Self-Adapting Machine Learning-based Systems via a Probabilistic Model Checking Framework

MARIA CASIMIRO, S3D, Carnegie Mellon University, USA and IST, Universidade de Lisboa, Portugal
DIOGO SOARES, INESC-ID, IST, Universidade de Lisboa, Portugal
DAVID GARLAN, S3D, Carnegie Mellon University, USA
LUÍS RODRIGUES, INESC-ID, IST, Universidade de Lisboa, Portugal
PAOLO ROMANO, INESC-ID, IST, Universidade de Lisboa, Portugal

This paper focuses on the problem of optimizing system utility of Machine-Learning (ML) based systems in the presence of ML mispredictions. This is achieved via the use of self-adaptive systems and through the execution of adaptation tactics, such as *model retraining*, which operate at the level of individual ML components.

To address this problem, we propose a probabilistic modeling framework that reasons about the cost/benefit trade-offs associated with adapting ML components. The key idea of the proposed approach is to decouple the problems of estimating **(i)** the expected performance improvement after adaptation and **(ii)** the impact of ML adaptation on overall system utility.

We apply the proposed framework to engineer a self-adaptive ML-based fraud-detection system, which we evaluate using a publicly-available, real fraud detection data-set. We initially consider a scenario in which information on model's quality is immediately available. Next we relax this assumption by integrating (and extending) state-of-the-art techniques for estimating model's quality in the proposed framework. We show that by predicting the 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 or reactive retraining.

CCS Concepts: • **Software and its engineering** → **Model checking**; • **Computing methodologies** → **Machine learning**; **Model development and analysis**; • **Computer systems organization** → *Dependable and fault-tolerant systems and networks.*

Additional Key Words and Phrases: self-adaptation, machine learning, model retrain, fraud detection system

ACM Reference Format:
Maria Casimiro, Diogo Soares, David Garlan, Luís Rodrigues, and Paolo Romano. 2023. Self-Adapting Machine Learning-based Systems via a Probabilistic Model Checking Framework. *ACM Trans. Autonom. Adapt. Syst.* 1, 1, Article 0 (February 2023), 29 pages. https://doi.org/XXXXXXX.XXXXXXX

1 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

Authors' addresses: Maria Casimiro, mdaloura@andrew.cmu.edu, S3D, Carnegie Mellon University, Pittsburgh, Pensylvania, USA and IST, Universidade de Lisboa, Lisbon, Portugal; Diogo Soares, diogo.sousa.soares2000@gmail.com, INESC-ID, IST, Universidade de Lisboa, Lisbon, Portugal; David Garlan, garlan@cs.cmu.edu, S3D, Carnegie Mellon University, Pittsburgh, Pensylvania, USA; Luís Rodrigues, ler@tecnico.ulisboa.pt, INESC-ID, IST, Universidade de Lisboa, Lisbon, Portugal; Paolo Romano, romano@inesc-id.pt, INESC-ID, IST, Universidade de Lisboa, Lisbon, Portugal.

ACM 1556-4665/2023/2-ART0
https://doi.org/XXXXXXX.XXXXXXX

production is inherently affected by the training data used for its creation, 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 triggering the need for 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 [67] and can be caused for instance by co-variate shift (i.e., shifts of the input features) and concept drift (i.e., shift in the relationship between input feature and the target variable) [56]. OOD samples are thus a common cause of ML mispredictions [20, 32] and while these problems and how to detect their occurrence have been extensively studied by the ML literature [26, 50, 57, 69], 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 [19, 22], which are systems capable of reacting to environment changes in order to optimize or 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 [48]) could ideally be applied to ML-based systems as a mechanism to deal with possible mispredictions.

While previous work in the self-adaptive systems literature [6, 8, 49] 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 trade-offs allows a planner to select one tactic over another, or more generally one adaptation strategy over another. However, due to the 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 [11, 66], 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 trade-offs 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 and present a general strategy for creating models that predict these benefits. The latter problem is solved by expressing inter-component dependencies via an architectural model, which enables automated reasoning via model checking techniques.

Further, in some domains the environment may not provide immediate feedback regarding the outcome of an event, thus preventing the system from gathering up-to-date ground truth labels to evaluate the ML model's predictive quality in real-time. To address this issue and demonstrate the applicability of the proposed framework in scenarios in which ground truth labels are assumed to become available only after a non-negligible time interval d, we integrate in the proposed framework a state-of-the-art approach for estimating the predictive quality of ML models (ATC) [27]. We also propose a novel variant, named CB-ATC, which addresses some shortcomings that we identified while integrating ATC in our framework.

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 [2] and evaluate it using a public data set [1], 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 an ML component, a self-adaptation manager can generate adaptation strategies that are closer to the optimal one when compared against baselines such as periodic or reactive retrains (triggered upon an SLA violation).

We also evaluate the impact of label delay on the efficiency of the self-adaptation strategies output by our framework, using both ATC and CB-ATC. Our study highlights the superiority of CB-ATC in estimating an ML model's predictive quality with respect to ATC, a state-of-the-art approach. It also points out that the usage of estimated (and hence inherently approximate) information about a model's predictive quality can have a detrimental effect on the ability to accurately predict the impact of adapting (i.e., retraining vs not-retraining) ML components. This, in turn, can hinder the efficiency of the self-adaptive system especially when ground truth labels are subject to large delays. Still, our experiments show that model estimation techniques, like CB-ATC, do represent a useful asset to enhance robustness in the presence of small label delays (1 day in our case study).

This article extends our previous paper [15] by introducing the following main contributions:

- We lifted the assumption that labels are immediately available (which allows for precisely estimating the current predictive quality of ML components) and tested techniques to address scenarios where ground-truth labels become available with non-negligible delays. More in detail, we integrated in our framework ATC [27], a state-of-the-art technique for model's quality estimation (see Section 3.2.1) and identified relevant shortcomings that arise when employing this technique in scenarios analogous to the one considered in our case study. This led us to propose a novel model's quality estimation technique, named CB-ATC (see Section 4.5.1), which we empirically show to provide more accurate estimates than ATC (see Section 6.5).
- We introduced architecture diagrams that detail the interaction between the Analyze and Plan components in order to clarify the overall organization and operation of the self-adaptation manager (see Section 4.1);
- We improved the description of the background on probabilistic model checking and clarified key concepts, such as "system utility" (see Section 3);
- We extended the discussion on generality, applicability, and limitations of the proposed framework (see Section 6.7).

The remainder of this article is organized as follows: Section 2 discusses related work; Section 3 presents the required background on probabilistic model checking (Section 3.1) and on methods to estimate ML model performance in the absence of ground truth labels (Section 3.2); Section 4 introduces the proposed framework for self-adaptation of ML-based systems; Section 5 discusses how to apply the proposed framework to a use case based on an ML-based credit card fraud detection system; Section 6 evaluates the usage of the framework for the use case and discusses current. Finally, Section 7 concludes the paper with a set of directions for future work.

2 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 [64] 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 [17] 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 trade-offs of full model retrains vs incremental retrains and at different periodicities [43]. 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 can be achieved by exploiting the proposed approach to construct estimators of the benefits of executing specific tactics (Section 4.4) to derive specialized predictors capable of estimating 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 [26, 50, 53, 57, 67, 69] as well as work that address the problem of detecting ML mispredictions [20, 32, 39, 65] are complementary to our work and provide useful solutions that can be employed to improve our framework, for instance triggering adaptation when shift or mispredictions have been detected by these approaches.

ML in self-adaptive systems. Many self-adaptive systems are now using ML techniques to improve their self-adaptation capabilities [29, 58, 63]. 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" [5]. Recent work has started to explore the area with Gheibi et al. [28] proposing a framework for lifelong self-adaptation that allows an ML-based self-adaptation manager to react to drifts in the data and learn new tasks. Similarly, the work of Langford et al. [42] 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 an ML component by reasoning about the cost-benefit trade-offs of the available adaptation tactics.

In our previous work, we have identified a set of ML adaptation tactics to deal with ML mispredictions [13, 14]. Specifically, these works presented a repertoire of adaptation tactics, illustrating under which conditions each tactic could be applied resorting to two use-cases from the enterprise and cyber-physical systems domains. Also, a sketch of the framework presented in Section 4 has been outlined in [12]. This paper builds on previous work and extends them in a number of ways. First, we redefine several key aspects of the framework proposed in [12], 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.

3 BACKGROUND

This section provides a brief introduction to the areas of probabilistic model checking and real time monitoring of ML model predictive performance.

3.1 Probabilistic Model Checking

Probabilistic model checking is a formal technique based on methods for reasoning about and analyzing systems that exhibit probabilistic and uncertain behavior. This approach has been extensively explored in the self-adaptive systems literature [6, 8, 49] 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* [49]. 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 [40], which is a probabilistic model checker commonly used in the literature [41, 49]. We define the formal models as Markov Decision Processes (MDPs) [55], 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. More formally, a MDP is defined by a 4-tuple $(\mathcal{S}, \mathcal{A}, P_a, R_a)$, where $\mathcal{S}$ is the set of states, $\mathcal{A}$ is the set of actions, P_a is the probability matrix that gives the probability of transitioning to state s' from state s by executing action a at time t. R_a is the reward associated to this transition. 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 optimizing system utility. The definition of system utility is domain dependent: it can be associated for example with the cost of operating a system, the user experience, or the energy consumption.

