $a()$ by modeling it as a generic function $a(\mathcal{M}, \mathcal{I}, \mathcal{N}) \longrightarrow \mathcal{M}'$ that produces a new ML model $\mathcal{M}'$, and takes as input: **(i)** model $\mathcal{M}$ prior to the execution of the adaptation; **(ii)** data $\mathcal{I}$, used to generate model $\mathcal{M}$; **(iii)** new data, $\mathcal{N}$, that became available since the last adaptation, e.g., by deploying the model in production and gathering new samples and corresponding ground truth labels. We assume that both $\mathcal{I}$ and $\mathcal{N}$ contain ground truth labels. Additionally, we assume that $\mathcal{M}$ and $\mathcal{M}'$ are generic supervised ML models that are queried and returned predictions for the input samples. These two assumptions allow to determine the confusion matrices of models $\mathcal{M}$ and $\mathcal{M}'$ at any future time interval, since their predictions can be compared with the ground truth labels.

We seek to build blackbox regressors (e.g., random forests or neural networks) that, given model $\mathcal{M}$ obtained at time 0 on dataset $\mathcal{I}$, and given new data $\mathcal{N}$ available at time $t > 0$, predict the confusion matrices of both models ($\mathcal{M}$ and $\mathcal{M}'$) at time $t + k$, where $k > 0$ is the prediction lookahead window.

Adaptation impact dataset. In order to train such a blackbox regressor, we build an Adaptation Impact Dataset (AID) by systematically simulating the execution of the adaptation tactic using production data in different points in time. This allows for gathering observations characterizing the execution of the adaption tactic in different environmental contexts, such as: **(i)** different sets of data used to adapt the model; **(ii)** variation in the time passed since the last execution of the tactic; **(iii)** different ML performance before and after adaptation

The first step of the procedure consists in monitoring model $\mathcal{M}_0$ of a ML component in production over T time intervals. During this monitoring period, given the absence of AIPs, we assume that no adaptation is executed.

Next, we deploy $\mathcal{M}_0$ on a testing platform (so as not to affect the production environment) and systematically apply adaptation $a()$ at each time interval $i > 0$, i.e., $a(\mathcal{M}_0, \mathcal{I}_0, \mathcal{N}_i)$. This yields a new model $\mathcal{M}_i$, which we evaluate at every future time interval $i < j \leq T$, obtaining the corresponding confusion matrices, noted as $\mathcal{C}_i(j)$. Overall, this procedures yields T models, resulting from the adaptation of $\mathcal{M}_0$ at different time intervals, and produces $T \cdot (T-1)$ measurements of the confusion matrices at times $j > i$.

For each of the aforementioned $T \cdot (T-1)$ measurements, we generate an AID entry, $e_{i,j,k}$, which describes the quality at time $j+k$ of model $\mathcal{M}_j$ obtained by executing $a(\mathcal{M}_i, \mathcal{I}_i, \mathcal{N}_j)$ at time j on model $\mathcal{M}_i$, where $\mathcal{I}_i$ denotes the data used at time i to generate model $\mathcal{M}_i$, and $\mathcal{N}_j$ the new data gathered from time i until time j. Each entry $e_{i,j,k}$ has as target variables the $N^2 - N$ independent entries of the confusion matrix at time $j+k$ of model $\mathcal{M}_j$ and stores the following features:

- *Basic Features:* provide basic information on **(i)** the amount of data (i.e., number of examples) used to generate model $\mathcal{M}_i$, i.e., $\mathcal{I}_i$, and gathered thereafter, i.e., $\mathcal{N}_j$; **(ii)** the accuracy of the model shortly after its generation and at the present time; **(iii)** the time elapsed since the last execution of the adaptation tactic, i.e., $j - i$.
- *Output Characteristics Features:* describe the distribution of the output of models $\mathcal{M}_i$ and $\mathcal{M}_j$. It also includes the distribution of the uncertainty of the models' predictions. This feature is included only when the ML model provides information regarding the uncertainty of a prediction. This information is usually provided by commonly employed ML models like random forests, Gaussian processes, and ensembles.
- *Input Characteristics Features:* aim to capture variations in the distributions of the features of datasets $\mathcal{I}_i$ and $\mathcal{N}_j$. Specifically, for each feature f, we compute the Pearson correlation coefficient between its values in $\mathcal{I}_i$ and $\mathcal{N}_j$.

