Safe Learning and Repairing of Numeric Action Models for Planning

Argaman Aloni Mordoch, Brendan Juba, Roni Stern

Safe Learning and Repairing of Numeric Action Models for Planning

Argaman Aloni-Mordoch[1] and **Brendan Juba**[2] and **Roni Stern**[1]
[1] Ben Gurion University of the Negev
e-mail: mordocha@post.bgu.ac.il, roni.stern@gmail.com
[2] Washington University in St. Louis
e-mail: bjuba@wustl.edu

Abstract

Automated planners often require a model of the acting agent's actions, given in some planning domain description language. Yet obtaining such an action model is a notoriously hard task. This task is even harder in mission-critical domains, in which a trial-and-error approach for learning how to act is not an option. In such domains, the action model used to generate plans must be *safe*, in the sense that plans generated with it must be applicable and achieve their goals. The challenge of learning safe action models for planning has been recently addressed for domains in which states are sufficiently described with Boolean variables. In this work, we go beyond this limitation and propose the Numeric Safe Action Model (*N-SAM*) learning algorithm. *N-SAM* runs in time that is polynomial in the number of observations and, under certain conditions, is guaranteed to return safe action models. Experimental results show that *N-SAM* is able to quickly learn a safe action model that can solve the majority of problems in a given domain. Finally, we describe an application of *N-SAM* to to repair action models that are observed to be incorrect.

1 Introduction

Planning is the fundamental task of choosing which actions to perform to achieve the desired outcome. Automated domain-independent planning is the fundamental challenge of developing an AI capable of solving a wide range of planning problems [1]. Powerful domain-independent planners have been developed for various types of planning problems. These planners often require a model of the acting agent's actions, given in some planning domain description language. Many planning languages have been proposed, such as STRIPS [2], the Planning Domain Definition Language (**PDDL**) [3], PDDL 2.1 [4], and RDDL [5]. Powerful corresponding planners have been developed such as FastForward [6] and FastDownward [7] for problems given in PDDL and COLIN [8], TLP-GP [9], and DiNo [10] for problems given in PDDL2.1.

Yet, defining an agent's action model in these planning languages for real-world problems is a notoriously hard task. This modeling challenge has been acknowledged in the literature, and algorithms for learning agent action models from observations have been proposed [11; 12; 13; 14]. Since the learned model may be different from the domain's actual action model, using it to plan raises two challenges: *safety* and *completeness*. The safety risk is that the learned model may generate a plan that cannot be applied in the domain or may not reach a state that satisfies the problem goals. The completeness risk is that the learned model may be too restrictive to enable generating plans even for planning problem that can be solved with the actual action model. In this work, we focus on safety, which is crucial in mission-critical domains where a trial-and-error approach for learning how to act or online replanning are not options. In such domains, the action model used to generate plans must be *safe*, in the sense that plans generated with it must be applicable and achieve their goals.

The challenge of learning safe action models for planning has been recently addressed by the *SAM* learning family of algorithms [15; 14; 16]. But, these algorithms are inapplicable for numeric planning, where may states include continuous state variables. The same safety considerations have motivated work in offline RL [17; 18; 19], with the major difference that planning models can be re-used for a wide range of different goals, in contrast to standard RL that trains to a specific reward process.

In this work, we explore the problem of learning a safe action model for numeric planning. Specifically, we focus on problems that can be defined in PDDL2.1 [4], a popular language for describing deterministic fully observable numeric planning problems. We prove that learning learning a safe action model is not possible without making some assumptions on the preconditions and effects of the agent's actions. Then, we identify a reasonable set of assumption in which learning a safe action model is possible, namely that action effects and preconditions are linear. Specifically, we introduce the Numeric Safe Action Model (*N-SAM*) learning algorithm, which runs in time that is polynomial in the number of observations and is guaranteed to return safe action models in domains that satisfy our linearity assumptions. The worst-case sample complexity of *N-SAM*, even with our assumptions, is intractable. Practically, experimental results over standard numeric planning benchmarks show that action models created by *N-SAM* were able to solve most relevant problems with fewer than 70 trajectories.

Finally, we describe an application of *N-SAM* for repairing faulty numeric action models. There, we apply *N-SAM* on trajectories created by a faulty action model, showing that the resulting action model is more effective than the original faulty action model.

2 Background

We focus on planning problems in domains where action outcomes are deterministic, states are fully observable, and states are described with both discrete and continuous state variables. Such problems are commonly modeled using the PDDL2.1 [4] language.[1] To define a numeric planning problem in PDDL2.1, we introduce the following notation. A domain is defined by a tuple $D = \langle F, X, A \rangle$ where F is a finite set of Boolean variables, X is a set of numeric variables, and A is a set of actions. A state is an assignment of values to all variables in $F \cup X$. For a state variable $v \in F \cup X$ we denote by $s(v)$ the value assigned to v in state s.

Every action $a \in A$ is defined by a tuple $\langle \textit{name}(a), \textit{pre}(a), \textit{eff}(a) \rangle$ representing to the action's name, preconditions, and effects, respectively. The preconditions of an action a are a set of assignments over (possibly a subset of) the Boolean variables and a set of conditions over (possibly a subset of) the numeric variables. These conditions are of the form (ξ, Rel, k) where ξ is an arithmetic expression over X, $Rel \in \{\leq, <=, >, \geq\}$, and k is a number. The effects of an action a, denoted $\textit{eff}(a)$, are a set of assignments over F and X representing how the state changes after applying a. An assignment over a Boolean variable is either True or False. An assignment over a numeric variable $x \in X$ is a tuple of the form $\langle x, op, \xi \rangle$ where ξ is a numeric expression over X and op is either increase ("+="), decrease ("-="), or assign (":="). The set of actions with their definitions is referred to as the *action model* of the domain. We say that an action a is applicable in a state s if s satisfies $\textit{pre}(a)$. Applying a in s, denoted $a(s)$, results in a state that differs from s only according to the assignments in $\textit{eff}(a)$. A planning problem is defined by a tuple $\langle D, s_0, G \rangle$ where D is a domain, s_0 is the initial state, and G is the problem goals. The problem goals G are an assignment of values to a subset of the Boolean variables and a set of conditions over the numeric variables. A solution to a planning problem is a *plan*, i.e., a sequence of actions, that are applicable in s_0 and results in a state s_G in which G is satisfied.