To specify these properties, PRISM relies on probabilistic computation-tree logic (PCTL) [31], which is an extension of Computation-Tree Logic (CTL) [21]. CTL is a formal language for specifying well-formed formulas (WFF) according to a pre-defined grammar. These formulas are then used by model checkers to verify system properties, typically expressed in terms of liveness and/or safety. Examples of temporal CTL operators are **F** and **X**, which specify that a given WFF *eventually* has to hold, or has to hold *at the next state*, respectively. PCTL extends CTL allowing the definition of formulas that account for probabilities associated with state transitions. By exploiting the fact that it is possible to associate rewards with specific states or state transitions in the formal model, and by specifying reward-based properties as a function of state-specific constraints of the system, PRISM generates optimal strategies that comply with these constraints. To specify such properties in PRISM using PCTL, one would leverage PRISM's *R* operator, which informs PRISM that we want to reason about ***R**ewards*. Such properties are expressed as **R** QUERY [REWARD_PROPERTY]. QUERY can be set to any of [=?, *min* =?, *max* =?] and allows the user to specify whether PRISM should find the exact, minimum or maximum value of the rewards for the reward property REWARD_PROPERTY. Finally, REWARD_PROPERTY is defined in terms of CTL operators, such as the operators **F** and **X** described above. For example, REWARD_PROPERTY can be defined as

[REWARD_PROPERTY] = [**F** PROP] $\Leftrightarrow$ [REWARD_PROPERTY] = [**F** $z = 2$].

This would lead PRISM to compute the reward accumulated along a path until the system eventually reaches a state satisfying PROP (i.e., a state where variable z is 2).

3.2 Real Time ML Model Quality Monitoring

While it is fairly trivial to compute values for features such as **(i)** the amount of new samples with which the ML model has not been trained and **(ii)** the time elapsed since the model was last retrained, computing the model's real time predictive performance may not be as straightforward. This is particularly the case in contexts for which ground truth labels may take a non-negligible time to become available. Without these labels, we only have access to a set of unlabeled samples, model predictions, and delayed (outdated) labeled samples, thus rendering it challenging to estimate a model's predictive quality in real-time.

To address this challenge, the ML literature has proposed numerous approaches to estimate the quality of models' predictions, [16, 16, 27, 35]. Existing methods can be coarsely classified according to the type of ML models that they target (e.g., deep neural networks vs generic ML predictors) and to their ability to estimate aggregate quality on an entire (unlabeled) test set [30] vs identify misclassified test inputs [27].

These approaches typically leverage the probability distribution that a model outputs[1] to gain insights into the level of uncertainty associated with each prediction. This uncertainty thus constitutes a meaningful proxy to estimate model performance/error. For instance, the approach by Jiang et al. [35] targets deep neural networks and estimates the error on unlabeled test sets by measuring the disagreement rate of instances of the same network trained with a different run of Stochastic Gradient Descent (SGD). Another approach is to develop self-trained ensembles of deep networks that predict which inputs will be misclassified by the classifier based on (known) errors in the training data-set [16]. Other works, like Average Thresholded Confidence (ATC) [27], take a model-agnostic approach and can estimate the correctness of individual model predictions — and then use the point-wise predicted labels to compute aggregate estimates of a model's predictive quality (e.g., class error rate) as well as of ground truth class probabilities.

By leveraging such techniques, our framework can be applicable to a wide range of domains, without requiring assumptions on ground truth label availability. Leveraging such techniques increases the framework's applicability and generalizability. However, the framework's accuracy is ultimately dependent on the accuracy of both **(i)** the predictors for estimating the benefits of executing each adaptation tactic and **(ii)** of the approaches employed for estimating the current quality of the ML model.

3.2.1 *Average Thresholded Confidence (ATC).*

For self-containment, we provide an overview of ATC. We start by introducing preliminary terminology. Let f be a k-class classifier and $f_k(x)$, $\forall k \in \mathcal{Y}$, the predicted probability of an input x belonging to class k, according to classifier f. ATC requires a score function, $s : [0,1]^k \to \mathbb{R}$, which takes as input the k-dimensional vector of probabilities output by classifier f, and outputs a real number. The score function s captures the confidence of classifier f in its prediction and is used to estimate the expected mis-classification rate. As such, the score function s is chosen such that if the classifier predicts that the output class for an input x is $i \in \mathcal{Y}$ with high probability relatively to the other classes, then $s(f_i(x))$ should be high: $f_i(x) >> f_j(x), \forall j \neq i \implies s(f_i(x)) > s(f_j(x))$.

Conversely, if the classifier predicts class i for input x with relatively low probability, then $s(f_i(x))$ should be low: $f_i(x) \leq f_j(x), \forall j \neq i \implies s(f_i(x)) < s(f_j(x))$.

ATC [27] considers two score functions: Maximum confidence — $s(f(x)) = \max_{j \in \mathcal{Y}} f_j(x)$; and Negative Entropy — $s(f(x)) = \sum_j f_j(x) \log\left(f_j(x)\right)$. Based on its results, we use only the Negative Entropy function in this study.

[1] The model predicts a probability of each class being the correct class for the sample.

Algorithm 1 Logic employed by ATC to determine the decision threshold t

```
procedure FIND_THRESHOLD($\mathcal{D}^{val}$)
    $Err(\mathcal{D}^{val}) \leftarrow \frac{||misclassified\ inputs||}{||inputs||}$        ▷ Compute the error rate on the validation set
    return Percentile$^{Err(\mathcal{D}^{val})}$(s(f(x)))        ▷ Return the $Err(\mathcal{D}^{val})$-th percentile of the score distribution for the validation set
end procedure
```

ATC estimates a model's predictive quality as follows: given a validation set $\mathcal{D}^{val}$ of labeled data (e.g., which includes the most recent ground truth labels for the past predictions of the classifier f) ATC identifies a threshold t on $\mathcal{D}^{val}$ such that the number of samples that obtain a score less than t match the number of errors of the classifier f on $\mathcal{D}^{val}$. This procedure is illustrated by the pseudo-code in Algorithm 1 and can be expressed compactly as follows:

$$\mathbb{E}_{x\sim\mathcal{D}^{val}}\left[\mathbb{I}\left[s(f(x)) < t\right]\right] = \mathbb{E}_{(x,y)\sim\mathcal{D}^{val}}\left[\mathbb{I}\left[\text{argmax}_{j\in\mathcal{Y}}\, f_j(x) \neq y\right]\right], \tag{1}$$

where $\mathbb{I}$ denotes the indicator function, and y the ground truth class for input x. The left side of the equation defines the ratio of samples in $\mathcal{D}^{val}$ with score below the threshold t and the right side specifies the error rate for $\mathcal{D}^{val}$ (i.e., number of errors of the classifier f on $\mathcal{D}^{val}$), which we also note as $Err(\mathcal{D}^{val})$. Threshold t can be easily computed as it corresponds to the $Err(\mathcal{D}^{val})$-percentile of the distribution of scores for $\mathcal{D}^{val}$. One can then estimate the correctness of individual predictions on a target, unlabeled data-set $\mathcal{D}^{\mathrm{T}}$ based on whether each prediction's score is above/below t. The error rate for $\mathcal{D}^{\mathrm{T}}$ can thus be computed as:

$$Err(\mathcal{D}^{\mathrm{T}}(s)) = \mathbb{E}_{x\sim\mathcal{D}^{\mathrm{T}}}\left[\mathbb{I}[s(f(x)) < t]\right]. \tag{2}$$

Note that ATC allows for estimating whether individual predictions are correct or not. In the general multi-class scenario ($k > 2$), this does not allow to pinpoint which class is expected to be the correct one, in case a classifier's prediction is deemed as incorrect. However, since our use case is a binary classification problem, estimating the correctness of a prediction implies also determining which is the expected class in case the classifier's prediction is estimated to be incorrect. This allows us to employ ATC to estimate the whole confusion matrix as well as the fraud rate.

4 FRAMEWORK FOR ML ADAPTATION

This section describes 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 **(iii)** how to integrate these predictions with the formal model.

