Learning Safe Numeric Action Models

Argaman Mordoch[1], **Brendan Juba**[2], **Roni Stern**[1]

[1]Ben Gurion University in Be'er Sheva, Israel
[2]Washington University in St. Louis, USA
mordocha@post.bgu.ac.il, bjuba@wustl.edu, roni.stern@gmail.com

Abstract

Powerful domain-independent planners have been developed to solve various types of planning problems. These 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 more challenging in mission-critical domains, where a trial-and-error approach to 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. Learning safe action models for planning has been recently explored 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 Learning (N-SAM) 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. We analyze its worst-case sample complexity, which may be intractable for some domains. Empirically, however, N-SAM can quickly learn a safe action model that can solve most problems in the domain.

Introduction

Planning is the fundamental task of choosing which actions to perform to achieve the desired outcome. An *automated domain-independent planner* refers to an Artificial Intelligence (AI) capable of solving a wide range of planning problems (Ghallab, Nau, and Traverso 2016). Developing a domain-independent planner is a long-term goal of AI research. 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 (Fikes and Nilsson 1971), the Planning Domain Definition Language (PDDL) (Aeronautiques et al. 1998), PDDL 2.1 (Fox and Long 2003), and RDDL (Sanner 2010). Powerful corresponding planners have been developed, such as FastForward (Hoffmann 2001) and FastDownward (Helmert 2006) for problems given in PDDL and COLIN (Coles et al. 2009), TLP-GP (Maris and Régnier 2008), and DiNo (Piotrowski et al. 2016) for PDDL2.1 problems.

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 (Cresswell and Gregory 2011; Aineto, Celorrio, and Onaindia 2019; Yang, Wu, and Jiang 2007; Juba, Le, and Stern 2021). Since the learned model may differ 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 generate plans for solving solvable problems. This work focuses 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*, i.e., it generates plans that are applicable and achieve their goals. Learning safe action models for planning has been recently explored by the Safe Action Model Learning (SAM) family of algorithms (Stern and Juba 2017; Juba, Le, and Stern 2021; Juba and Stern 2022). These algorithms are inapplicable for numeric planning, where states include continuous state variables. The same safety considerations have motivated work in offline RL (Kidambi et al. 2020; Yu et al. 2020; Levine et al. 2020). Yet planning models can be reused for various goals, in contrast to standard RL that trains to a specific reward function.

We explore the problem of learning a safe action model for numeric planning. Specifically, we focus on problems defined in PDDL2.1 (Fox and Long 2003), a popular language for describing deterministic, fully observable numeric planning problems. We prove that learning a safe action model is impossible without making assumptions about the preconditions and effects of the agent's actions. Then, we identify a reasonable set of assumptions in which learning a safe action model is possible. For domains that satisfy these assumptions, 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. Even with our assumptions, the worst-case sample complexity of N-SAM is intractable. However, we show empirically that with fewer than 60 trajectories, N-SAM can learn action models that can solve most of the relevant problems in two standard numeric planning benchmarks. This, coupled with its safety guarantee, suggests the practical applicability of N-SAM.

Background and Problem Definition

We focus on planning problems in domains where action outcomes are deterministic and states are fully observable, and described with discrete and continuous state variables. Such problems are commonly modeled using the PDDL2.1 (Fox and Long 2003) language.[1] We introduce the following notation to define a numeric planning problem in PDDL2.1. 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 $\langle \mathit{name}(a), \mathit{param}(a), \mathit{pre}(a), \mathit{eff}(a) \rangle$ representing the action's name, parameters, preconditions, and effects, respectively. The preconditions of 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, $\mathit{Rel} \in \{\leq, <, =, >, \geq\}$, and k is a number. The effects of action a, denoted $\mathit{eff}(a)$, 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 $\langle x, \mathit{op}, \xi \rangle$ 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 $\mathit{pre}(a)$. Applying a in s, denoted $a(s)$, results in a state that differs from s only according to the assignments in $\mathit{eff}(a)$. A planning problem is defined by $\langle D, s_0, G \rangle$ where D is a domain, s_0 is the initial state, and G are the problem goals. The problem goals G are assignments 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 applicable in s_0 and resulting in a state s_G in which G is satisfied. A *trajectory* is a list of state transitions of the form $\langle s_i, a_i, a_i(s_i) \rangle$ 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.