Different algorithms have been proposed for learning planning action models [20; 13; 21; 14]. Some action-model learning algorithms, such as LOCM [11] and its extensions LOCM2 [20], analyze observed plan sequences, where each action appears as an action name and arguments in the form of a vector of object names. Other algorithms, such as FAMA [21], can also utilize information about the states reached while executing plans in the domain. This information is commonly described as a set of trajectories. A *trajectory* is a list of state transitions of the form $\langle s_i, a_i, ai(s_i) \rangle$ created by executing some plan $(\ldots, a_i, \ldots)$ in the domain. For a state transition $\langle s, a, s' \rangle$ the states s and s' are referred to as the *pre-state* and *post-state*, respectively.

Most action model learning algorithms do not provide any guarantee that plans created with the learned action model are applicable in the real action model. The *SAM* learning algorithm [15] address this gap by providing the following guarantee: the learned action model is *safe* in the sense that plans generated with it are guaranteed to be applicable in the real action model and yield the predicted states. *SAM* also runs in polynomial time and number of samples required to guarantee the learned action model is sufficient to solve most problems scales gracefully. However, *SAM* learning and its recent extensions [14; 16] are limited to learning action models that do not support numeric state variables. In fact, learning numeric action models have been scarcely studied. PlanMiner [22] is a notable exception. It is an algorithm that learns numeric action models from partially known and potentially noisy trajectories. But, PlanMiner does not provide any safety guarantees. In addition, its publicly available implementation does not learn numeric preconditions.

[1] Technically, we focus on level 2 of PDDL2.1.

3 Learning Numeric Action Models

We consider the challenge of solving numeric planning problems of the form $\langle D = \langle F, X, A \rangle, s_0, G \rangle$ while assuming the problem solver is not given explicit information about the set of actions A. The problem solver is given a set of *trajectories* $\mathcal{T}$, created by executing plans that solve other planning problems in the domain, and observing the resulting states. Such trajectories may have been created by following instructions of a human operator, random exploration, or some other domain-specific manner. We assume the problem solver has full observability of the states and actions in given trajectories, including knowing the name and parameters of all observed actions. In addition, we assume the plans used to create the given trajectories are valid, in the sense that preconditions and effects of every observed action, are satisfied.

Our approach for solving such problems, referred to as *planning with offline learning*, comprises two steps: (1) *learning* an action model $\hat{A}$ using the given trajectories $\mathcal{T}$, and (2) *planning* using the learned action model $\hat{A}$, i.e., using an off-the-shelf PDDL 2.1 planner to find a plan for the planning problem $\left\langle \left\langle F, X, \hat{A} \right\rangle, s_0, G \right\rangle$. There are many planning algorithms to solve such PDDL2.1 problems, such as Metric FF [23] and ENHSP [24]. Recall that since the learned action model may differ from the actual action model, planning with it raises both safety and completeness risks. The relative importance of each risk is application dependent. In this work, we emphasize addressing the safety risk, which is key in applying our method to mission-critical applications or applications in which plan failure is very costly. To this end, we aim to learn an *safe action model* [15; 14].

Definition 1 (Safe Action Model)**.** *An action model $\hat{A}$ is ϵ-safe w.r.t. a norm $\|\cdot\|$ in a planning domain $D = \langle F, X, A \rangle$ if for every action $\hat{a} \in \hat{\mathcal{A}}$ there exists an action $a \in A$ with the same name such that for every state s: (1) if $\hat{a}$ is applicable in s then so is a, and (2) if $\hat{a}$ is applicable in s then applying it in s results in a state that is ϵ-close to that obtained by applying a to s, i.e., $\|\hat{a}(s) - a(s)\| \leq \epsilon$.*

It is easy to see that plans created with a safe action models are safe, in the sense that they execute as anticipated by the model. We allow for a small ϵ error in our definition because this is unavoidable in most situations, due to numerical issues, limited precision sensors, etc. We refer to an algorithm that learns a safe action model from a given set of trajectories as a *safe action model learner*.

3.1 Negative Results

In our model, the examples consist of trajectories, i.e., plans that have been successfully executed. In particular, this means that in each step of every plan, for the action chosen at that step the preconditions are necessarily satisfied. Thus, if we view the preconditions of an action as a Boolean

concept, we are learning the preconditions only from positive examples. Supervised learning with only positive examples was first considered by [25], and it is known that many classes of representations cannot be learned [25; 26; 27; 28]. Indeed, we can formalize this connection and thus translate these findings into our model, establishing that a many natural families of preconditions cannot be learned without stronger assumptions. [25] observed that learning from positive examples alone entails that the learner's hypothesis never makes a false-positive error – corresponding to our *safety* property for preconditions. Thus, for convenience, we will formalize the model in these terms.

Definition 2. *Let $\mathcal{X}$ denote our set of instances. Let $\mathcal{C}$ and $\mathcal{H}$ be sets of Boolean-valued functions on $\mathcal{X}$. The problem of* PAC learning $\mathcal{C}$ with $\mathcal{H}$ using positive examples *is as follows: we are given parameters $\epsilon, \delta \in (0,1)$, and access to examples from $\mathcal{X}$ sampled from a distribution D supported on $\{x \in \mathcal{X} : c(x) = 1\}$ for some $c \in \mathcal{C}$. With probability $1-\delta$, we must return $h \in \mathcal{H}$ such that*

- No false positives: $h(\mathcal{X}) \subseteq c(\mathcal{X})$
- $1-\epsilon$ accurate: $\Pr_{x \in D}[h(x) = 1] \geq 1 - \epsilon$.