Overall, the AID can be seen as composed of pairs of features, where each pair describes a specific "characteristic" of the data or model at two different points in time, e.g., amount of data available at time i and j, or distribution of predicted classes at time $j+k$ by models $\mathcal{M}_i$ and $\mathcal{M}_j$. The last step of the process consists of extending the AID by encoding the variation of each feature as follows: **(i)** for scalar features (e.g., amount of data) we encode their variation using the ratio and difference; **(ii)** for features described via probability distributions (e.g., prediction's uncertainty) we quantify their variation using the Jensen-Shannon divergence [32] (inspired by previous work [25]), which yields a scalar measurement of the similarity between two probability distributions. This generic methodology can also be applied to the case of the NOP tactic. In this case, the dataset describes how the accuracy of a model originally obtained at time i will evolve at time $j+k$, based on the information available at time j.

Building the AIPs. We exploit the AID dataset to train a set of independent AIPs, which can be simple linear models or blackbox predictors such as random forests or neural networks. Each AIP is trained to predict the value of a different cell of the confusion matrix. Given an n-ary classification problem, we have $n^2 - n$ independent values for the corresponding confusion matrix, given that each row must sum to 1. For the case of binary classification, $n = 2$, it is sufficient to predict the values of the two elements on the diagonal, which, being in different rows, are not subject to any mutual constraint. For the general case of $n > 2$, it is necessary to ensure that the predictions of the AIPs targeting different cells of the same row sum to 1. This can be achieved by using a softmax function [33] to normalize the predictions generated by the AIPs into a probability distribution.

Integrating the AIPs in the formal model. As for the integration of the AIPs in the formal model, which is checked via a tool such as PRISM, a key practical issue is related with the fact that these tools do not typically allow for interacting with external processes (which could be used to encapsulate the implementation of the AIPs) during model analysis. This can be necessary if the model checker is used to reason on a lookahead horizon of $l > 1$ time intervals. This generates up to a^l possible adaptation strategies, where a is the number of adaptation tactics available, requiring up to $l \cdot a^l$ predictions.

This problem can be circumvented by integrating directly the AIPs as part of the formal model to be checked. This approach is reasonable if the AIPs are implemented via simple methods, such as linear models, but is cumbersome and unpractical for the case of more complex models, such as neural networks. An alternative approach, which is the one currently implemented in our framework, is to precompute all the predictions that will be required during the model checking phase and provide them as input constants to the model checker tool. This approach is viable only when the lookahead window and the set of available adaptations are small, but allow us to use arbitrary external predictors.

IV. Self-Adaptive Fraud Detection System

To demonstrate the proposed framework, we instantiate an online adaptation manager for a ML-based credit card fraud detection system. Typically, fraud detection systems rely on supervised binary classifiers to classify incoming (credit/debit card) transactions as either legitimate or fraudulent and have banks and merchants as their clients. In this domain, quality attributes of interest are for example the overall cost of service level agreement (SLA) violations. Hence, we consider that our system has SLAs on the target: **(i)** TPR (or recall) – percentage of fraudulent transactions actually caught – and **(ii)** FPR – percentage of fraudulent transactions not caught – which should be kept within pre-defined thresholds:

$$\text{SYSTEM}(\text{recall}) \geq \text{recall\_threshold};$$
$$\text{SYSTEM}(\text{FPR}) \leq \text{FPR\_threshold};$$

SLA violations can occur when the ML component misclassifies a substantial amount of samples, such that either the TPR (recall) decreases below the threshold, the FPR becomes higher than acceptable, or both. These misclassifications are typically caused by environmental changes through data shifts, i.e., the input to the ML component changes such that it is no longer capable of correctly classifying those samples. This occurs for example, when the amount of fraud in a given period increases, or when fraudsters change their strategies [22], [26].

Whenever these SLAs are violated, the system incurs non-negligible costs, which we assume are fixed. We further assume that the fraud-detection system is deployed in production and that new data is gathered continuously, along with the corresponding ground truth labels. The framework can either do nothing (*NOP*) or *retrain* the model, leveraging the newly collected data and labels. We consider fixed retrain costs since the problem of estimating these costs has already been addressed [15], [16]. We also capture retrain latency by having it weigh in the system utility. Specifically, retrain latency is first translated into a percentage of the time period. For this percentage of time, system utility is computed based on the confusion matrix of the ML component that represented the state prior to the retrain. For the remainder of the time interval, system utility is computed based on the confusion matrix of the retrained model. Since the distribution of environment generated events may not be uniform, system utility is further weighted by the percentage of events in each period (during adaptation and after). Finally, driven by the desire to keep the problem tractable (eschewing the need to estimate several future states of the ML component as described in Section III-D) and since we consider retrain latency to be less than one time interval, we fix the lookahead horizon to one time interval.

The framework solves the problem of deciding when to retrain such that the global cost given by the sum of SLA and retrain costs is minimized. Next, we describe the formal model of the system, illustrating some of its components resorting to PRISM syntax, and the process of creating the AIPs.