Problem Definition. We consider a problem solver tasked with solving a numeric planning problem $\langle D = \langle F, X, A \rangle, s_0, G \rangle$. The main challenge is that the problem solver does not receive explicit information about the set of actions A. Instead, it receives a set of *trajectories* $\mathcal{T}$, created by executing solutions – plans – for other planning problems in the same domain D. A human operator, random exploration, or some other domain-specific process may have created these plans. We assume the problem solver has full observability of these trajectories, in the sense that it knows the value of every variable in every state in every trajectory $T \in \mathcal{T}$, and it knows the name and parameters of every action in every trajectory $T \in \mathcal{T}$.

Related Work. Different algorithms have been proposed for learning planning action models (Cresswell, McCluskey, and West 2013; Yang, Wu, and Jiang 2007; Aineto, Celorrio, and Onaindia 2019; Juba, Le, and Stern 2021). Some action-model learning algorithms, such as LOCM (Cresswell and Gregory 2011) and its extension LOCM2 (Cresswell, McCluskey, and West 2013), analyze observed plan sequences, where each action appears as an action name and arguments in the form of a vector of object names. Other action-model learning algorithms, such as FAMA (Aineto, Celorrio, and Onaindia 2019), can also utilize information about the states reached while executing plans in the domains. Most action model learning algorithms do not guarantee that plans created with the learned action model are applicable in the real action model. The SAM learning algorithm (Stern and Juba 2017) addresses 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 the 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 (Juba, Le, and Stern 2021; Juba and Stern 2022) are limited to learning action models that do not support numeric state variables. In fact, learning numeric action models has been scarcely studied. PlanMiner (Segura-Muros, Pérez, and Fernández-Olivares 2021; Segura-Muros, Fernández-Olivares, and Pérez 2021) is a notable exception. It is an algorithm that learns numeric action models from partially known and potentially noisy trajectories. PlanMiner does not, however, provide any safety guarantees.

Safe Numeric Planning with Offline Learning

Our approach for solving this problem, 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 (Hoffmann 2003) and ENHSP (Scala et al. 2016). 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. This work emphasizes addressing the safety risk, which is crucial 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* (Stern and Juba 2017; Juba, Le, and Stern 2021).

Definition 1 (Safe Action Model). *For $\epsilon \geq 0$, 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 $\hat{a} \in \hat{A}$ there exists $a \in A$ such that name(a) = name$(\hat{a})$ and 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*

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

obtained by applying a to s, i.e., $\|\hat{a}(s) - a(s)\| \leq \epsilon$.

Plans created with safe action models are safe in the sense that they execute as anticipated by the model up to a total error of magnitude at most ϵ times the length of the plan (by the triangle inequality). We allow for a small ϵ error in our definition because such an error 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*.

Theoretical Analysis

The problem of learning numeric preconditions and effects of a safe action model can be mapped to known results in the Probably Approximately Correct (PAC) Learning literature. This allows us to establish learnability results in our settings and identify which assumptions may enable efficient learning. To formalize the notion of efficient learning in our setting, we introduce the following terminology. We say that an action model solves a given problem if a complete planner using that action model will be able to solve it. We say that an action model learning algorithm is $(1-\epsilon)$-complete it returns an action model that can solve a given problem with probability at least $1-\epsilon$.[2]

Learning Preconditions Learning the preconditions of actions can be viewed as the problem of learning a Boolean-valued function, where the training examples are the given trajectories that include that action. The given trajectories were created by successfully executing plans in the domain. In every trajectory transition, the preconditions of the executed actions have necessarily been satisfied. Thus, learning the preconditions of actions is a special case of learning a Boolean-valued function from only positive examples Kearns, Li, and Valiant (1994).

Definition 2 (PAC Learning from Positive Examples)**.** *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 P 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 (1) h* has no false positives: *$h(\mathcal{X}) \subseteq c(\mathcal{X})$, and (2) h* is $1-\epsilon$ accurate: $\Pr_{x \in P}[h(x) = c(x)] \geq 1-\epsilon$, *where $\Pr_{x \in P}[h(x) = c(x)]$ means the probability that $h(x) = c(x)$ for an example $x \in \mathcal{X}$ sampled from a distribution P. 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.

Next, we reduce the problem of PAC learning from positive examples to the problem of learning preconditions of a safe action model.