4.1 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 trade-offs 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**

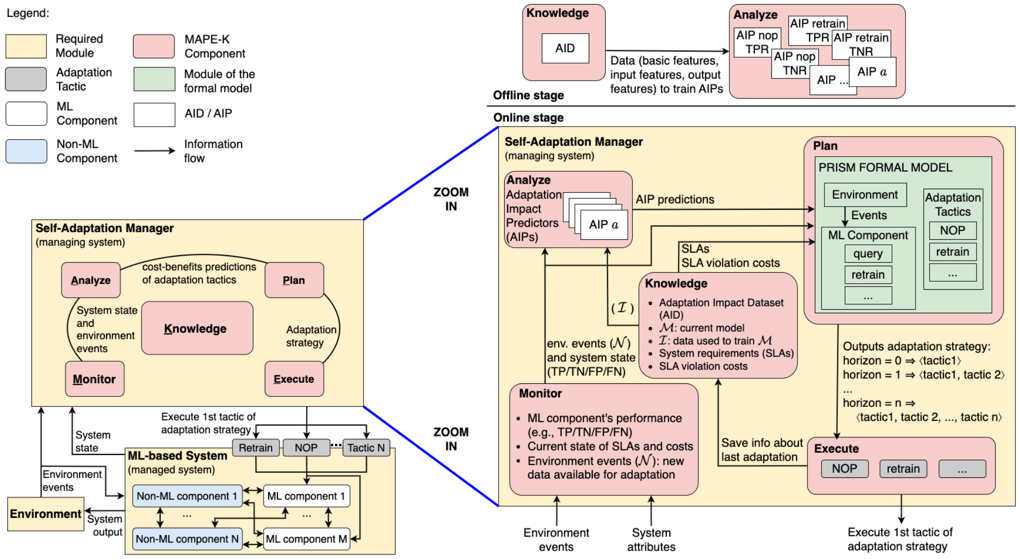


Fig. 1. Architecture of the self-adaptation framework.

– designed to be easily extended to incorporate additional adaptation tactics (e.g., transfer learning, unlearning), as discussed by [13], and customized to capture application specific dynamics.

Our framework makes the following assumptions:

A1 There are fluctuations of the ML model's quality over time – ML techniques are inherently approximate (they can mispredict even in the absence of changes); or the system may be operating under changing environments which lead to ML mispredictions (data shift);

A2 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 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, an optimal adaptation strategy is synthesized to be used 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.

4.2 Self-Adaptation Manager's Architecture

Similarly to previous work [6, 8, 49], the proposed framework leverages a self-adaptation manager that adopts a MAPE-K [37] 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 ML-based 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. ML-based systems include, for instance, financial fraud detection [3, 54] and machine translation systems [23, 59]. These systems normally rely on both ML and non-ML components to fulfil their objectives, namely detecting fraudulent transactions and translating sentences, respectively. A system component that incorporates an ML model is considered an ML component. Such components can be adapted via the execution of the tactic selected by the self-adaptation manager.

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 (referred to as Adaptation Impact Predictors or AIPs, introduced in Section 4.4), which leverage historical data of previous adaptations and of their impact on the ML component's quality (e.g., accuracy) 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 optimizes system utility.

The self-adaptation manager triggers the execution of any of the tactics available to adapt the system. Although the diagram in Figure 1 considers two example tactics (retraining ML components or NOP – not performing any adaptation), as discussed in our prior work [13], the ML literature has proposed a number of approaches that can be leveraged as adaptation tactics, such as:

- Unlearning [9, 10]: useful when data in the ML model's knowledge base no longer represents the environment or contributes to increased model quality. Removes unwanted samples faster and more efficiently than by retraining the model without those samples;
- Transfer learning [34, 51]: helpful if ML model $M1$ is expected to have to deal with environmental conditions that a different ML model $M2$ has dealt with. Leverages data from $M2$ so that $M1$ learns how to react/predict for the (expected) upcoming and previously unknown/unseen scenario/context.
- Human based labelling [62]: convenient when the ML model has high uncertainty and the decision/prediction is critical, or has high impact on the system. Offloads decision to human operator/user, who can typically offer more assurance (especially in the case of expert users/operators).

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.

4.3 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 [38], in the context of fraud detection [44, 54] or medical diagnosis systems [25] 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 [61], 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 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 [13] (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*. The ML component interface can be extended to incorporate additional adaptation tactics (e.g., transfer learning [51] or unlearning [9, 10]), if those are indeed available for the managed system. As the name suggests, *retrain* 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 in Section 4.4. 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 $\mathbf{O}$ (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 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. C^* can be simply computed by multiplying each row i of the normalized confusion matrix 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 [7, 33, 49], 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. [48] 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) [36] 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.

4.4 Predicting the Effects of Adapting and Not Adapting

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. 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 trade-offs.

We focus on the problem of how to predict the performance evolution of the ML component and describe, in the next section, how we tackle the problem of implementing AIPs for the retrain and NOP tactics for generic ML components. Indeed, for 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 [11] based on observing previous retraining procedures, possibly mixed with white-box methods [66], can generate accurate predictive models of the retrain cost.

Predicting future quality of ML components. Given the reliance on a row-normalized confusion matrix C to characterize the performance of ML components, predicting their performance evolution requires estimating how 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 $\mathcal{M}' \leftarrow a(\mathcal{M}, \mathcal{I}, \mathcal{N})$ 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 with data-set $\mathcal{I}$, 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 data-set. In order to train such a blackbox regressor, we build an Adaptation Impact Data-set (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 of monitoring model $\mathcal{M}_0$ of a ML component in production over T time intervals. During this 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 $C_i(j)$. 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 $j > i$. This testing platform is required to support the data pre-processing pipeline, model building, and inference stages of the ML components targeted by the adaptation. Such testing platform is then leveraged by the framework to create and evaluate different versions of these components, eschewing the need to reproduce the full production system, comprising the whole set of ML and non-ML components. We expect such testing platforms to be normally available due to the common DevOps/MLOps practice [60] of testing ML models' quality prior to their actual deployment in production.

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 **(BF1)** 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., $\mathcal{N}_j$; **(BF2)** the predictive quality of the model shortly after its generation and at the present time; **(BF3)** the time elapsed since the last execution of the adaptation tactic, i.e., $j - i$; **(BF4)** the ground truth distribution of classes at the time model $\mathcal{M}_i$ was generated and at the present time.
- *Output Characteristics Features:* describe variations in 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 data-sets $\mathcal{I}_i$ and $\mathcal{N}_j$. The current version of the framework computes, for each feature f, the Pearson correlation coefficient between its values in $\mathcal{I}_i$ and $\mathcal{N}_j$. However, other metrics could also be used to detect shifts in the input distributions, e.g., using different distributional distances like Jensen-Shannon divergence (JSD) [46] or Kolmogorov [45].

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 [46] (inspired by previous work [54]), 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 data-set 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 data-set to train a set of independent Adaptation Impact Predictors (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, since each row must sum to 1. For the case of binary classification, where $n = 2$, 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 [4] 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 would be beneficial for cases when the model checker is used to reason on a look-ahead horizon of $l > 1$ time intervals. In such a case, up to a^l possible adaptation strategies are generated, where a is the number of adaptation tactics available, thus 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.

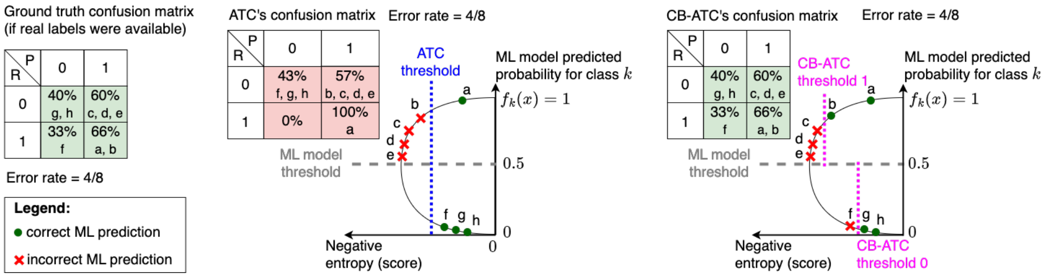
4.5 Accounting for Label Delay

In general, label delay can be thought of as a form of temporal misalignment between the data and the labels. More formally, for a delay d and current timestamp t, ground truth labels are available for transactions completed up to time instant $t - d$. This means that in order to estimate the current quality of the model, one either resorts to (i) delayed labels, hence using stale data as a proxy for the current model's quality, or (ii) to methods (see Section 3.2) that aim to predict the current model's predictive quality in the absence of labelled data.

To test the proposed framework, which requires knowledge of both the current performance of the ML model and the distribution of the target (features BF2 and BF4, c.f. Section 4.4) in order to estimate the impact of an adaptation, we explore three different approaches to label estimation. The first, less realistic approach, assumes that **(i)** labels are immediately available. As this is not typically the case in reality, for our second and third approaches we relax this assumption by **(ii)** leveraging only delayed labels and **(iii)** resorting to methods for predicting model performance under unseen data distributions. Among the approaches described in Section 3.2, we use our own variant of the Average Thresholded Confidence (ATC) method proposed in [27], that we name Class-Based ATC (CB-ATC). We build on ATC due to its reduced computational complexity and to its ability to estimate the individual ground-truth labels of point-wise model predictions.

4.5.1 Class-Based-ATC (CB-ATC). While integrating ATC within our framework we identified two relevant shortcomings, which led us to propose a new method: CB-ATC. The next paragraphs, with the aid of Figures 2 and 3, illustrate ATC's limitations and describe how CB-ATC circumvents them.