If our algorithm runs in time polynomial in the representation size of members of $\mathcal{X}$, the representation size of c, $1/\epsilon$, and $1/\delta$, then we say that the algorithm is efficient. *If $\mathcal{C} = \mathcal{H}$, this is known as the* proper *variant of the problem, otherwise the learner is said to be* improper.

We now observe that safe action model learners that learn domains with preconditions from $\mathcal{C}$ using preconditions from $\mathcal{H}$ can be used to obtain PAC learning algorithms for learning $\mathcal{C}$ with $\mathcal{H}$ using positive examples:

Proposition 1. *Suppose there exists a safe action model learning algorithm for domains with preconditions from $\mathcal{C}$ that produces preconditions from $\mathcal{H}$, for a $\mathcal{C}$ that for any ith attribute, contains a nontrivial precondition that depends only on that attribute. Suppose furthermore that there is a probability distribution with support equal to $\mathcal{X}$ that can be efficiently sampled. Then there is a PAC learning algorithms for learning $\mathcal{C}$ with $\mathcal{H}$ using positive examples. Moreover, if the safe action model learning algorithm is efficient, then so is the PAC learning algorithm.*

Proof. We reduce PAC learning with positive examples to safe action model learning as follows: We consider a domain with two actions, "pos" and "neg," and with states given by $\mathcal{X}$ extended by two Boolean attributes, s and t. Let s_0 and s_1 be values for s that would, respectively, violate and satisfy some precondition in $c_s \in \mathcal{C}$. We provide the following trajectories to our safe action model learner: with probability $1/2$, we sample x from D, and append it with $s = s_1$ and $t = 0$ to obtain the initial state; and, otherwise, we sample x from our distribution with support $\mathcal{X}$, and append it with $s = s_1$ and $t = 0$ to obtain the initial state. In the first case, the trajectory consists of taking the action pos, followed by a state with x appended with $s = s_0$ and $t = 1$. In the second case, the trajectory consists of taking the action neg, followed by a state with x appended with $s = s_0$ and $t = 1$. In both cases, the goal is "$t = 1$." We obtain an action model from the algorithm run with δ and $\epsilon/2$, and return its precondition for pos as our solution h for PAC learning. h must have no false positives because the precondition for pos must be safe: suppose the true precondition for pos were $c \in \mathcal{C}$; note that this is consistent with the data. Then, if there exists $x \in \mathcal{X}$ such that $h(x) = 1$ but $c(x) = 0$, our action model would permit pos for some x where its precondition is violated, thus violating safety. Similarly, h must be $1-\epsilon$-accurate: our action model learner guarantees that with probability $1-\delta$, it is $1-\epsilon/2$ complete. But if neg has the precondition c_s that is only satisfied by s_1, then only pos can be executed for those examples with $s = s_0$, i.e., the examples sampled from D. Hence, the precondition for pos must be satisfied with probability at least $1-\epsilon$ on D or else the action model would fail with probability greater than $\epsilon/2$. Hence, h is indeed as required for PAC learning. The "moreover" part is immediate from the construction. □

By the contrapositive of Proposition 1 and prior results, we obtain that:

Corollary 1 ([27])**.** *The family of preconditions given by single linear inequalities with at most two variables cannot be safely learned by any $\mathcal{H}$.*

Corollary 2 ([28])**.** *The family of preconditions given by the disjunction of two univariate inequalities cannot be safely learned by any $\mathcal{H}$.*

Note that in particular, therefore, classes $\mathcal{C}$ that contain the above representations as special cases cannot be safely learned. Indeed, essentially the strongest class that *is* known to be learnable is "axis-aligned boxes," i.e., conjunctions of univariate inequalities [29]. The problem of learning effects, on the other hand, is essentially similar to regression under the "sup norm loss": that is, we demand a bound on the maximum error that holds with high probability. We can characterize the sample complexity of learning effects easily when the errors are considered under the ℓ_∞-norm, and observe that since all ℓ_q-norms are equivalent up to polynomial factors in the dimension, this in turn characterizes which families of effects are learnable for all ℓ_q norms.

Theorem 1. *(cf. [30, Theorem 3]) Let $\mathcal{A}$ be a class of functions mapping X to X, such that the true effects function A^* is in $\mathcal{A}$, and let $\mathcal{A}'_\epsilon$ be the set of Boolean-valued functions of the form $\{A'(s) = I[\|A(s) - A^*(s)\|_\infty \leq \epsilon] : A \in \mathcal{A}\}$. Let d be the VC-dimension of $\mathcal{A}'_\epsilon$. Suppose training and test problems are drawn from a common distribution D. Then $\Omega(\frac{1}{\delta_1}(d + \log\frac{1}{\delta_2}))$ training trajectories from D are necessary to identify $A \in \mathcal{A}$ that satisfy $\|A(s) - A^*(s)\|_\infty \leq \epsilon$ with probability $1-\delta_1$ on test trajectories with probability $1-\delta_2$ over the training trajectories. In particular, if $d = \infty$, then $\mathcal{A}$ is not learnable.*

Proof. The sup norm regression problem can be reduced to learning of effects as follows: given a training set $\{(x_i, f^*(x_i))\}_{i=1}^m$, we construct one-step trajectories for a planning domain with a single action, initial states given by x_i, and post-states given by $f^*(x_i)$. Then an estimate of the effect f that is ϵ-close to f^* with probability $1-\delta_1$ indeed yields a solution to the original regression problem. The bound thus follows from Theorem 3 of [30]. □

Thus, we see that some restrictions on the family of effects are necessary for learnability. Fortunately, unlike preconditions, these restrictions are relatively mild. For example, for linear functions in k dimensions, the VC-dimension of the corresponding $\mathcal{A}'_\epsilon$ is $O(k^2)$ [30, Prop. 18].