A. Formal Model of the Fraud Detection System

The formal model of the system requires modules for each of the different moving parts that have an impact on the system. Thus, we model: **(i)** the environment under which the system is operating; **(ii)** the actual system, to analyze how mispredictions affect system utility, to simulate the execution of the tactics and to understand their impact on system utility; and **(iii)** the adaptation tactics, which in our case consist of either retraining the model or sticking with the current one.

Since we assume that the two tactics cannot be executed simultaneously, we further consider an adaptation manager module that prevents this from happening and non deterministically selects which tactic to execute. As shown in Listing 1, whenever there is a new event generated by the environment (line 5) – for the fraud detection system an event consists of a batch of transactions – the adaptation manager enters the *selectTactic* stage and can select to execute one tactic among the available ones. For example, while tactic *nop* (do nothing) can always be executed (line 8), tactic *retrain* can only be executed when there is *newData* with which to train the ML component (lines 9-10). Finally, since our approach assumes that time is divided into fixed-sized intervals, we further model

Listing 1: Adaptation manager and System rewards[1]

```
1  module adaptation_manager
2      selectTactic : bool init false;
3      currTactic : [none .. retrain] init none;
4
5      [newEvent] !selectTactic -> (selectTactic'=true)&(currTactic'=none);
6
7      // non-deterministic choice between adaptation tactics
8      [nop] (selectTactic=true) -> 1:(currTactic'=nop)&(selectTactic'=false);
9      [retrain] (selectTactic=true)&(newData > 0) ->
10       1:(currTactic'=retrain)&(selectTactic'=false);
11
12     [tick] (currTactic != none) -> 1:(currTactic' = none);
13 endmodule
14
15 formula tactic_cost = (currTactic = retrain) ? retrainCost : 0;
16 formula fpr_violation_cost = (fpr > FPR_THRESHOLD) ? FPR_COST : 0;
17 formula tpr_violation_cost = (tpr < TPR_THRESHOLD) ? TPR_COST : 0;
18
19 rewards "systemUtility"
20   [tick] true & (time>0) :
21     (tacticLatency * percentTxs * (
22      ((INIT_FPR > FPR_THRESHOLD) ? FPR_COST : 0)
23      + ((INIT_TPR < TPR_THRESHOLD) ? TPR_COST : 0)
24     ) + (1 - tacticLatency) * (1 - percentTxs) *
25      (tacticCost + fpr_violation_cost + tpr_violation_cost));
26 endrewards
```

a clock whose purpose is to keep track of the passing of each time interval. The clock module is implemented as in [11].

Synthesizing optimal adaptation policies. As can be seen in Listing 1, each *tick* of the clock triggers the accrual of a reward. For this specific use-case, the rewards consist of the total costs incurred by the system during that time period due to possible SLA violations and tactics executed. To generate optimal adaptation policies, PRISM requires the specification of a property. Since for this use-case system utility is defined as the total costs incurred by the system and the goal is to minimize these costs, the property that leads to the optimal adaptation policy corresponds to minimizing system utility, which is defined in PRCTL (reward-based property specification logic, c.f. Section II) as $R^{\text{systemUtility}}_{\text{min=?}}[F\ \text{end}]$, which means "*minimum system utility when time 'end' is reached*". '*end*' defines the simulation horizon, i.e., how many future time intervals we want the formal model to simulate.

Extending the tactic's repertoire. To reason about self-adaptation considering more adaptation tactics, the formal model needs to be changed only through the addition of the corresponding tactics' modules such that the adaptation manager can consider them as available when making its nondeterministic choice. This can be performed by adding these tactics to Listing 1, in addition to *nop* and *retrain*.

B. AIPs for the Fraud Detection System

As discussed in Section III-D, the framework instantiates an AIP for each adaptation tactic. In this case since there are two adaptation tactics (retrain and nop), the framework instantiates one AIP for each which is composed of two predictors: one for predicting the increase/decrease in the True Positive Rate (TPR) and a second one to predict the True Negative Rate (TNR). Thanks to the properties of the confusion matrix, by

[1]In PRISM commands are encoded as probabilistic state transitions following the format *[action] guard* $\rightarrow$ *prob*$_1$:*update*$_1$ + ... + *prob*$_n$:*update*$_n$. When $guard$ is true, $update_1$ is applied with probability $prob_1$ (called transition probability). $action$ allows to specify a name for the command or to synchronize commands between modules. Thus, commands with the same $action$ are only triggered when all the $guard$ of all commands is true.

predicting the future TPR and TNR, we can fully characterize the ML component's confusion matrix in the following time interval. These predictions are then provided as inputs to the formal model and leveraged by the probabilistic model checker to synthesize an optimal adaptation strategy.