Limitation 1. The first limitation of ATC is related to its (implicit) assumption on the distributions of scores for the correct/incorrect predictions of each class being "similar enough" so that by using a single threshold, it is possible to fit the error rate on validation data accurately for both predicted classes. However, if depending on the predicted class the scores of incorrect/correct predictions are distributed in different regions, as illustrated by Figure 2, ATC is unable to correctly fit the confusion matrix using a single threshold. Intuitively, CB-ATC addresses this limitation by



Ground truth confusion matrix

R \ P	0	1
0	40% g, h	60% c, d, e
1	33% f	66% a, b

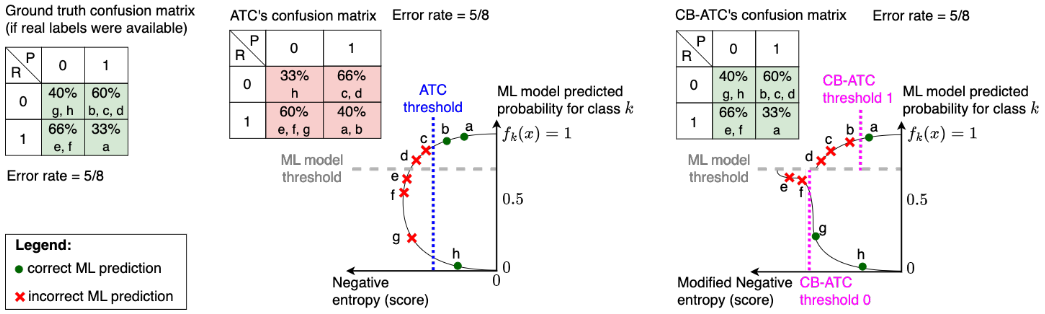
ATC's confusion matrix

R \ P	0	1
0	43% f, g, h	57% b, c, d, e
1	0%	100% a

CB-ATC's confusion matrix

R \ P	0	1
0	40% g, h	60% c, d, e
1	33% f	66% a, b

Fig. 2. ATC (middle plot) is unable to correctly fit the actual confusion matrix (left plot; 'r' stands for real/actual ground truth; 'p' stands for predicted labels) of the validation data considered in this example via a single threshold. This is due to the different characteristics of the distributions of scores for the samples that are predicted as class 0 and 1 (i.e., below and above the assumed 0.5 model threshold). By using one threshold per class, CB-ATC (right plot) accounts for the different distributions of scores in each class and fits the actual confusion matrix precisely.



Ground truth confusion matrix

R \ P	0	1
0	40% g, h	60% b, c, d
1	66% e, f	33% a

ATC's confusion matrix

R \ P	0	1
0	33% h	66% c, d
1	60% e, f, g	40% a, b

CB-ATC's confusion matrix

R \ P	0	1
0	40% g, h	60% b, c, d
1	66% e, f	33% a

Fig. 3. ATC (middle) fails to correctly fit the actual confusion matrix (left) in a scenario in which the model threshold is set to a value different from 0.5 (to control the trade-off between recall and precision). In the same scenario, CB-ATC (right) can correctly fit the actual confusion matrix by: i) fitting the error rate of each class via a different threshold; ii) employing a Modified Negative Entropy score function, which ensures that P(f(x)=1) monotonically decreases as it approaches the ML model's threshold (i.e., as a prediction's uncertainty increases).

computing a threshold per predicted class, as illustrated in Figure 2 (right). This is set to match the error rate for the prediction of that specific class. More formally, in CB-ATC each class threshold, denoted as t_i, where $i \in \mathcal{Y}$ and $\mathcal{Y}$ is the set of output classes, is computed as follows:

$$\mathbb{E}_{x \sim \mathcal{D}^{val}} \left[\mathbb{I} \left[pred(f(x)) = i \wedge s(f(x)) < t_i \right] \right] = \mathbb{E}_{(x,y) \sim \mathcal{D}^{val}} \left[\mathbb{I} \left[pred(f(x)) = i \wedge y \neq i \right] \right], \quad (3)$$

where $pred(f(x))$ denotes the class predicted by classifier f for input x. The use of a per class threshold provides CB-ATC with additional flexibility with respect to ATC and, as illustrated in Figure 2, it allows CB-ATC to fit precisely the classifier's confusion matrix.

Limitation 2. The second limitation that is inherent to ATC's design is related to the fact that in many real world applications (including ML-based financial fraud detection systems), the predicted class does not necessarily coincide with the one having higher probability according to the classifier. Focusing on the binary classification case, this means that the threshold used to decide the class to which a prediction belongs (based on the model's output probabilities and referred to as "ML model threshold" in Figures 2 and 3) is often not 0.5. Indeed, this is precisely the case for our target study, where, after training classifier f, the model threshold is configured to ensure that classifier f achieves the SLA-defined false positive rate. This scenario is illustrated in Figure 3, which considers the case in which the ML model's threshold is set by design above 0.5.

This example demonstrates that ATC fails to correctly fit the confusion matrix and we argue that this is due to two main causes: **i)** the use of a single threshold to fit the global error rate rather than a per class error rate (as already discussed); **ii)** the use of a scoring function (like negative entropy), which is symmetric around 0.5, fails to capture a key desirable property whenever the model's threshold is not 0.5: as the model predicted probability gets closer to the model's threshold (from any given direction, i.e., above or below in Figures 2 and 3), the scoring function should also decrease (uncertainty grows as we approach the model's threshold). In order to tackle the latter issue, CB-ATC uses a modified version of the Negative Entropy function, which we call Modified Negative Entropy (MNE) and define as follows:

$$MNE(f(x), pred(f(x))) = \begin{cases} NE(f(x)), & \text{if } pred(f(x)) = \text{argmax}_{j\in\mathcal{Y}}(f_j(x)) \\ -NE(f(x)) - 2, & \text{otherwise} \end{cases} \quad (4)$$

where $NE()$ stands for the Negative Entropy function. Analyzing Figure 3, it is possible to see that $MNE()$ ensures, by design, that the two misclassified samples for which class 0 was predicted get a lower score than any other predicted class-0 sample in the validation set (samples e and f are the closest to the ML model's threshold and correspond, as such, to more uncertain predictions).

5 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:

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

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 [13, 44].

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. However, we make no assumptions regarding label availability, i.e., ground truth labels for transactions can have delays and hence be available only d time units after the transaction has been processed. Section 4.5 details how we address this challenge.

Periodically, the self-adaptation manager can decide to either do nothing (*NOP*), i.e., not to adapt the model, or to *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 in the literature [11, 66]. 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 4.4) 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. System utility (*sysU*) is thus defined as:

$$\begin{aligned} sysU &= \text{total cost} = \\ &= \textsc{cost}(\text{TPR SLA violation}) + \textsc{cost}(\text{FPR SLA violation}) + \textsc{cost}(\text{adaptation tactic}), \end{aligned} \tag{5}$$

where total_cost is the global cost the system is expected to incur in the next time interval; cost(TPR SLA violation) and cost(FPR SLA violation) are the costs of violating the recall and FPR SLAs, respectively, established for the system. The system is charged either of these costs when the monitored recall is below the pre-defined recall threshold and/or when the FPR is above the FPR threshold. Finally, cost(adaptation tactic) encodes the cost of executing a given adaptation tactic. This cost is set to zero when the adaptation tactic selected is *NOP*. System utility could be expressed as a maximization problem by subtracting the costs (and possibly adding rewards). We had to formulate it as an equivalent minization problem only because PRISM does not currently support negative rewards. Finally, although the definition of system utility is application-dependent, it is expected that the cost (monetary and/or latency) of adaptation should often need to be accounted for, hence making this term of the equation general to other applications.

Next, we describe the formal model of the system, illustrating some of its components resorting to PRISM syntax, and the process of creating the adaptation impact predictors (AIPs).

5.1 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 (NOP and retrain) 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* state and can select to execute one tactic among the available ones[2]. 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 (line 9). Finally, since our approach assumes that time is divided into

[2] In PRISM commands are encoded as probabilistic state transitions following the format *[action] guard* $\longrightarrow$ *prob*$_1$*:update*$_1$ + ... + *prob*$_n$*:update*$_n$, where *guard* is a predicate over all variables in the model (including variables from other modules). When *guard* is true, *update*$_1$ is applied with probability *prob*$_1$ (called transition probability). Transition probabilities of a command must sum to 1. *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.

Feature group	Basic Features	Output Features
Feature name	**BF1.1:** $\lvert\mathcal{N}_j\rvert$, # transactions never used for training **BF1.2:** $\lvert\mathcal{I}_i\rvert$, # transactions used for training **BF1.3:** total data = $\lvert\mathcal{I}_i\rvert + \lvert\mathcal{N}_j\rvert$ **BF1.4:** ratio new-old data = $\lvert\mathcal{N}_j\rvert / \lvert\mathcal{I}_i\rvert$ **BF2.1:** current TPR **BF2.2:** current TNR **BF3.1:** time elapsed since the last retrain **BF4.1:** current fraud rate	scores-JSD

Table 1. Features used by the AIPs to predict the benefits of retraining and not adapting.

fixed-sized intervals, we further model a clock whose purpose is to keep track of the passing of each time interval. The clock module is implemented as in [48].