3.2 Assumptions

In the numeric planning setting we consider, the signature of actions in a trajectory is observable and the actions' preconditions are conjunctions of conditions over both discrete and numeric state variables. In addition, we limit our attention to scenarios that satisfy the following assumptions: (1) The conditions over the numeric state variables in actions' preconditions are linear inequalities, (2) The numeric expressions defining actions' effects are linear combinations of state variables, and (3) The set of numeric state variables that are involved in each action's preconditions and effects are known in advance.

While these assumptions restrict the types of domains we consider, they still cover a wide range of applications. The first two assumptions hold in most of the domains in the 3rd International Planning Competition (IPC) for numeric planning [4] and other benchmarks we considered [31]. The third assumption requires a human modeler to specify the relevant state variables, which is still significantly easier than manually defining the entire action model. Without this assumption, the space of possible preconditions may become intractably large. Next, we propose Numeric SAM (*N-SAM*), an action model learning algorithm for numeric domains that under the above assumptions is guaranteed to output a safe action model.

4 Numeric SAM (*N-SAM*)

The *N-SAM* algorithm learns an action model that includes all actions it observed in the given trajectories $\mathcal{T}$. First, it uses *SAM* learning [14] to learns the Boolean preconditions and effects of every observed action. Then, it creates numeric preconditions for every observed action a by constructing a convex hull over the relevant numeric variables' values observed in states before a was applied. Finally, it creates numeric effects by solving a linear regression problem for every numeric variable that is part of the effects of that action. Figure 1 illustrates the learning process of the numeric preconditions and effects. Next, we describe these steps in detail.

Learning Numeric Preconditions. For any action a, let $pre_X(a)$ be the set of numeric state variables used in its preconditions. If $pre_X(a) = \emptyset$, then of course there is no need to learn numeric preconditions for a. Otherwise, *N-SAM* creates a dataset of $|pre_X(a)|$-dimensional points by iterating over every observed state transition $\langle s, a, s' \rangle$ and extracting from s the values for the variables in $pre_X(a)$. We refer to this dataset as $DB_{pre(a)}$. [29] observed that when learning from positive examples, the optimal hypothesis is the intersection of all consistent candidate hypotheses. In our case, this is precisely the *convex hull* of the observed points. Thus, *N-SAM* computes the convex hull of the points in $DB_{pre(a)}$, and sets the preconditions of a as the set of linear inequalities that define the convex hull. Obtaining these inequalities can be done with off the shelf tools in polynomial time in the number of inequalities and attributes. [2] The top part of Figure 1 illustrates this process of learning numeric preconditions with *N-SAM*. We currently do not support equality constraints, and so we cannot represent a lower-dimensional convex hull, as may occur for example, if we do not have enough datapoints or if two numeric variables are linearly dependent. In such cases, *N-SAM* creates a degraded version of the preconditions that is a disjunction over the relevant parts of the previously observed states in which a has been applied.

Learning Numeric Effects. As when learning preconditions, we define $eff_X(a)$ to be the set of numeric state variables used in the effects of a. Under our linear effects assumption, the change in any variable $x \in eff_X(a)$ is a linear combination of the values of $eff_X(a)$ in the state immediately before a was applied. Thus, we propose to learn the effects of an action using standard methods for linear regression. In more detail, for every variable $x \in eff_X(a)$ and given state transition $\langle s, a, s' \rangle$ *N-SAM* creates an equation of the form $s'(x) - s(x) = w_0 + \sum_{x' \in eff_X(a)} w_{x'} \cdot s(x')$. If the resulting system of linear equations contains more than $|eff_X(a)| + 1$ linearly independent equations, then we can solve them to obtain the values of w_0 and $w_{x'}$ for all $x' \in eff_X(a)$ (note that otherwise the convex hull has lower dimension).[3] Correspondingly, *N-SAM* sets $x := w_0 + \sum_{x' \in eff_X(a)} w_{x'} \cdot x'$ as an effect of a. This process is illustrated in the bottom of Figure 1.

5 Theoretical Properties

The runtime of *N-SAM* is polynomial in the number of state transitions, state variables, and actions, because computing convex hulls and solving linear regression problems can be done in polynomial time. Regarding safety, we first show that the preconditions we learn are safe for a broad family of planning models in which the constraints are *convex*. Recall that a set of points (in our case, the points satisfying the precondition) is said to be convex if for any two points s and t in the set, every *convex combination* $\lambda s + (1 - \lambda)t$ for $\lambda \in [0, 1]$ is also in the set. In particular, linear inequalities define a convex set.

Theorem 2. *Consider a family of preconditions given by conjunctions of convex properties. Then the precondition given by the convex hull of states from a set of trajectories in which a given action was taken is safe.*

Proof. Consider any point s' in the convex hull; s' may be written $\lambda s_i + (1 - \lambda)s_j$ for states s_i and s_j that satisfied the action's preconditions. Note that s_i and s_j both satisfied all of the conditions in the conjunction defining the actual precondition; since these conditions are convex, s' also satisfies each of them, so s' satisfies the actual preconditions of the action. □

Next, we show that if furthermore the actions are affine functions of the pre-state, then the effects are also accurate given the convex hull precondition is satisfied.

Theorem 3. *Fix $q \in \mathbb{N} \cup \{\infty\}$. Suppose Θ is a set of parameters such that for all pre-states s of a given action in a set of trajectories, Θs is ϵ-close to the post-state in the ℓ_q-norm. Suppose furthermore that the true action model is given by an affine function with parameters Θ^*. Then for any state s satisfying the convex hull precondition, Θs is also ϵ-close to the true post-state in ℓ_q norm.*

Proof. We wish to show $\|\Theta s - \Theta^* s\| \leq \epsilon$. Note that since both functions are affine, $\Theta s - \Theta^* s = (\Theta - \Theta^*)s$, and since s is in the convex hull of observed points, $s = \sum_{j=1}^{m} \lambda_j s_j$

[2] In our implementation, we used the convex hull algorithm available in the SciPy library.

[3] In our implementation, we used a standard least-squares linear regression algorithm to obtain these weights.