V. Evaluation

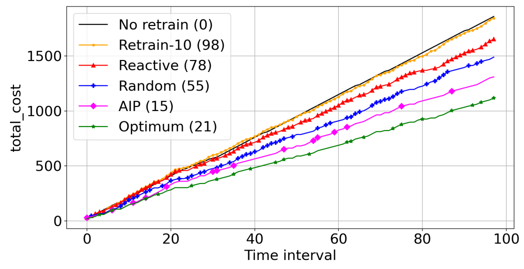
This section validates the usage of the framework for the self-adaptive credit card fraud detection system described in Section IV by evaluating the following research questions:

RQ1 Can the benefits of a model retrain be predicted with acceptable accuracy?
RQ2 Does the proposed approach allow to improve system utility when compared against baselines such as periodic retrains, or reactive policies that retrain the model whenever there is an SLA violation?
RQ3 How are the gains achievable with this approach affected by alternative execution contexts?
RQ4 Is the time complexity of the approach acceptable for a real-time system deployment?

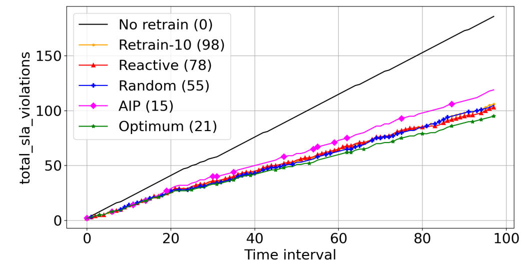
Experimental Settings. We leverage Kaggle's IEEE-CIS Fraud Detection dataset [18] and the winning solution of the challenge [17] as basis for our implementation. We utilize the winning solution to implement the data cleaning, and feature selection tasks. The data splits for training, validation, and test, the self-adaptation mechanisms, and the generation of the retrain benefits dataset are then implemented on top of that base solution. Further, for the purpose of our use-case we leverage only the train dataset of the Kaggle competition, since the test dataset does not have labels of the transactions. The absence of labels would prevent us from being able to assess the performance of the system or the benefits of retraining. Also, we always ensure the transactions of the dataset are given to the models respecting their original time-stamps, as we do not wish to give any advantage to the models by providing them with future information. As such, we use the first 1/3 of the original Kaggle train dataset to train (70%) and validate (30%) the initial fraud detection model. The remaining 2/3 are divided as follows: 70% are used for training and validation of the AIPs (80% and 20%, respectively), and the remaining 30% for testing the framework. Throughout the evaluation, the cost of an SLA violation is fixed to 10 and the AIPs are random-forest predictors of the sklearn package [34] with default parameter values except for the number of trees which we set to 12, similarly to the fraud detection model. The time interval corresponds to 10 hours and the horizon to one future time interval. Our implementation is available at https://github.com/cmu-able/ACSOS22-ML-Adaptation-Framework

Baselines. We consider the following baselines:

- **No-retrain:** the fraud detection model is only trained once, at the beginning of the testing period;
- **Periodic:** the model is retrained at every time interval;
- **Reactive:** the fraud detection model is retrained whenever there is an SLA violation;



(a) Cumulative cost incurred by each baseline.



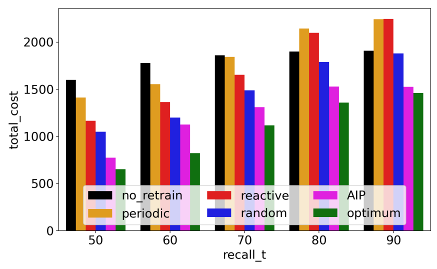
(b) Cumulative SLA Violations incurred by each baseline.

Fig. 2: Utility improvements achievable through the use of the proposed framework. The execution context for this experiment is: fpr threshold = 1, recall threshold = 70, retrain cost = 8, retrain latency = 0. The number of retrains executed by each approach is shown in the legend of each plot, between brackets after the approach's name. The retrains are also represented by the squares in each line.

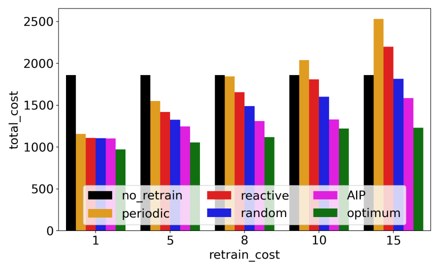
- **Random:** at each time step, there is a 50-50 choice that the model will be retrained;
- **Optimum:** this is the optimal solution which is computed by looking at the actual future results of both retraining and not retraining the model.