Synthesizing optimal adaptation policies. As can be seen in Listing 1, each *tick* of the clock triggers the accrual of a reward[3]. For this specific use-case, the rewards (lines 18-22) consist of the total costs incurred by the system during that time period due to the tactics executed and possible SLA violations. We account for the latency of executing the adaptation tactic by considering that there is a percentage of transactions that is classified resorting to the previous, non-adapted model, while the remaining transactions are classified resorting to the adapted model. Specifically, the tactic is assumed to take a given percentage of the time interval to execute. This percentage is given by *tacticLatency*. During this period, the system is receiving and classifying transactions. The percentage of transactions classified during this period is encoded in *percentTxs*. Thus, if the non-adapted model violates any threshold, the cost the system incurs is proportional to *tacticLatency* × *percentTxs* × *sum of costs of SLA violations*. For the remainder of the time instant, i.e., *1 - tacticLatency*, the remaining transactions *1 - percentTxs* are classified with the adapted model, hence leading the system to possibly incur a different cost due to variations in the system's TPR and FPR.

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 (Equation (5)) and since 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 PCTL (reward-based property specification logic, c.f. Section 3.1) as $\mathbf{R}^{\text{systemUtility}}_{\text{min=?}}[\mathbf{F}\ \text{`END'}]$, which means "*minimum system utility when time 'end' is reached*", and where **R** and **F** are the operators described in Section 3.1, min=? is the QUERY, and systemUtility specifies the reward structure to use as target. '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 a broader set of 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 achieved by adding these tactics to Listing 1, in addition to *nop* and *retrain*.

[3] The basic building blocks of PRISM's syntax are modules and rewards structures, and formulas. Each module is composed of a set of variables and commands, which affect the variables belonging to the module. The state of the MDP is given by the composition of all variables of all modules. The actions and transitions that the MDP can execute and take at a particular state are given by the commands that are enabled at a specific moment in time, by the different variables. Finally, the rewards that the MDP collects are specified with the rewards structure.

Listing 1. Adaptation manager and System rewards[4]

```
module adaptation_manager
    selectTactic : bool init false;
    currTactic : [none .. retrain] init none;

    [newEvent] !selectTactic -> (selectTactic'=true)&(currTactic'=none);

    // non-deterministic choice between adaptation tactics
    [nop] (selectTactic=true) -> 1:(currTactic'=nop)&(selectTactic'=false);
    [retrain] (selectTactic=true)&(newData > 0) -> 1:(currTactic'=retrain)&(selectTactic'=false);

    [tick] (currTactic != none) -> 1:(currTactic' = none);
endmodule

formula tactic_cost = (currTactic = retrain) ? retrainCost : 0;
formula fpr_violation_cost = (fpr > FPR_THRESHOLD) ? FPR_COST : 0;
formula tpr_violation_cost = (tpr < TPR_THRESHOLD) ? TPR_COST : 0;

rewards "systemUtility"
    [tick] true & (time>0) : (tacticLatency * percentTxs * (
        ((INIT_FPR > FPR_THRESHOLD) ? FPR_COST : 0) + ((INIT_TPR < TPR_THRESHOLD) ? TPR_COST : 0)
    ) + (1 - tacticLatency) * (1 - percentTxs) * (tacticCost + fpr_violation_cost + tpr_violation_cost));
endrewards
```

5.2 AIPs for the Fraud Detection System

As discussed in Section 4.4, the framework instantiates an AIP for each adaptation tactic, trained using the features presented in Table 1. 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 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.

6 EVALUATION

We use the self-adaptive credit card fraud detection system described in Section 5 to evaluate the performance of our framework. Namely, we address 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?
- **RQ5** What is the impact of label delay when estimating model performance?

Experimental Settings. We leverage Kaggle's IEEE-CIS Fraud Detection data-set [1] and the winning solution of the challenge [2] as basis for our implementation. This approach relies on an XGBBoost model [18] that exploits 216 features, including both features originally present in the IEEE-CIS Fraud Detection data-set as well as additionally engineered features. We utilize this 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 data-set are then implemented on top of that base solution. Further, for the purpose of our use-case we leverage only the train data-set of the Kaggle competition for which labels are available (the test data-set does not have labels of the transactions). The presence of labels is required to assess the performance of the system and the benefits of retraining.

Also, we always ensure the transactions of the data-set 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 data-set 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 we vary the retrain cost. This approach is justified as the costs/benefits of adaptation are, in practice, determined by the relative values of these costs, rather than by their absolute values. Thus, by fixing the SLA violation cost and varying the retrain cost we can conduct a sensitivity analysis to evaluate the effectiveness of the proposed framework in a broad range of scenarios (including different retrain latencies).

In this study, the AIPs are random-forest predictors of the sklearn package [52] 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:

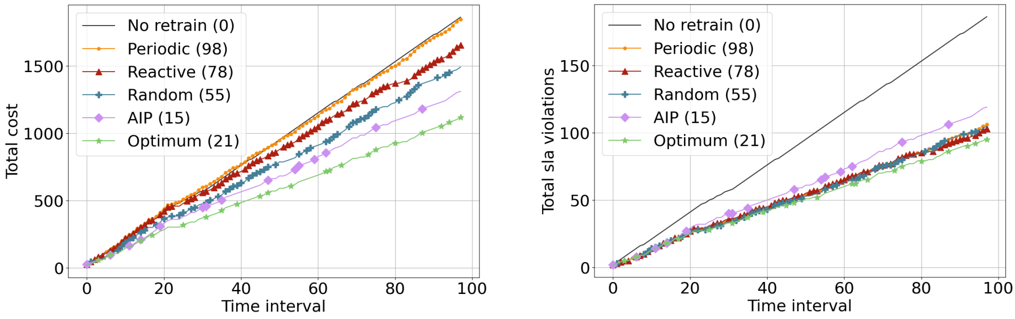
- **No-retrain:** the fraud detection model is trained once, at the beginning of the testing period;
- **Periodic:** the model is retrained at every time interval;
- **Reactive:** the fraud detection model is retrained whenever there is an SLA violation;
- **Random:** at each time step, there is a 50-50 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.

When studying the impact of label delay, we assume the following variants of our framework:

- **AIP:** base implementation of the framework which assumes that labels are immediately available (zero delay);
- **AIP_del:** version that estimates current model performance based on the delayed metrics;
- **AIP_atc:** version that leverages ATC to estimate model performance;
- **AIP_cbatc:** version that leverages CB-ATC to estimate model performance;

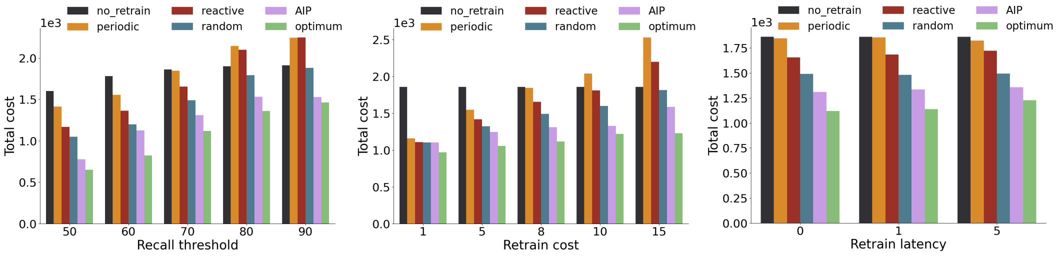
6.1 Utility Improvement due to Retrain

Figure 4 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% [3] and FPR $\leq$ 1% [68], fix the retrain latency to 0 and the retrain cost to 8. These SLA threshold values were set based on values typically employed by related works in this domain [3, 68]. As Figure 4a 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 4b, 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) Cumulative cost incurred by each baseline. (b) Cumulative SLA Violations incurred by each baseline.

Fig. 4. 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.



(a) Variation of Recall SLA (b) Variation of retrain cost (c) Variation of retrain latency

Fig. 5. 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 2. Values tested for different execution contexts.

6.2 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 2 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 5 displays these results.

Regarding the recall threshold (Figure 5a) 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

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

Table 3. 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 (PCC). *NOP* represents the AIPs that estimate the future TPR and TNR when the model is not retrained.

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 5b) 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 5c) 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.

6.3 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 4.4); M – medium set, which includes the basic features and output characteristics; L – encompasses the features of the previous sets and the input characteristics. Table 3 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. 5, good enough to allow implementing effective adaptation strategies. On the other hand, the absolute accuracy metrics reported in Table 3 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 [24]).

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

Table 4. 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.

6.4 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 4 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 6.1 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.

6.5 Impact of Label Delay

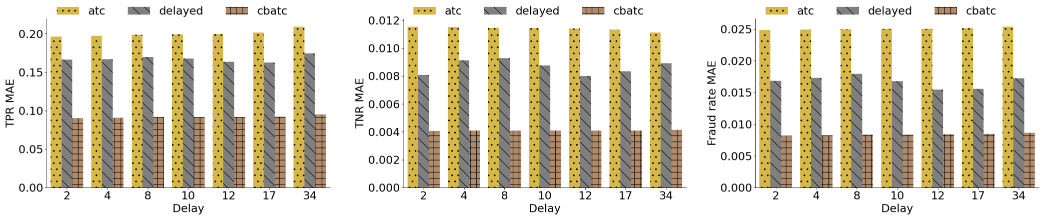
We now evaluate the impact of label delay when estimating model performance and how the accrued estimation error affects the performance. We structure this study in two parts.