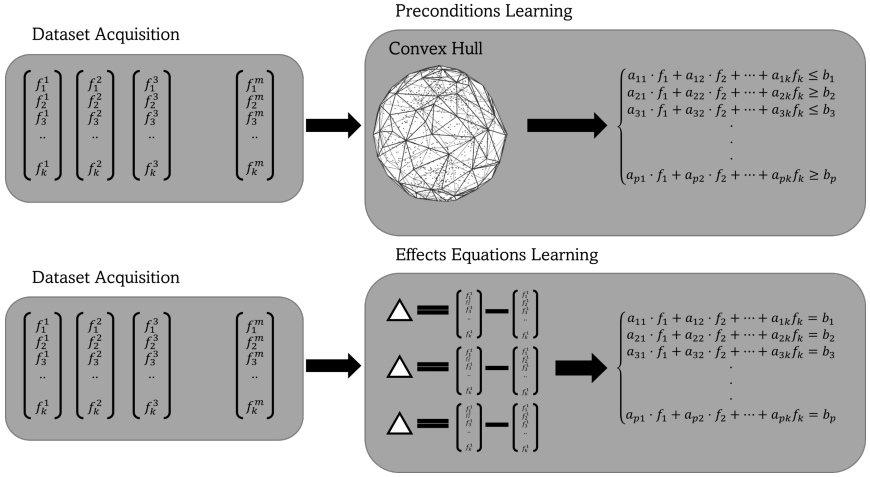


Figure 1: Graphical illustration of how *N-SAM* learns numeric preconditions and effects.

for $\lambda_j \in [0,1]$ such that $\sum_{j=1}^{m} \lambda_j = 1$. So, by the triangle inequality, $\|\Theta s - \Theta^* s\| \leq \sum_{j=1}^{m} \lambda_j \|(\Theta - \Theta^*)s_j\|$. We are supposing that $\|(\Theta - \Theta^*)s_j\| \leq \epsilon$ for all j, so in turn this is at most $\sum_{j=1}^{m} \lambda_j \epsilon = \epsilon$. □

Thus, *N-SAM* is guaranteed to return a safe action model. However, that action model can be too restrictive, raising the mentioned above completeness risk. Unlike *SAM* learning in discrete domains, *N-SAM* does not have nice worst-case sample complexity guarantees, as shown in Section 3.1.

6 Experimental Results

Although we do not obtain theoretical completeness guarantees for worst-case distributions, it remains an empirical question whether or not *N-SAM* learns useful action models in practice. We fully implemented *N-SAM* and performed a set of experiments to evaluate its performance on a set of benchmark numeric problems. In this section, we describe these experiments and their results. We considered a total of 12 domains for our evaluation, listed in Table 1. The Depot, Zenotravel, Rovers, Satellite, Settlers, and UMT domains are numeric domains from the 3rd International Planning Competition (IPC3) [32]. The Farmland, Counters, Plant-watering, and Sailing are more recent domains introduced by [31]. The first column in Table 1 contains the names of the domains. The next three column lists properties of the domain that relates to the applicability of *N-SAM*. The second column (labelled "L") indicate whether the preconditions and effects in the domain satisfy our linearity assumption, i.e., include only linear combination of the state variables. The third column ("CE") indicates whether the domain contain conditional effects. The fourth column ("EP") indicates whether the domain contain preconditions with equality, as opposed to inequalities. Currently, *N-SAM* supports domains that are linear and do not have conditional effects or equality preconditions. 7 out of the 12 domains satisfy these requirements. We highlighted these domains in bold. The other domains were discarded from our experiments. The columns $|A|, |F|, |X|, |C|$ represent the number of actions ($|A|$), Boolean state variables ($|F|$), and numeric state variables ($|X|$) in each domain. The column $\max pre_X$ is the maximal number of numeric variables involved in an action's precondition.

Domain	L	CE	EP	$\|A\|$	$\|F\|$	$\|X\|$	$max\ pre_X$	$max\ \|\mathcal{T}\|$
DriverLog (IPC)	✓	✗	✗	**6**	**5**	**4**	**0**	10
Farmland	✓	✗	✗	**2**	**1**	**2**	**1**	67
Rovers (IPC)	✓	✗	✗	**10**	**26**	**2**	**1**	11
Satellite (IPC)	✓	✗	✗	**5**	**8**	**6**	**2**	10
Sailing	✓	✗	✗	**8**	**1**	**3**	**3**	36
Depot (IPC)	✓	✗	✗	**5**	**6**	**4**	**3**	25
Counters	✓	✗	✗	**4**	**0**	**3**	**3**	20
Plant watering	✓	✗	✓	10	0	11	8	NA
Zenotravel (IPC)	✗	✗	✗	5	2	8	3	NA
Settlers (IPC)	✓	✓	✗	24	20	6	2	NA
UMT (IPC)	✓	✓	✓	38	38	24	11	NA

Table 1: Domains considered in our evaluation. The domains marked in bold are the domains in which all our assumptions hold.

For each domain, we performed the following type of experiments. First, we created a set of trajectories by solving planning problems in the domain using a numeric planner. Then, we run *N-SAM* on this set of trajectories, which outputs an action model $\hat{A}$. Next, we select a different planning problem from the same domain, and check if the same numeric planner can solve it given the learned action model $\hat{A}$. To obtain problems in a given domain, we either relied on an available set of problems or generated ones if a problem generator was available. To solve problems, we used the well-known Metric FF numeric planner (version 2.1) [23] with the running configuration EHC+H and then BFS with no cost minimization. We limited the running time of the solver to 60 seconds per problem. Our dataset of problems consists of only the problems the planner was able to solve within the time limit. Column $\max |\mathcal{T}|$ in Table 1 lists the maximal number of trajectories we had available in each domain. Since the number of trajectories is small, we used the leave-one-out cross-validation (LOOCV) method. That is, given X problems in the domain, we used $X - 1$ trajectories to learn the action model and evaluate it on the held-out problem. This process is repeated X times and the reported results are averaged over these repetitions. All experiments were run on a CentOS Linux CPU cluster with 6 cores and 32 GB of RAM.