A. Utility Improvement due to Retrain

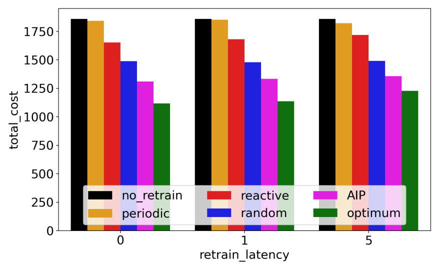
Figure 2 compares the proposed framework (represented by line *AIP*) against the baselines. To evaluate whether the use of the framework allows to improve system utility over baselines that do not explicitly estimate the benefits of retrain, we define the SLA thresholds as RECALL $\geq$ 70% and FPR $\leq$ 1%, fix the retrain latency to 0 and the retrain cost to 8. As Figure 2a shows, by leveraging the proposed framework it is possible to have the fraud detection system minimize its total costs and be closer to the optimal possible cost. This answers **RQ2** and shows that the framework does improve system utility over simpler, model-free baselines due to its ability to estimate the benefits of executing the retrain tactic. As for the number of SLA violations, as shown in Figure 2b, the AIP violates slightly more SLAs than all other baselines except No retrain – which, as expected, is by far the approach that violates the most SLAs. However, as seen previously, this does not translate into higher incurred costs, which is the quality attribute under optimization.



(a) Variation of Recall SLA



(b) Variation of retrain cost



(c) Variation of retrain latency

Fig. 3: Impact of execution context on the total cost incurred.

Recall threshold	Retrain cost	Retrain latency
[50, 60, 70, 80, 90]	[1, 5, 8, 10, 15]	[0, 1, 5]

TABLE I: Values tested for different execution contexts.

Model type	**TPR**		**TNR**	
	MAE	**Corr-coef**	**MAE**	**Corr-coef**
Retrain S	0.1141	0.6811	0.0055	0.7436
Retrain M	0.1158	0.6727	0.0060	0.7086
Retrain L	0.1162	0.6731	0.0059	0.7121
NOP S	0.1343	0.6165	0.0066	0.6142
NOP M	0.1254	0.5793	0.0068	0.6008
NOP L	0.1276	0.5737	0.0068	0.5943

TABLE II: Performance of the AIPs on different sets of features (S, M, L) and evaluated resorting to the mean absolute error (MAE) and to the Pearson correlation coefficient (corr-coef). *NOP* represents the AIPs that estimate the future TPR and TNR when the model is not retrained.

B. Impact of Execution Context

In order to evaluate how different execution contexts impact the need for retrain and answer **RQ3**, we ran experiments for different SLA thresholds, retrain costs, and retrain latencies. Specifically, we tested the values shown in Table I for each dimension, fixing the remaining two dimensions to the values of the base scenario (recall threshold = 70, retrain cost = 8, retrain latency = 0). Figure 3 displays these results.

Regarding the recall threshold (Figure 3a) the results show that, as expected, the cost incurred by the approaches increases as the recall threshold increases. This is justified by the fact that an increase in the recall threshold yields a more difficult problem – the system tolerates less incorrect classifications of fraud transactions. This is translated into an increase of the SLA violations, thus increasing the cost. The optimum and AIP approaches also suffer a cost increase since retraining does not prevent them from violating the thresholds.

Focusing now on the retrain cost (Figure 3b) we see that if the cost is very low, the decision of whether to retrain is fairly trivial and so all approaches that retrain the ML component are close to the optimum. However, as the retrain cost increases, we start to notice how being careful in selecting when to retrain, accounting for the costs and benefits of the tactic, does pay off, as AIP is closer to the optimum than the other approaches. As expected, the no retrain approach is not affected by this dimension.

Finally, regarding the retrain latency dimension, the tested values correspond to percentages of the time interval that are occupied with the process of retrain. That is, retrain latency = 0: retrain is assumed instantaneous; retrain latency = 1: during the first 10% hours of the time interval the model is being retrained and as such transactions are classified using the existing (non-retrained) model. The same rationale applies to retrain latency = 5. The results (Figure 3c) show that this dimension has relatively little impact on the cost of any approach, although as expected the total cost of the optimum solution increases slightly as the retrain latency grows. In fact, even if this baseline can always determine correctly whether it is worth retraining the model at any time t, if the retrain latency grows, a fraction of the transactions in input for the t-th interval will be classified using an old model, thus suffering from an increase in misclassifications and SLA violations.

C. Accuracy of the AIPs

This section answers **RQ1** by evaluating the performance of the AIPs resorting to the mean absolute error (MAE) and to the Pearson correlation coefficient (corr-coef), and considering different sets of features employed by each predictor. Specifically, we consider three different feature sets: S – minimal set with only the basic features (c.f. Section III-D); M – medium set, which includes the basic features and output characteristics; L – encompasses the features of the previous sets and the input characteristics. Table II displays these results. Interestingly, we see that an increase in the size and complexity of the feature set does not yield better AIPs. Additionally, the results also show that the models responsible for predicting the future TPR and TNR when the model is retrained achieve a higher accuracy (lower MAE and higher correlation) than their NOP counterparts (which predict the future TPR and TNR when the model is not retrained). Overall, on the one hand, the accuracy of the AIPs proposed in this work is, as shown in Fig. 3, good enough to allow implementing effective adaptation strategies.