We start by analyzing the effectiveness of ATC and CB-ATC (see Section 4.5) in predicting the current confusion matrix of the fraud-detection classifier, as well as the fraud-rate. Recall that this information constitutes some of the input features to the AIP predictors, which are then used to predict the expected variation of the confusion matrix in the following time window, depending on whether the model is retrained or not.

Next, we evaluate to what extent using ATC and CB-ATC to predict the input features for the AIPs impacts the effectiveness of the self-adaptative framework to optimize system utility.

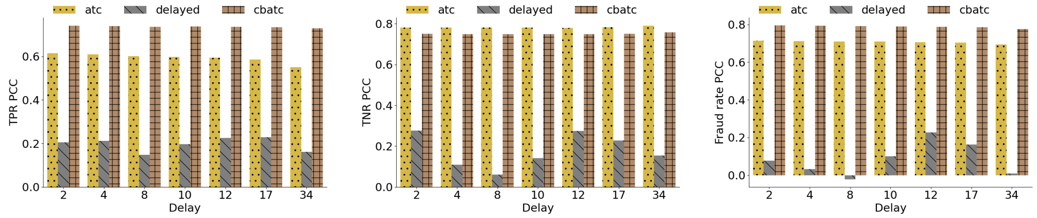
Estimation of Confusion Matrix and Fraud Rate. Figures 6 and 7 evaluate the performance of both ATC and CB-ATC when used to estimate the current (i.e., before adaptation is enacted) TPR, TNR and fraud rate of the fraud detection model. Specifically, we report the mean absolute error (MAE) (Figure 6) and the Pearson correlation coefficient (PCC) (Figure 7) for each approach, and for each value of label delay tested. Regarding the mean absolute error, we observe that CB-ATC substantially reduces the estimation error for all metrics (TPR – Figure 6a; TNR – Figure 6b; fraud rate – Figure 6c) when compared against the delayed labels baseline. ATC however is actually outperformed by the delayed labels baseline. This is due to the fact that ATC was developed to estimate the accuracy of the classifier and not the performance of individual classes, which is the current case. Regardless, and particularly for the TNR and the fraud rate (Figures 6b and 6c) all approaches have a low estimation error.

In terms of the Pearson correlation coefficient (Figure 7) both ATC and CB-ATC present much higher correlations than the delayed approach, for all metrics evaluated. In this setting, high correlation means that the estimations obtained by both ATC and CB-ATC provide meaningful insights into the actual real values of the metrics. Since among the three approaches CB-ATC is the



(a) True Positive Rate (TPR) MAE (b) True Negative Rate (TNR) MAE (c) Fraud rate MAE

Fig. 6. Mean Average Error (MAE) of the delayed, ATC and CB-ATC baselines.



(a) True Positive Rate PCC (b) True Negative Rate PCC (c) Fraud rate PCC

Fig. 7. Pearson Correlation Coefficient (PCC) of the delayed, ATC and CB-ATC baselines.

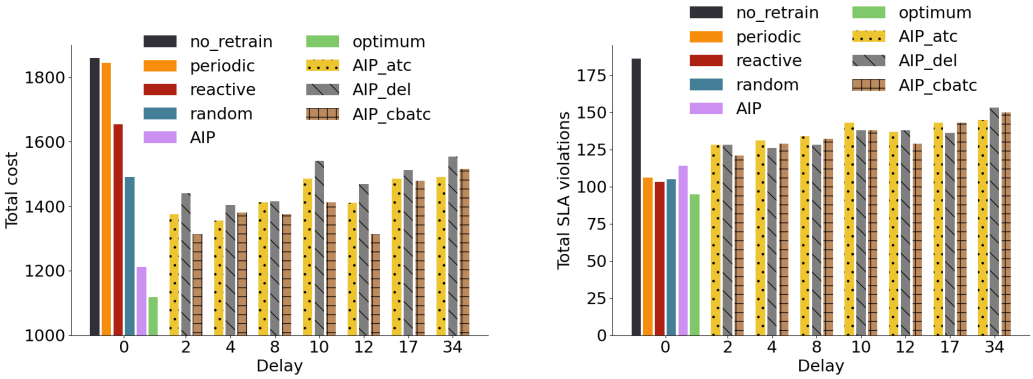
one that overall presents the best trade-off between the error (low) and the correlation (high), we expect it to yield better performance when leveraged by the framework.

These experimental results corroborate our hypothesis on ATC's limitations (Section 4.5.1) and that CB-ATC can effectively solve these issues, leading to lower estimation error.

Impact on system utility. Next we evaluate the impact that using these three approaches has on system utility (i.e., total cost) when dealing with different values of delay. Specifically, we consider delay intervals ranging from 2 to 34, which correspond to, approximately, 1 and 14 days, respectively (recall that a delay interval corresponds to 10 hours). Figure 8 reports the total cost incurred by each baseline for the same environmental settings of Figure 4. These results confirm the superiority of CB-ATC, which globally appears to be the most competitive solution across the considered delay values up to delay 12 (corresponding to 5 days). More precisely, the average of the total cost in the range of delay values [2, 12] for the approaches CB-ATC (AIP_cbatc), ATC (AIP_atc) and delayed metrics (AIP_del) achieve an average of 1359, 1407 and 1453, respectively. This was expected given the conclusions drawn by the analysis of Figures 6 and 7.

Further, these results show that, for relatively small delay values (i.e., delay 2, corresponding to approximately 1 day), the use of the CB-ATC method allows to achieve a system utility that is close to (i.e., 8% worse than) the one obtained in a setting where labels are immediately available (i.e., the AIP baseline).As the delay grows beyond 12 (corresponding to 5 days), the performances of CB-ATC, as well as ATC, tend to degrade relatively to the baseline that has only access to delayed information, leading them to achieve a performance that is on par with a random approach. Conversely, in the [2, 12] interval of delays, CB-ATC achieves an average gain of approx. 10% with respect to a random approach.

We suspect that the root cause of the problem is that the quality of the AIP models degrades significantly as the delay grows, independently of whether the AIPs' input features are being predicted (via ATC or CB-ATC) or if delayed values are being used. Regardless, and as previously



(a) Total cost incurred by each baseline at the end of the simulation and for each delay value.

(b) Total SLA violations incurred by each baseline at the end of the simulation and for each delay value.

Fig. 8. Total cost and SLA violations incurred by the different approaches when accounting for label delay. The values reported correspond to the last instant of the simulation, corresponding to time=98 in Figure 4.

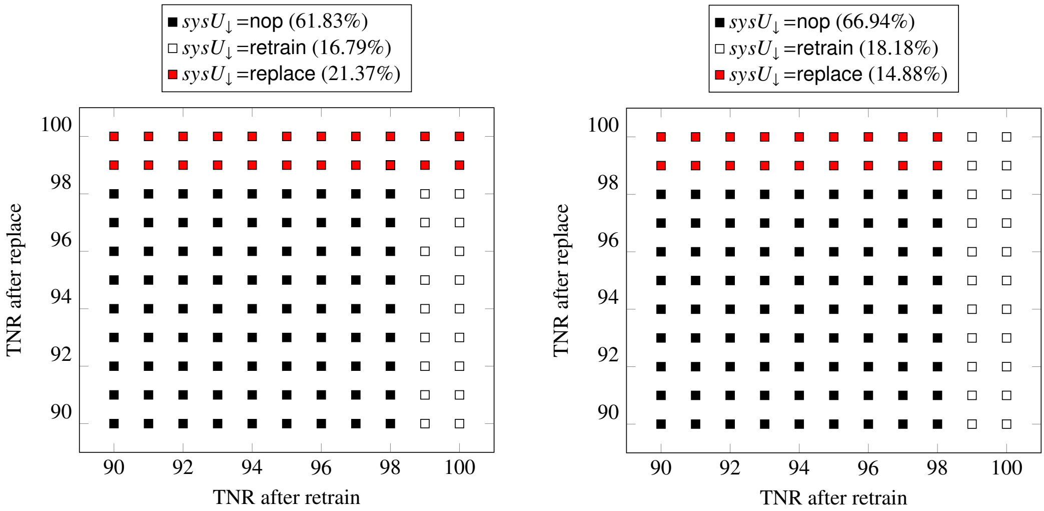
observed, CB-ATC can predict these features more accurately than an oracle that simply outputs delayed information. We argue that this limitation might be imputed to the modeling approach that we currently use to construct the AIPs, which we intend to address in future work by investigating alternative ML modeling techniques and different feature engineering methods.

6.6 Using Additional Tactics

In this work we have created AIPs only for the *nop* and *retrain* tactics. However, even in absence of AIPs capable of estimating the effects of additional tactics, the planning component (i.e., the probabilistic model checker) of our framework can still be effectively used to support “what-if” analysis aimed at identifying in which scenarios the use of additional adaptation tactics would optimize system utility.