domain	$\|\mathcal{T}\|$	Dis.	R_X	MSE_X	P_F	P_F	Solved
DriverLog	10	1.00	1.00	0.00	0.93	1.00	1.00
satellite	10	0.80	0.97	0.00	0.95	0.97	0.30
rovers	11	0.99	0.94	0.00	0.77	0.84	0.41
counters	20	1.00	0.93	0.00	1.00	1.00	0.81
depot	25	1.00	0.97	0.00	0.97	1.00	0.77
sailing	36	0.87	0.92	0.00	1.00	1.00	0.73
farmland	67	0.50	1.00	0.00	0.50	1.00	0.98

Table 2: Performance results of *N-SAM* for the maximal number of trajectories in each domain.

6.1 Evaluation Metrics

The main metric we consider is the number of problems solved using the learned domain. We also measured the relation between the learned action model $\hat{A}$ and the actual action model A by computing precision and recall of $\hat{A}$, where precision is $\frac{TP}{TP+FP}$ and recall is $\frac{TP}{TP+FN}$. We defined TP, FP, and FN differently for the Boolean and the numeric parts of $\hat{A}$. For the Boolean parts, TP is the number of Boolean preconditions that are in both $\hat{A}$ and A; FP is the number of Boolean preconditions that are in $\hat{A}$ but not in A; and FN is the number of Boolean preconditions that are in A but not in $\hat{A}$. These definition do not carry over well to numeric parts of the action model due to its continuous nature. Instead, we calculate the values of TP, FP and FN by iterating over the trajectories created for the test problems using the correct action model A and checking for every given state transition $\langle s, a, s' \rangle \in \mathcal{T}$ if the action a is also applicable in s according to $\hat{A}$. Here, TP is the number of triplets where a is applicable in its pre-state according to $\hat{A}$, FN is the number of triplets where a is not applicable in its pre-state according to $\hat{A}$. FP is set to zero, since it represents triplets where a is not applicable in its pre-state according to A. Such triplets do not exist as the trajectory itself was created using A. Thus, for the numeric action model precision is always one and thus only recall is of interest. Finally, we also measured the Mean Squared Error of the numeric effects, comparing the post-state in the trajectory created with A and the expected post-state according to $\hat{A}$. We denote by P_F and R_F the precision and recall of the Boolean part of the action model and denote by R_X, and MSE_X the recall and MSE of the numeric part.

6.2 Learning Results

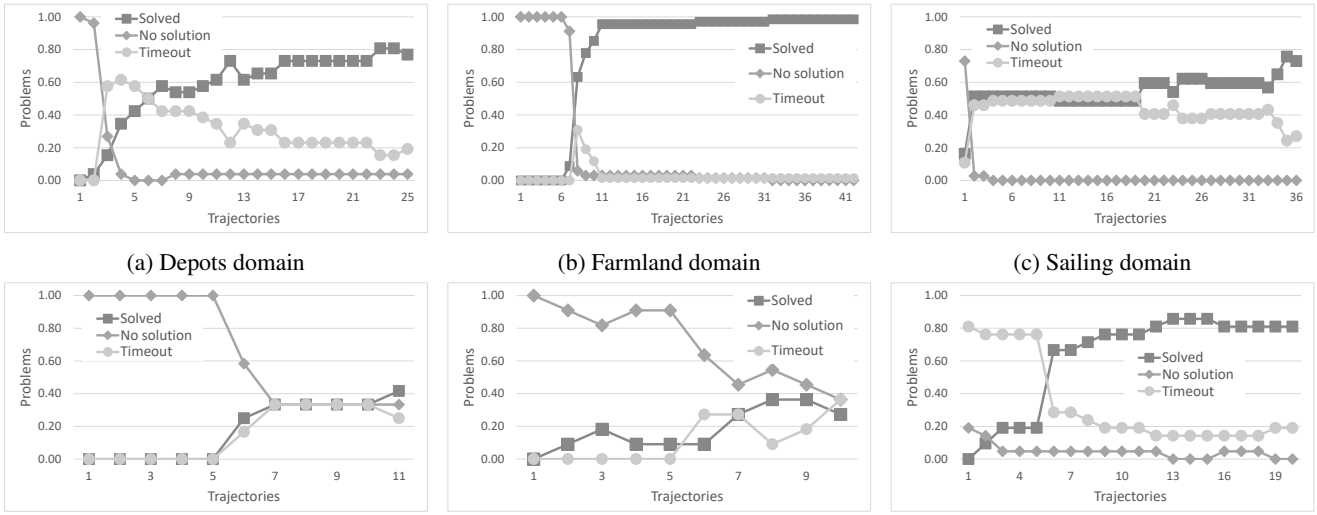
Table 2 presents results of our experiments. The column $|\mathcal{T}|$ refers to the maximal number of trajectories used in each domain. The column "Dis." gives the ratio of actions observed by by *N-SAM* out of the entire set of actions in A. The columns "Solved" indicate the ratio of problems solved using *N-SAM*. The results show several trends. First, we observe that even with such a small number of trajectory, *N-SAM* is able to learn a safe action model that allows solving more than half of the problems. Second, with the exception of Driverlog, the worst results appear in the domains with the fewest trajectories (satellite and rovers). This suggests having more trajectories will allow *N-SAM* to succeed in these two domains as well. The remarkable results for DriverLog — solving all problems with only 10 trajectories — can be explained by the simplicity of this domain. In particular, the drivelog domain has no numeric precondition but only numeric effects, which are much easier to learn. Interestingly, some actions remained undiscovered in some of the domains. The satellite, rover, farmland, and sailing domains are concrete examples. Yet, even in these cases, the learned action model was able to solve some of the tested problems. As expected, since *N-SAM* returns a safe action model, the MSE of all actions' effects is constantly 0. This means *N-SAM* learned the numeric effects perfectly. Also impressive are the recall values of the numeric preconditions (R_X) are constantly higher than 0.9, suggesting that the learned action models are not too restrictive, allowing most values that are allowed according to the original model. We calculated the discrete precision and recall values based on the observed actions. The discrete precision and recall values were affected since some actions were not learned due to safety issues. The farmland domain exhibits this since the discrete model precision is only 0.5. Since an unlearned action has no redundant predicates, the recall values are not badly influenced (as seen in the farmland domain).