Total (mean)	Total (stdev)	PRISM (mean)	PRISM (stdev)
3.459	0.070	3.113	0.061

TABLE III: Time overhead (secs) of the process of generating the adaptation strategy. The columns named 'PRISM' encompass only the time overhead due to verifying the formal model. The remaining columns display the total time overhead due to the AIPs and to the probabilistic model checking.

On the other hand, the absolute accuracy metrics reported in Table II confirm that predicting the future performance of ML models is far from trivial and that the proposed predictive methodology has still significant margins of improvement (e.g., by identifying different features, blackbox predictors or possibly combining white-box methods [35]).

D. Time Complexity

Since the purpose of the framework is to enable run-time adaptation of ML components in order to improve system utility, we evaluate the time complexity of the process of generating the adaptation strategy. This process corresponds to querying the AIPs and having PRISM verify the property of interest for the formal model of the system. Table III shows the average and standard deviation of the time overhead due to the whole process and also of the formal model verification alone. These values correspond to the execution context defined in Section V-A and were obtained by running the experiments on a machine with an AMD EPYC 7282 CPU@2.8GHz, with 16 cores and 128GB RAM. As can be seen, the process of generating the adaptation strategy takes around 3.5 seconds, which is perfectly affordable considering that retraining ML components has a much higher time overhead. This answers **RQ4** and shows that it is feasible to employ the proposed framework on an online scenario.

E. Threats to Validity

The findings regarding the predictability of the impacts of retrain are dataset- and domain- dependent and so they cannot be generalized to other domains or datasets (external validity). This also applies to the time complexity of the approach, which depends on the complexity of the formal model. Thus, further research is required to understand how the proposed framework and architecture fare in different domains. Also, we have evaluated the use-case for specific execution contexts (regarding system SLAs, tactic cost and latency) which impact the difficulty of the problem (internal validity).

The label assumption required for evaluating the impact of retrain also affects external validity. In fact, real-time performance monitoring is a complex and orthogonal problem. However, it is always possible to get some labels, even if with some delay or at some cost (e.g. via human labeling).

VI. Related Work

ML component retrain. ML model retrain approaches have gained relevance and are being studied by different research fields. In the ML literature, DeltaGrad [36] proposes a method to accelerate the retraining of ML models leveraging information saved during initial model training. Similarly, in the self-adaptive systems literature, T. Chen [37] studies two different types of model retrain (full retrain versus incremental retrain), comparing them in terms of quality and latency. Work on the fraud detection domain has also researched the tradeoffs of full model retrains vs incremental retrains and at different periodicities [38]. Our work differs from these as our goal is to reason on the cost/benefits of generic adaptation tactics targeting ML components (including model retrains) to generate adaptation strategies that maximize an application dependent system utility function. Further, our work can incorporate, in its repertoire of adaptation tactics, incremental retraining techniques such as DeltaGrad. This is achieved by exploiting the proposed AIP construction approach to derive specialized AIPs capable of predicting the benefits (and costs) of this alternative training technique.

Data shift and ML misprediction detection. Recent research work that address the problem of data shift [1], [5]–[8], [39] as well as work that address the problem of detecting ML mispredictions [3], [4], [40], [41] are complementary to our work and provide useful solutions that can be employed to improve our framework, for instance by querying the AIPs and the model checker only when shift or mispredictions have been detected by these approaches.

ML in self-adaptive systems. Recent work in the field of self-adaptive systems has shown an ever-growing trend for the use of ML techniques in self-adaptive systems' managers to improve their self-adaptation capabilities [42]–[44]. Researchers have also argued for the need for a tighter relationship between self-adaptation and AI, such that they can "benefit from and improve one another" [45]. Recent work has started to explore the area with Gheibi et al. [46] proposing a framework for lifelong self-adaptation that allows a ML-based self-adaptation manager to react to drifts in the data and learn new tasks. Similarly, the work of Langford et al. [47] proposes a framework to monitor learning enabled systems and evaluate their compliance with the required objectives. Differently from these works, our framework aims to decide whether to adapt a ML component by reasoning about the cost-benefit tradeoffs of the available tactics.

In our previous work, we have identified a set of ML adaptation tactics to deal with ML mispredictions [22]. Also, a sketch of the framework presented in Section III has been outlined in [48]. This paper builds on previous work and extends them in a number of ways. First, we redefine several key aspects of the framework, including redesigning the interface of the ML component, and introduce a methodology to build blackbox predictors of the impact of adaptations targeting ML components. Further, this work validates the effectiveness of the framework with a use-case based on a realistic data set and complex ML models.