Proposition 1. *Suppose there exists a safe action model learning algorithm $\mathcal{A}$ for domains with preconditions from $\mathcal{C}$ that produces preconditions from $\mathcal{H}$ such that $\mathcal{A}$ is guaranteed to be $1-\epsilon$-complete with probability $1-\delta$ when given at least $m(\epsilon,\delta)$ trajectories as input for some function m. Then there is a PAC learning algorithm for learning $\mathcal{C}$ with $\mathcal{H}$ using $m(\epsilon,\delta)$ positive examples. Moreover, if the safe action model learning algorithm is efficient, so is the PAC learning algorithm.*

Proof. Consider a domain with one action a and states given by $\mathcal{X}$ extended by one Boolean attribute, t. We provide the following trajectories to our safe action model learner: given an example x sampled from P for the problem of learning from positive examples, we construct the trajectory with $(x, 0)$ as the first state, i.e., with $t = 0$, followed by the action a and $(x, 1)$ as the final state. The associated goal is "$t = 1$." We obtain an action model from the algorithm run with δ and ϵ and given $m(\epsilon,\delta)$ examples, and return its precondition for a with t set to 0 as our solution h for PAC learning. Note that the trajectories are consistent with an action model in which $c \in \mathcal{C}$ (the correct hypothesis for the positive-example PAC learning problem) is the precondition of a and the only effect of a is to set $t = 1$. h must have no false positives because the precondition for a must be safe: if there exists $x \in \mathcal{X}$ such that $h(x) = 1$ but $c(x) = 0$, our action model would permit a 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 $1-\epsilon$ complete. Observe that in our distribution over examples, $t = 0$ initially, so to satisfy the goal, the plan must include the action a. Hence, the precondition for a must be satisfied with probability at least $1-\epsilon$ on P, or else the action model would fail with probability greater than ϵ. Hence, h is indeed as required for PAC learning. The "moreover" part is immediate from the construction. □

The above reduction allows us to identify which classes of preconditions cannot be efficiently learned by the family of Boolean-valued functions that cannot be efficiently learned from positive examples.

Corollary 1 ((Goldberg 1992))**.** *The family of preconditions given by single linear inequalities with at most two variables cannot be safely learned by any $\mathcal{H}$.*

Corollary 2 ((Kivinen 1995))**.** *The family of preconditions given by the disjunction of two univariate inequalities cannot be safely learned by any $\mathcal{H}$.*

In particular, classes of preconditions $\mathcal{C}$ that contain the above representations as special cases cannot be safely learned. The strongest class of Boolean-valued function that *is* known to be learnable is "axis-aligned boxes," i.e., conjunctions of univariate inequalities (Natarajan 1991).

Learning Effects. The problem of learning effects is essentially similar to regression under the "sup norm loss": 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.

[2] This, of course, assumes the training and testing problems are drawn from the same distribution of problems.

Theorem 1. *(cf. Anthony et al. (1996, 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 $\{(x_i, f^*(x_i))\}_{i=1}^m$, 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-δ_1 indeed yields a solution to the original regression problem. The bound thus follows from Theorem 3 of Anthony et al. (1996). □

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 $O(k^2)$ (Anthony et al. 1996, Prop. 18).

Assumptions

In our numeric planning setting, the actions' names and parameters are observable in the trajectories. The actions' preconditions are conjunctions of conditions over 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 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 variety of applications. The first two assumptions hold in most of the domains in the 3rd International Planning Competition (IPC) for numeric planning (Fox and Long 2003) and other benchmarks we considered (Scala et al. 2017). 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.

Numeric SAM (N-SAM)

The N-SAM algorithm learns an action model that includes all actions observed in the given trajectories $\mathcal{T}$. First, it uses SAM learning (Juba, Le, and Stern 2021) to learn every observed action's Boolean preconditions and effects. 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.

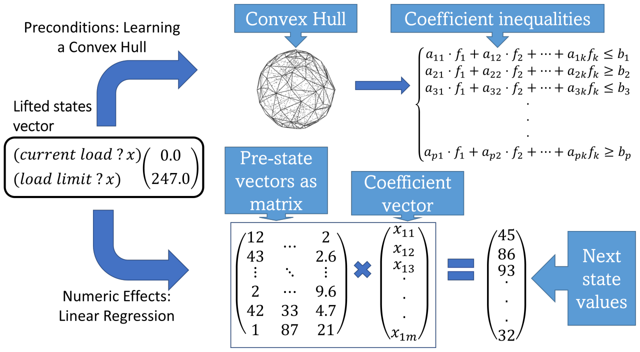

Preconditions: Learning a Convex Hull
Convex Hull
Coefficient inequalities
$a_{11} \cdot f_1 + a_{12} \cdot f_2 + \cdots + a_{1k} f_k \leq b_1$
$a_{21} \cdot f_1 + a_{22} \cdot f_2 + \cdots + a_{2k} f_k \geq b_2$
$a_{31} \cdot f_1 + a_{32} \cdot f_2 + \cdots + a_{3k} f_k \leq b_3$
$a_{p1} \cdot f_1 + a_{p2} \cdot f_2 + \cdots + a_{pk} f_k \geq b_p$
Lifted states vector
(current load ? x) 0.0
(load limit ? x) 247.0
Pre-state vectors as matrix
Coefficient vector
12 … 2
43 2.6
2 … 9.6
42 33 4.7
1 87 21
x_{11} x_{12} x_{13} x_{1m}
45 86 93 32
Next state values
Numeric Effects: Linear Regression