Next, we explore why some problems were not solved with the learned action model. When using a planner to solve a problem with an action model learned by *N-SAM*, we can expect one of three outcomes: (1) the planner was able to solve the problem, (2) the planner declared the problem cannot be solved with the given action model, and (3) the planner was not able to solve the problem within the given time limit. We refer to these outcomes as "solved", "no solution", and "timeout", respectively. We note that the benchmark problems are all solvable by design. Thus, a "no solution" outcome indicates the learned action model is too restrictive, while a "timeout" outcome may only mean that the planner was not fast or efficient enough. Figure 2 shows how many problems reached each of these outcomes as a function of the number of trajectories used for training. We omitted the DriverLog since, as noted above, it has no numeric preconditions and *N-SAM* learns numeric effects very fast. As the results show, in all cases the number of "no solution" outcomes decreases as we are given more trajectories. This supports that the action model returned by *N-SAM* is becoming less restrictive (yet still safe) with more data. On the other hand, the number of "timeout" outcomes increases with the number of trajectories, as having more data results in a richer model, which allows the planner to explore a larger search space. Thus, in some cases adding more trajectories did not lead to an increase in the number of problems solved. That being said, the general trend in all domains is that increasing the number of data results in more "solved" outcomes, as desired. This is especially visible in farmland, depot, DriverLog, and counters domain.

7 Use Case: Repairing Faulty Action Models

The main use case of *N-SAM* is where the problem solver has no knowledge of the underlying action model and all the state transitions in the observed trajectories are *valid*. That is, in every state transition $\langle s, a, s' \rangle$ all the preconditions of a were met in s and the difference between s' and s exactly correspond to a's effects. However, *N-SAM* can also be used to repair a *faulty action model*.

Consider the following use case, which involves a planning-based autonomous agent acting in a domain $D = \langle F, X, A \rangle$. The agent uses an action model $A_f \neq A$ to plan which actions to perform in the domain in order to achieve its goals. Since $A_f \neq A$, some of the plans generated by this agent may fail. The agent does not know A, but it has full



(a) Depots domain (b) Farmland domain (c) Sailing domain

(d) Rovers domain (e) Satellite domain (f) Counters domain

Figure 2: Ratio of problems for each possible planner outcome as a function of # of trajectories.

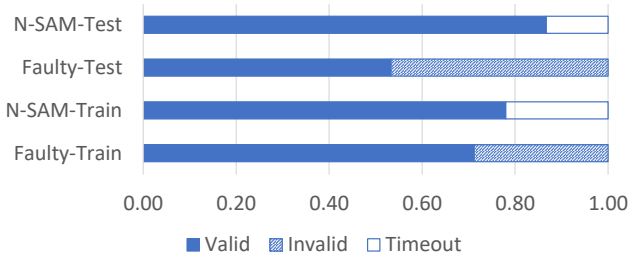


Figure 3: Model repair results for the sailing domain.

observability. Thus, it can collect a set of state transitions from previously executed plans and split them into valid and and invalid state transitions. This is where *N-SAM* fits naturally. We can run *N-SAM* only on the valid state transition, and then update the faulty action model A_f with the safe action model returned by *N-SAM*. The resulting action model is safe, and will become more complete as it processes more trajectories.

We created a preliminary implementation of this *N-SAM* use case and demonstrated its applicability on the *sailing* domain (see Table 1 for domain details). To create the faulty action model A_f, we injected a faulty to the domain's action model by changing the sign of one random numeric precondition (i.e. if the sign was $\geq$ we changed it to $\leq$). Then, we generated two sets of problems denoted *Train* and *Test*, comprising 59 and 15 problems, respectively. We used Metric FF with the faulty action model to generate plans for all problems in Train and in Test. The subset of valid problems created for Train is given as input to *N-SAM*, which outputs an action model $A_{N\text{-}SAM}$. Then, we used Metric FF with $A_{N\text{-}SAM}$ to generate plans for Train and for Test. Figure 3 shows the ratio of problems for which (1) a valid plan has been generated ("Valid" in the figure), (2) an invalid plan has been generated ("Invalid"), and (3) no plan was found until a timeout has been reached ("Timeout").

As can be seen, the action model created by *N-SAM* is able to generate more valid plans than the original faulty action model. In addition, since $A_{N\text{-}SAM}$ is a safe action model using it never yields an invalid plan. These preliminary results suggest that *N-SAM* can be applied for repairing faulty action models. But, further improvements for this use case can be done. For example, *N-SAM* can be extended to consider also the failed state transitions. Such negative examples can be extremely useful for training purposes. This is left to future work.

8 Conclusions and Future Work

We explored the problem of learning a safe action model for numeric planning. Unlike the discrete case, guaranteeing worst-case safety in a non-trivial way is not possible for most numeric planning domains. However, we identified as set of reasonable assumptions in which such learning is possibly, namely, that preconditions and effects are linear. Then, we proposed the *N-SAM* algorithm for learning a safe action model under these assumptions. The worst-case sample complexity of *N-SAM* does not scale gracefully, but it works well on standard numeric planning benchmarks, requiring less than 70 sample trajectories to learn an action model that is sufficient to find safe plans for most problems in almost all benchmark domains. This suggests *N-SAM* can be applied in practice in domains that satisfy our assumptions. We also show that *N-SAM* can be used to repair a faulty action model by learning from successful state transitions created with the faulty action model.