In order to demonstrate these capabilities, we consider the availability of a third tactic, which we refer to as *component replacement*, that consists of replacing the ML model used for fraud detection with a rule-based model defined by human experts. We assume that the rule-based model offers a fixed TPR of 75% and that the current performance of the ML model is TPR = 75% and TNR = 98%. The SLA thresholds and costs are the ones previously defined. We then use the planning component of our framework to conduct a what-if analysis that aims to identify the optimal adaptation tactic when varying: (i) the TNR for the rule-based model, (ii) the TNR for the ML model after it is retrained, and (iii) the costs for the retrain and component replacement tactics.

Figure 9 shows the results of this analysis, indicating with different colours the optimal tactic for each setting of rule-based model TNR, retrained ML model TNR, and tactic costs. The figure reports on the X axis the TNR of the ML-based component after retrain and on the Y axis the TNR of the rule-based model. However, Figure 9a and Figure 9b consider different execution costs for each tactic. This study demonstrates that the model checker is capable of selecting between multiple adaptation tactics by reasoning on the impact of each alternative tactic on system utility and identifying the one that yields the maximum gains.



(a) Replace cost = 0 and retrain cost = 8

(b) Replace cost = 8 and retrain cost = 5

Fig. 9. Optimal adaptation tactic selection as a function of the True Negative Rate (TNR) offered by the component replacement and retrain adaptation tactics. Recall that the TNR offers a direct proxy to the FPR SLA since $FPR = 1 - TNR$. The figures demonstrate that the model checker is capable of selecting between multiple adaptation tactics by analyzing the expected cost-benefits offered by each and selecting the one that optimizes system utility.

6.7 Threats to Validity, Limitations, and Discussion

The findings regarding the predictability of the impacts of retrain are data-set and domain- dependent and so they cannot be generalized to other domains or data-sets (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 (e.g., self-adaptive intrusion detection, spam detector, or machine translation systems) and for systems that rely on other types of ML models (e.g., neural-networks, support vector machines, linear models). Analogous considerations apply to the evaluation of the CB-ATC method, which has only been conducted in the context of the case study considered in this work. The proposed method should be evaluated on a broader set of data-sets in order to verify whether the benefits observed in our study generalize to different domains.

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). Specifically, for the considered use-case and evaluation, retrain latency was assumed to be lower than one time interval. This assumption holds for the considered use-case since the ML components employed by the fraud detection system are relatively simple and hence it is reasonable to assume that their training takes less than 10 hours. However, for more complex ML components, such as large language models (LLMs) [47] this assumption might not hold. A simple approach to circumvent this issue would consist in setting the time interval used by the model checking tool to be at least as large as the retrain latency, while preserving the frequency with which the framework analyzes the need for adaptation (i.e., the frequency with which we query the framework). On the one hand, this would avoid the need to estimate the ML model's quality over multiple time steps in the future, which would be necessary if the time interval was set to be smaller than the retrain latency. On the other hand, it would still preserve the framework's reactivity to possible changes in the environment, as this is dictated by the frequency with which we query the framework.

Further, although in this article we have constructed and evaluated AIPs to predict the impact of executing only two tactics (nop and retrain), our framework has been designed to support other tactics, such as the ones identified in our prior work [14]. Additionally, this work constitutes a first step towards enabling long term planning of ML adaptations, which the probabilistic model checker intrinsically supports. In fact, the current challenge hindering this extension lies in the computational complexity of creating the AID when accounting for longer horizons (i.e., larger than one time interval). This same challenge also impacts the extension to a setting with multiple adaptation tactics, due to the need to account for possible dependencies among tactics (i.e., tactics whose outcome depends on whether a different tactic was executed, e.g., retrain and fine-tune). Nonetheless, the methodology presented in this work established the required building blocks for coping with more than two tactics (also discussed in Section 6.6) and long term planning extensions.

Finally, the several building blocks that constitute the framework are what makes it general. Specifically, to instantiate the framework for additional applications, it is necessary to **(i)** create an AID by following the proposed methodology (Section 4.4), **(ii)** instantiate AIPs for the available adaptation tactics and metrics of interest,[5] leveraging the AID that was created, **(iii)** create a formal model of the system, capturing the error of the ML component as described in Section 4.3. Further, the proposed strategy to formally model ML components does not constrain the applicability of the framework to specific types of ML models. In fact, classification models can be evaluated through confusion matrices independently of the underlying ML algorithms they employ (e.g., neural networks, random forests).

7 CONCLUSIONS AND FUTURE WORK

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 adapting 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 trade-offs of retraining ML components allows for better adaptation decisions when compared against model-free baselines.

We further evaluated the impact of delays in the availability of labeled data, required to quantify the current model's performance, on the effectiveness of the proposed framework. Thus, we integrated in our framework a state-of-the-art technique for model's quality estimation (ATC) and identified relevant shortcomings that arised when employing it in our use case. This led us to propose a novel variant, named CB-ATC, which we empirically showed to provide more accurate estimates of model's quality (e.g., its normalized confusion matrix) than ATC. We finally demonstrated that, by employing CB-ATC, it is possible to approximate the ideal scenario of immediate label availability in the presence of delays of up to 1 day, in the considered use case. For larger delays, the effectiveness of the self-adaptive framework degrades and the use of model quality estimation methods such as CB-ATC does not seem to provide substantial gains over simpler approaches that leverage delayed labels.

This work opens a number of avenues for future work, namely: **(i)** validate the framework in a broader range of domains; **(ii)** study the impact of resorting to different model types and techniques to instantiate the AIPs; **(iii)** investigate the use of alternative feature engineering approaches to develop more accurate AIPs; **(iv)** increase the repertoire of adaptation tactics employed by the

[5] For the considered use-case these metrics were *recall* and *false positive rate* (FPR) and the tactics were *nop* and *retrain.*

framework and demonstrate the trade-offs of each adaptation tactic in the presence of different environmental drifts (e.g., different types of data-set shifts [56], changes in the costs of computational resources or workload characteristics); **(v)** extend the framework to plan for the long-term (e.g., by considering adaptation tactics whose execution latency span over multiple time intervals).

REFERENCES

[1] 2019. *IEEE-CIS Fraud Detection.* https://www.kaggle.com/competitions/ieee-fraud-detection/overview

[2] 2019. *IEEE-CIS Fraud Detection Winner Solution.* https://www.kaggle.com/code/cdeotte/xgb-fraud-with-magic-0-9600

[3] David Aparício et al. 2020. ARMS: Automated rules management system for fraud detection. *arXiv preprint arXiv:2002.06075* (2020).

[4] Christopher Bishop and Nasser Nasrabadi. 2006. *Pattern recognition and machine learning.* Vol. 4. Springer.

[5] T. Bureš. 2021. Self-Adaptation 2.0. In *Procs. of SEAMS* (Madrid, Spain).

[6] Radu Calinescu et al. 2017. Synthesis and verification of self-aware computing systems. In *Self-Aware Computing Systems.* Springer.

[7] Radu Calinescu, Raffaela Mirandola, Diego Perez-Palacin, and Danny Weyns. 2020. Understanding uncertainty in self-adaptive systems. In *Procs. of ACSOS.*

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

[9] Yinzhi Cao et al. 2018. Efficient repair of polluted machine learning systems via causal unlearning. In *Procs. of Asia CCS.*

[10] Yinzhi Cao and Junfeng Yang. 2015. Towards making systems forget with machine unlearning. In *Procs. of S&P.* IEEE.

[11] Maria Casimiro, Diego Didona, Paolo Romano, Luis Rodrigues, Willy Zwaenepoel, and David Garlan. 2020. Lynceus: Cost-efficient Tuning and Provisioning of Data Analytic Jobs. In *Procs. of ICDCS* (virtual event).

[12] M. Casimiro, D. Garlan, J. Cámara, L. Rodrigues, and P. Romano. 2021. A Probabilistic Model Checking Approach to Self-Adapting Machine Learning Systems. In *Procs. of ASYDE, Co-located with SEFM 2021.*

[13] Maria Casimiro, Paolo Romano, David Garlan, Gabriel Moreno, Eunsuk Kang, and Mark Klein. 2021. Self-Adaptation for Machine Learning Based Systems.. In *ECSA (Companion).*

[14] Maria Casimiro, Paolo Romano, David Garlan, Gabriel A Moreno, Eunsuk Kang, and Mark Klein. 2022. Self-adaptive Machine Learning Systems: Research Challenges and Opportunities. In *Software Architecture: 15th European Conference, ECSA 2021 Tracks and Workshops; Växjö, Sweden, September 13–17, 2021, Revised Selected Papers.* Springer, 133–155.

[15] Maria Casimiro, Paolo Romano, David Garlan, and Luís Rodrigues. 2022. Towards a Framework for Adapting Machine Learning Components. In *2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS).* 131–140. https://doi.org/10.1109/ACSOS55765.2022.00031

[16] Jiefeng Chen, Frederick Liu, Besim Avci, Xi Wu, Yingyu Liang, and Somesh Jha. 2021. Detecting errors and estimating accuracy on unlabeled data with self-training ensembles. *Advances in Neural Information Processing Systems* 34 (2021), 14980–14992.