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

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 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 $pre_X(a)$. This dataset is denoted as $DB_{pre(a)}$. Natarajan (1991) 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. These inequalities can be obtained with off-the-shelf tools in polynomial time in the number of inequalities and attributes. [3] State variables that are linearly dependent on other state variables in the given trajectories are extracted from the convex hull, and the linear dependency is translated into equality preconditions. The top part of Figure 1 illustrates this process of learning numeric preconditions with N-SAM.

Example: consider a domain with numeric fluents f_1 and f_2 and an action that is observed in three transitions where the pre-states are $(f_1 = 0, f_2 = 0)$, $(f_1 = 1, f_2 = 0)$, and $(f_1 = 0, f_2 = 1)$. The preconditions learned for a with N-SAM are the inequalities: $0 \leq f_1, f_2 \leq 1$, and $f_1 + f_2 \leq \sqrt{2}$.

Learning Numeric Effects. Let $eff_X(a)$ be the set of numeric state variables relevant to computing the effects of a. Under the 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 before applying a. Thus, we learn the effects of an action using standard linear regression. In

[3] We used the convex hull algorithm available in the SciPy library in our implementation.

more detail, for every variable $x \in \mathit{eff}_X(a)$ and given state transition $\langle s, a, s' \rangle$ N-SAM creates an equation of the form $s'(x) = w_0 + \sum_{x' \in \mathit{eff}_X(a)} w_{x'} \cdot s(x')$. If the resulting system of linear equations contains fewer than $|\mathit{eff}_X(a)| + 1$ linearly independent equations, we consider a unsafe and do not include it in the returned action model. Otherwise, we find the unique solution to this set of equations and obtain the values of w_0 and $w_{x'}$ for all $x' \in \mathit{eff}_X(a)$.[4] Correspondingly, N-SAM sets $x := w_0 + \sum_{x' \in \mathit{eff}_X(a)} w_{x'} \cdot x'$ as an effect of a. This process is illustrated in the bottom of Figure 1.

Example: following our running example, assume that the post-state in the three observed transitions are $(f_1 = 1, f_2 = 0)$*,* $(f_1 = 2, f_2 = 0)$*, and* $(f_1 = 1, f_2 = 1)$*. N-SAM learns the effects that set the value of* f_1 *by solving the resulting three independent equations:(1)* $c = 1$*, (2)* $a + c = 2$*, and (3)* $b + c = 1$*. The solution for this set of equations is* $a = 1, b = 0, c = 1$*. For* f_2*, N-SAM will also determine using the same pre-state values that* a *does not affect the fluent* f_2*. Thus, the effect of the action is* $f_1 := f_1 + 1$*.*

Extension to Nonlinear Domains. N-SAM can be easily extended to learn preconditions and effects with polynomials of some low degree, e.g., quadratic or cubic polynomials. We achieve this by mapping our example trajectories into trajectories in a domain with a more extensive set of fluents: for each possible monomial up to the desired degree, we create a fluent taking value equal to the value of the corresponding monomial evaluated on the original fluents' values. For example, if in some state in an example trajectory fluents x and y take values 3 and 5, respectively, the new domain might have a fluent xy taking value 15 in that state in the new trajectory. Our original numeric fluents X are thus mapped to a set of $O(|X|^d)$ fluents, where d is the degree of the polynomials involved. We run N-SAM as before, obtaining a set of linear inequalities and effects in this new domain. We can then substitute the corresponding monomials for the fluents in those expressions to obtain polynomial inequalities and effects. We observe that polynomials in the original domain are representable as linear expressions in this expanded domain (and vice-versa). Thus, the expanded domain satisfies the linear preconditions and effects assumption; therefore, N-SAM can be applied, and the resulting preconditions and effects can be applied in the original domain.