Future work includes implementing *N-SAM* in richer domains, e.g., domains that include equality constraints and polynomial effects and preconditions. Similarly, we are currently working on extending *N-SAM* to handle partial observability. At the same time, our success in learning adequate models for these domains contrasts strongly with the negative theoretical results for the worst case. It is an interesting question whether we can identify some properties that these domains possess that enables the observed success of *N-SAM* or some other method.

Acknowledgements

This research is partially funded by NSF award IIS-1908287 to Brendan Juba; and BSF grant #2018684 to Roni Stern.

References

[1] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated planning and acting*. Cambridge University Press, 2016.

[2] Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4), 1971.

[3] Constructions Aeronautiques, Adele Howe, Craig Knoblock, ISI Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, David Wilkins SRI, Anthony Barrett, Dave Christianson, et al. Pddl| the planning domain definition language. *Technical Report, Tech. Rep.*, 1998.

[4] Maria Fox and Derek Long. Pddl2.1: An extension to pddl for expressing temporal planning domains. *Journal of artificial intelligence research*, 20, 2003.

[5] Scott Sanner. Relational dynamic influence diagram language (rddl): Language description. *Unpublished ms. Australian National University*, 32:27, 2010.

[6] Jörg Hoffmann. Ff: The fast-forward planning system. *AI magazine*, 22(3):57–57, 2001.

[7] Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[8] Amanda Coles, Andrew Coles, Maria Fox, and Derek Long. Temporal planning in domains with linear processes. In *Twenty-First International Joint Conference on Artificial Intelligence*, 2009.

[9] Frédéric Maris and Pierre Régnier. Tlp-gp: Solving temporally-expressive planning problems. In *2008 15th International Symposium on Temporal Representation and Reasoning*, pages 137–144. IEEE, 2008.

[10] Wiktor Mateusz Piotrowski, Maria Fox, Derek Long, Daniele Magazzeni, and Fabio Mercorio. Heuristic planning for pddl+ domains. In *Workshops at the Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[11] Stephen Cresswell and Peter Gregory. Generalised domain model acquisition from action traces. In *International Conference on Automated Planning and Scheduling (ICAPS)*, pages 42–49, 2011.

[12] Diego Aineto, Sergio Celorrio, and Eva Onaindia. Learning action models with minimal observability. *Artificial Intelligence*, 275:104–137, 05 2019.

[13] Qiang Yang, Kangheng Wu, and Yunfei Jiang. Learning action models from plan examples using weighted max-sat. *Artificial Intelligence*, 171(2-3), 2007.

[14] Brendan Juba, Hai S. Le, and Roni Stern. Safe learning of lifted action models. In *International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 379–389, 2021.

[15] Roni Stern and Brendan Juba. Efficient, safe, and probably approximately complete learning of action models. In *the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 4405–4411, 2017.

[16] Brendan Juba and Roni Stern. Learning probably approximately complete and safe action models for stochastic worlds. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2022.

[17] Julian Schrittwieser, Thomas Hubert, Amol Mandhane, Mohammadamin Barekatain, Ioannis Antonoglou, and David Silver. Online and offline reinforcement learning by planning with a learned model. *Advances in Neural Information Processing Systems*, 34:27580–27591, 2021.

[18] Rahul Kidambi, Aravind Rajeswaran, Praneeth Netrapalli, and Thorsten Joachims. Morel: Model-based offline reinforcement learning. In *Advances in Neural Information Processing Systems 33*, 2020.

[19] Tianhe Yu, Garrett Thomas, Lantao Yu, Stefano Ermon, James Y Zou, Sergey Levine, Chelsea Finn, and Tengyu Ma. Mopo: Model-based offline policy optimization. In *Advances in Neural Information Processing Systems 33*, pages 14129–14142, 2020.

[20] Stephen N Cresswell, Thomas L McCluskey, and Margaret M West. Acquiring planning domain models using locm. *The Knowledge Engineering Review*, 28(2):195–213, 2013.

[21] Diego Aineto, Sergio Jiménez Celorrio, and Eva Onaindia. Learning action models with minimal observability. *Artificial Intelligence*, 275:104–137, 2019.

[22] José Á Segura-Muros, Raúl Pérez, and Juan Fernández-Olivares. Discovering relational and numerical expressions from plan traces for learning action models. *Applied Intelligence*, 51(11):7973–7989, 2021.

[23] Jörg Hoffmann. The metric-ff planning system: Translating"ignoring delete lists"to numeric state variables. *Journal of artificial intelligence research*, 20:291–341, 2003.

[24] Enrico Scala, Patrik Haslum, Sylvie Thiébaux, and Miquel Ramirez. Interval-based relaxation for general numeric planning. In *European Conference on Artificial Intelligence (ECAI)*, pages 655–663, 2016.

[25] Michael Kearns, Ming Li, and Leslie Valiant. Learning boolean formulas. *Journal of the ACM (JACM)*, 41(6):1298–1328, 1994.

[26] Haim Shvaytser. A necessary condition for learning from positive examples. *Machine Learning*, 5(1):101–113, 1990.

[27] Paul W. Goldberg. *PAC-learning geometrical figures*. PhD thesis, University of Edinburgh, 1992.

[28] Jyrki Kivinen. Learning reliably and with one-sided error. *Mathematical systems theory*, 28(2):141–172, 1995.

[29] B. K. Natarajan. Probably approximate learning of sets and functions. *SIAM Journal on Computing*, 20(2):328–351, 1991.

[30] Martin Anthony, Peter Bartlett, Yuval Ishai, and John Shawe-Taylor. Valid generalisation from approximate interpolation. *Combinatorics, Probability and Computing*, 5(3):191–214, 1996.

[31] Enrico Scala, Patrik Haslum, Daniele Magazzeni, Sylvie Thiébaux, et al. Landmarks for numeric planning problems. In *IJCAI*, pages 4384–4390, 2017.

[32] Derek Long and Maria Fox. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research*, 20:1–59, 2003.