[17] Tao Chen. 2019. All Versus One: An Empirical Comparison on Retrained and Incremental Machine Learning for Modeling Performance of Adaptable Software. In *Procs. of SEAMS.*

[18] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining.* 785–794.

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

[20] Jürgen Cito, Isil Dillig, Seohyun Kim, Vijayaraghavan Murali, and Satish Chandra. 2021. Explaining mispredictions of machine learning models using rule induction. In *Procs. of ESEC/FSE* (virtual event).

[21] Edmund M. Clarke, E Allen Emerson, and A Prasad Sistla. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 8, 2 (1986), 244–263.

[22] Rogério de Lemos et al. 2013. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II.* Springer.

[23] José GC de Souza, Ricardo Rei, Ana C Farinha, Helena Moniz, and André FT Martins. 2022. QUARTZ: Quality-Aware Machine Translation. In *Proceedings of the 23rd Annual Conference of the European Association for Machine Translation.* 315–316.

[24] Diego Didona, Francesco Quaglia, Paolo Romano, and Ennio Torre. 2015. Enhancing Performance Prediction Robustness by Combining Analytical Modeling and Machine Learning. In *Procs. of ACM/SPEC ICPE.*

[25] B. Erickson, P. Korfiatis, Z. Akkus, and T. Kline. 2017. Machine learning for medical imaging. *Radiographics* 37, 2 (2017).

[26] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. 2014. A survey on concept drift adaptation. *ACM CSUR* 46, 4 (2014).

[27] Saurabh Garg, Sivaraman Balakrishnan, Zachary C Lipton, Behnam Neyshabur, and Hanie Sedghi. 2022. Leveraging unlabeled data to predict out-of-distribution performance. *arXiv preprint arXiv:2201.04234* (2022).

[28] O. Gheibi and D. Weyns. 2022. Lifelong Self-Adaptation: Self-Adaptation Meets Lifelong Machine Learning. In *Procs. of SEAMS* (virtual event).

[29] Omid Gheibi, Danny Weyns, and Federico Quin. 2021. Applying machine learning in self-adaptive systems: A systematic literature review. *ACM TAAS* 15, 3 (2021).

[30] Devin Guillory, Vaishaal Shankar, Sayna Ebrahimi, Trevor Darrell, and Ludwig Schmidt. 2021. Predicting with confidence on unseen distributions. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 1134–1144.

[31] Hans Hansson and Bengt Jonsson. 1994. A logic for reasoning about time and reliability. *Formal aspects of computing* 6 (1994), 512–535.

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

[33] Sara Hezavehi, Danny Weyns, Paris Avgeriou, Radu Calinescu, Raffaela Mirandola, and Diego Perez-Palacin. 2021. Uncertainty in self-adaptive systems: A research community perspective. *ACM TAAS* 15, 4 (2021).

[34] Pooyan Jamshidi et al. 2017. Transfer learning for improving model predictions in highly configurable software. In *Procs. of SEAMS.*

[35] Yiding Jiang, Vaishnavh Nagarajan, Christina Baek, and J Zico Kolter. 2021. Assessing generalization of sgd via disagreement. *arXiv preprint arXiv:2106.13799* (2021).

[36] D. Keefer. 1994. Certainty equivalents for three-point discrete-distribution approximations. *Management science* 40, 6 (1994).

[37] J. Kephart and D. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (2003).

[38] B. Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad Al Sallab, Senthil Yogamani, and Patrick Pérez. 2021. Deep reinforcement learning for autonomous driving: A survey. *IEEE T-ITS* (2021).

[39] P. Kourouklidis, D. Kolovos, J. Noppen, and N. Matragkas. 2021. A Model-Driven Engineering Approach for Monitoring Machine Learning Models. In *Procs. of MODELS-C*. IEEE.

[40] M. Kwiatkowska, G. Norman, and D. Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *Procs. of CAV'11* (Snowbird (UT), USA) *(LNCS, Vol. 6806)*. Springer.

[41] M. Kwiatkowska, G. Norman, and D. Parker. 2022. Probabilistic model checking and autonomy. *Annual Review of Control, Robotics, and Autonomous Systems* 5 (2022).

[42] Michael Langford, Kenneth Chan, Jonathon Fleck, Philip McKinley, and Betty Cheng. 2021. MoDALAS: Model-Driven Assurance for Learning-Enabled Autonomous Systems. In *Procs. of MODELS.*

[43] Bertrand Lebichot, Gian Paldino, Wissam Siblini, Liyun He-Guelton, Frédéric Oblé, and Gianluca Bontempi. 2021. Incremental learning strategies for credit cards fraud detection. *International Journal of Data Science and Analytics* 12, 2 (2021), 165–174.

[44] Y. Lucas and J. Jurgovsky. 2020. Credit card fraud detection using machine learning: A survey. *CoRR* abs/2010.06479 (2020).

[45] Frank J Massey Jr. 1951. The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American statistical Association* 46, 253 (1951), 68–78.

[46] ML Menéndez, JA Pardo, L Pardo, and MC Pardo. 1997. The jensen-shannon divergence. *Journal of the Franklin Institute* 334, 2 (1997).

[47] Tomáš Mikolov, Anoop Deoras, Daniel Povey, Lukáš Burget, and Jan Černockỳ. 2011. Strategies for training large scale neural network language models. In *2011 IEEE Workshop on Automatic Speech Recognition & Understanding*. IEEE, 196–201.

[48] Gabriel Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. 2015. Proactive Self-Adaptation under Uncertainty: A Probabilistic Model Checking Approach. In *Procs. of ESEC/FSE* (Bergamo, Italy).

[49] G. Moreno, J. Cámara, D. Garlan, and M. Klein. 2018. Uncertainty reduction in self-adaptive systems. In *Procs. of SEAMS* (Gothemburg, Sweden).

[50] Yaniv Ovadia, Emily Fertig, Jie Ren, Zachary Nado, David Sculley, Sebastian Nowozin, Joshua Dillon, Balaji Lakshminarayanan, and Jasper Snoek. 2019. Can you trust your model's uncertainty? Evaluating predictive uncertainty under dataset shift. In *Procs. of NIPS* (Vancouver, Canada).

[51] Sinno Jialin Pan and Qiang Yang. 2009. A survey on transfer learning. *IEEE TKDE* 22, 10 (2009).

[52] Fabian Pedregosa et al. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[53] J. Perdomo, T. Zrnic, C. Mendler-Dünner, and M. Hardt. 2020. Performative Prediction. In *Procs. of ICML.*

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

[55] Martin L Puterman. 2014. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons.

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

[57] A. Rabanser, S. Günneman, and Z. Lipton. 2019. Failing Loudly: An Empirical Study of Methods for Detecting Dataset Shift. In *Procs. of NIPS* (Vancouver, Canada).

[58] T. Saputri and S.-W. Lee. 2020. The Application of Machine Learning in Self-Adaptive Systems: A Systematic Literature Review. *IEEE Access* 8 (2020).

[59] Felix Stahlberg. 2020. Neural machine translation: A review. *Journal of Artificial Intelligence Research* 69 (2020), 343–418.

[60] Georgios Symeonidis, Evangelos Nerantzis, Apostolos Kazakis, and George A Papakostas. 2022. Mlops-definitions, tools and challenges. In *2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE, 0453–0460.

[61] J. Townsend. 1971. Theoretical analysis of an alphabetic confusion matrix. *Perception & Psychophysics* 9, 1 (1971).

[62] Zijie J Wang, Dongjin Choi, Shenyu Xu, and Diyi Yang. 2021. Putting humans in the natural language processing loop: A survey. *arXiv preprint arXiv:2103.04044* (2021).

[63] Danny Weyns, Bradley Schmerl, Masako Kishida, Alberto Leva, Marin Litoiu, Necmiye Ozay, Colin Paterson, and Kenji Tei. 2021. Towards Better Adaptive Systems by Combining MAPE, Control Theory, and Machine Learning. In *Procs. of SEAMS*.

[64] Yinjun Wu, Edgar Dobriban, and Susan Davidson. 2020. DeltaGrad: Rapid retraining of machine learning models. In *Procs. of ICML*.

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

[66] N. Yadwadkar, B. Hariharan, J. Gonzalez, B. Smith, and R. Katz. 2017. Selecting the Best VM Across Multiple Public Clouds: A Data-driven Performance Modeling Approach. In *SoCC* (Santa Clara (CA), USA).

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

[68] Yongchun Zhu, Dongbo Xi, Bowen Song, Fuzhen Zhuang, Shuai Chen, Xi Gu, and Qing He. 2020. Modeling users' behavior sequences with hierarchical explainable network for cross-domain fraud detection. In *Proceedings of The Web Conference 2020*. 928–938.

[69] I. Žliobaitė. 2010. Learning under concept drift: an overview. *arXiv preprint arXiv:1010.4784* (2010).

Received 3 March 2023; revised 12 March 2009; accepted 5 June 2009