VII. Conclusion

This work proposes a self-adaptation framework for ML-based systems. We proposed a strategy for formally modeling the behavior and state evolution of ML components in order to leverage probabilistic model checking techniques and synthesize optimal adaptation strategies. We presented a general approach for generating blackbox predictors that estimate the impact of adaptation strategies on the ML component. We instantiated the proposed framework on a use-case from the credit card fraud detection domain and showed that reasoning about the cost-benefit tradeoffs of retraining ML components allows for better adaptation decisions when compared against model-free baselines. As future work, we plan to: validate the framework in a broader range of domains; study the impact of resorting to different model types to instantiate the AIPs; and investigate the use of alternative modeling techniques and feature engineering approaches to develop more accurate AIPs.

References

[1] J. Yang, K. Zhou, Y. Li, and Z. Liu, "Generalized out-of-distribution detection: A survey," *arXiv preprint arXiv:2110.11334*, 2021.

[2] J. Quionero-Candela, M. Sugiyama, A. Schwaighofer, and N. Lawrence, *Dataset shift in machine learning*. The MIT Press, 2009.

[3] J. Cito, I. Dillig, S. Kim, V. Murali, and S. Chandra, "Explaining mispredictions of machine learning models using rule induction," in *Procs. of ESEC/FSE*, 2021.

[4] D. Hendrycks and K. Gimpel, "A baseline for detecting misclassified and out-of-distribution examples in neural networks," *arXiv preprint arXiv:1610.02136*, 2016.

[5] A. Rabanser, S. Günneman, and Z. Lipton, "Failing loudly: An empirical study of methods for detecting dataset shift," in *Procs. of NIPS*, 2019.

[6] Y. Ovadia, E. Fertig, J. Ren, Z. Nado, D. Sculley, S. Nowozin, J. Dillon, B. Lakshminarayanan, and J. Snoek, "Can you trust your model's uncertainty? evaluating predictive uncertainty under dataset shift," in *Procs. of NIPS*, 2019.

[7] I. Žliobaitė, "Learning under concept drift: an overview," *arXiv preprint arXiv:1010.4784*, 2010.

[8] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia, "A survey on concept drift adaptation," *ACM CSUR*, vol. 46, no. 4, 2014.

[9] B. Cheng *et al.*, *Software Engineering for Self-Adaptive Systems: A Research Roadmap*. Springer, 2009.

[10] R. de Lemos *et al.*, "Software engineering for self-adaptive systems: A second research roadmap," in *Software Engineering for Self-Adaptive Systems II*. Springer, 2013.

[11] G. Moreno, J. Cámara, D. Garlan, and B. Schmerl, "Proactive self-adaptation under uncertainty: A probabilistic model checking approach," in *Procs. of ESEC/FSE*, 2015.

[12] R. Calinescu *et al.*, "Synthesis and verification of self-aware computing systems," in *Self-Aware Computing Systems*. Springer, 2017.

[13] J. Cámara, W. Peng, D. Garlan, and B. Schmerl, "Reasoning about sensing uncertainty and its reduction in decision-making for self-adaptation," *Science of Computer Programming*, vol. 167, 2018.

[14] G. Moreno, J. Cámara, D. Garlan, and M. Klein, "Uncertainty reduction in self-adaptive systems," in *Procs. of SEAMS*, 2018.

[15] M. Casimiro, D. Didona, P. Romano, L. Rodrigues, W. Zwaenepoel, and D. Garlan, "Lynceus: Cost-efficient tuning and provisioning of data analytic jobs," in *Procs. of ICDCS*, 2020.

[16] N. Yadwadkar, B. Hariharan, J. Gonzalez, B. Smith, and R. Katz, "Selecting the best vm across multiple public clouds: A data-driven performance modeling approach," in *SoCC*, 2017.

[17] (2019) Ieee-cis fraud detection winner solution. [Online]. Available: https://www.kaggle.com/code/cdeotte/xgb-fraud-with-magic-0-9600

[18] (2019) Ieee-cis fraud detection. [Online]. Available: https://www.kaggle.com/competitions/ieee-fraud-detection/overview

[19] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *Procs. of CAV'11*, ser. LNCS, vol. 6806. Springer, 2011.

[20] ——, "Probabilistic model checking and autonomy," *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 5, 2022.

[21] S. Andova, H. Hermanns, and J.-P. Katoen, "Discrete-time rewards model-checked," in *In Procs. of FORMATS*. Springer, 2003.

[22] M. Casimiro, P. Romano, D. Garlan, G. Moreno, E. Kang, and M. Klein, "Self-adaptation for machine learning based systems." in *ECSA (Companion)*, 2021.

[23] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, 2003.

[24] B. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. Al Sallab, S. Yogamani, and P. Pérez, "Deep reinforcement learning for autonomous driving: A survey," *IEEE T-ITS*, 2021.

[25] F. Pinto, M. Sampaio, and P-Bizarro, "Automatic model monitoring for data streams," *arXiv preprint arXiv:1908.04240*, 2019.

[26] Y. Lucas and J. Jurgovsky, "Credit card fraud detection using machine learning: A survey," *CoRR*, vol. abs/2010.06479, 2020.

[27] B. Erickson, P. Korfiatis, Z. Akkus, and T. Kline, "Machine learning for medical imaging," *Radiographics*, vol. 37, no. 2, 2017.

[28] J. Townsend, "Theoretical analysis of an alphabetic confusion matrix," *Perception & Psychophysics*, vol. 9, no. 1, 1971.

[29] R. Calinescu, R. Mirandola, D. Perez-Palacin, and D. Weyns, "Understanding uncertainty in self-adaptive systems," in *Procs. of ACSOS*, 2020.

[30] S. Hezavehi, D. Weyns, P. Avgeriou, R. Calinescu, R. Mirandola, and D. Perez-Palacin, "Uncertainty in self-adaptive systems: A research community perspective," *ACM TAAS*, vol. 15, no. 4, 2021.

[31] D. Keefer, "Certainty equivalents for three-point discrete-distribution approximations," *Management science*, vol. 40, no. 6, 1994.

[32] M. Menéndez, J. Pardo, L. Pardo, and M. Pardo, "The jensen-shannon divergence," *Journal of the Franklin Institute*, vol. 334, no. 2, 1997.

[33] C. Bishop and N. Nasrabadi, *Pattern recognition and machine learning*. Springer, 2006, vol. 4, no. 4.

[34] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[35] D. Didona, F. Quaglia, P. Romano, and E. Torre, "Enhancing performance prediction robustness by combining analytical modeling and machine learning," in *Procs. of ACM/SPEC ICPE*, 2015.

[36] Y. Wu, E. Dobriban, and S. Davidson, "DeltaGrad: Rapid retraining of machine learning models," in *Procs. of ICML*, 2020.

[37] T. Chen, "All versus one: An empirical comparison on retrained and incremental machine learning for modeling performance of adaptable software," in *Procs. of SEAMS*, 2019.

[38] B. Lebichot, G. Paldino, W. Siblini, L. He-Guelton, F. Oblé, and G. Bontempi, "Incremental learning strategies for credit cards fraud detection," *International Journal of Data Science and Analytics*, vol. 12, no. 2, pp. 165–174, 2021.

[39] J. Perdomo, T. Zrnic, C. Mendler-Dünner, and M. Hardt, "Performative prediction," in *Procs. of ICML*, 2020.

[40] Y. Xiao, I. Beschastnikh, D. S. Rosenblum, C. Sun, S. Elbaum, Y. Lin, and J. S. Dong, "Self-checking deep neural networks in deployment," in *Procs. of ICSE*, 2021.

[41] P. Kourouklidis, D. Kolovos, J. Noppen, and N. Matragkas, "A model-driven engineering approach for monitoring machine learning models," in *Procs. of MODELS-C*. IEEE, 2021.

[42] T. Saputri and S.-W. Lee, "The application of machine learning in self-adaptive systems: A systematic literature review," *IEEE Access*, vol. 8, 2020.

[43] O. Gheibi, D. Weyns, and F. Quin, "Applying machine learning in self-adaptive systems: A systematic literature review," *ACM TAAS*, vol. 15, no. 3, 2021.

[44] D. Weyns, B. Schmerl, M. Kishida, A. Leva, M. Litoiu, N. Ozay, C. Paterson, and K. Tei, "Towards better adaptive systems by combining mape, control theory, and machine learning," in *Procs. of SEAMS*, 2021.

[45] T. Bureš, "Self-adaptation 2.0," in *Procs. of SEAMS*, 2021.

[46] O. Gheibi and D. Weyns, "Lifelong self-adaptation: Self-adaptation meets lifelong machine learning," in *Procs. of SEAMS*, 2022.

[47] M. Langford, K. Chan, J. Fleck, P. McKinley, and B. Cheng, "Modalas: Model-driven assurance for learning-enabled autonomous systems," in *Procs. of MODELS*, 2021.

[48] M. Casimiro, D. Garlan, J. Cámara, L. Rodrigues, and P. Romano, "A probabilistic model checking approach to self-adapting machine learning systems," in *Procs. of ASYDE, Co-located with SEFM 2021*, 2021.