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. For polynomial domains, the runtime is polynomial in the number of relevant monomials, which can be exponential in the degree of the polynomial. 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 preconditions given by the convex hull of states from a set of trajectories in which a given action was taken are 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 satisfied all the conditions in the conjunction defining the actual precondition; since these conditions are convex, s' also satisfies each of them, so s' satisfies the action's actual preconditions. □

Next, we show that if the actions are affine functions of the pre-state, the effects are also accurate, given that 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, $s = \sum_{j=1}^{m} \lambda_j s_j$ 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 above completeness risk. Unlike SAM learning in discrete domains, N-SAM does not have nice worst-case sample complexity guarantees.

Experimental Results

Although we do not obtain theoretical completeness guarantees for worst-case distributions, whether or not N-SAM learns applicable action models in practice is an empirical question. Thus, we implemented N-SAM and evaluated its performance on 12 benchmark domains, namely Depot, Driverlog, Zenotravel, Rovers, Satellite, Settlers, and UMT domains from the 3rd International Planning Competition (IPC3) (Long and Fox 2003), and Farmland, Counters, Plant-watering, and Sailing from (Scala et al. 2017). N-SAM's code is available in the link - https://github.com/Search-BGU/numeric-sam.

Table 1 provides details about these domains. The first column contains the domains' names. The next three columns list domain properties related to the applicability of N-SAM. The first column (labeled "L") indicates whether the preconditions and effects in the domain include only a linear combination of the state variables. The third column ("CE") indicates whether the domain contains conditional effects. The fourth column ("EP") indicates whether the domain contains preconditions with equality instead of inequalities. Currently,

[4] In our implementation, we use least-squares linear regression to obtain these weights.

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

Table 1: PDDL 2.1 domains. The bolded domains are those in which all our assumptions hold and are thus used in the experiments.

N-SAM does not support domains with conditional effects, and our current implementation does not support equality preconditions. 9 out of the 12 domains satisfy these requirements (highlighted in bold). We discarded the other domains from our experiments. The columns $|A|$, $|F|$, and $|X|$ represent the number of actions, Boolean state variables, and numeric state variables, respectively. The column $\max pre_X$ is the maximal number of numeric variables involved in a precondition.

For each domain, we performed the following type of experiments. First, we generated a set of trajectories by solving problems in the domain using a numeric planner. Then, we ran N-SAM on these trajectories, returning an action model $\hat{A}$. Next, we selected a different set of planning problems from the same domain, and checked if the same numeric planner can solve these problems given the learned action model $\hat{A}$. We generated the problems using publicly available problem generator. To solve problems, we run two well-known planners with a timeout of 60 seconds: Metric-FF (version 2.1) (Hoffmann 2003) and ENHSP (Li et al. 2018). [5] The planner with the best coverage under the given timeout was used, Metric FF for Rovers and Counters, and ENHSP for all other domains. In our experiments, we did not use unsolvable problems. These problems in each domain were split to train and test using a 5-fold cross-validation. All generated plans were validated using VAL (Fox and Long 2006). All experiments were run on a Linux machine with 8 cores and 16 GB of RAM.

Evaluation Metrics

The primary evaluation metric is the number of problems solved using the learned action model $\hat{A}$. We also measured the precision and recall of $\hat{A}$ with respect to the actual action model 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 $\hat{A}$. Due to its continuous nature, these definitions do not carry over well to numeric parts of the action model. Instead, we calculate the values of TP, FP, and FN by iterating over the trajectories created for the test problems using 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 was created using A. Thus, precision is always one for the numeric action model; therefore, only recall is of interest. Finally, we also measured the Mean Squared Error (MSE) 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, R_F, R_X, and MSE$_X$ the precision and recall of the Boolean part of the action model and the recall and MSE of the numeric part, respectively. These metrics were only computed for *observed* actions.

Domain	Dis.	R_X	MSE$_X$	P_F	R_F	Solved	Var
Farmland	0.50	1.00	0.00	0.50	1.00	0.98	0.01
DriverLog L	1.00	1.00	0.00	0.64	1.00	0.97	0.02
DriverLog P	1.00	1.00	0.00	0.64	1.00	1.00	0.00
Depot	1.00	0.99	0.00	0.77	1.00	0.93	0.13
Sailing	0.87	0.99	0.00	1.00	1.00	0.81	0.25
Counters	1.00	0.74	0.00	N/A	N/A	0.80	0.92
Satellite	1.00	0.99	0.00	0.74	1.00	0.66	0.10
Rovers	0.95	0.94	0.00	0.58	0.84	0.38	0.69
Zenotravel	0.80	0.99	0.00	0.84	1.00	0.94	0.05

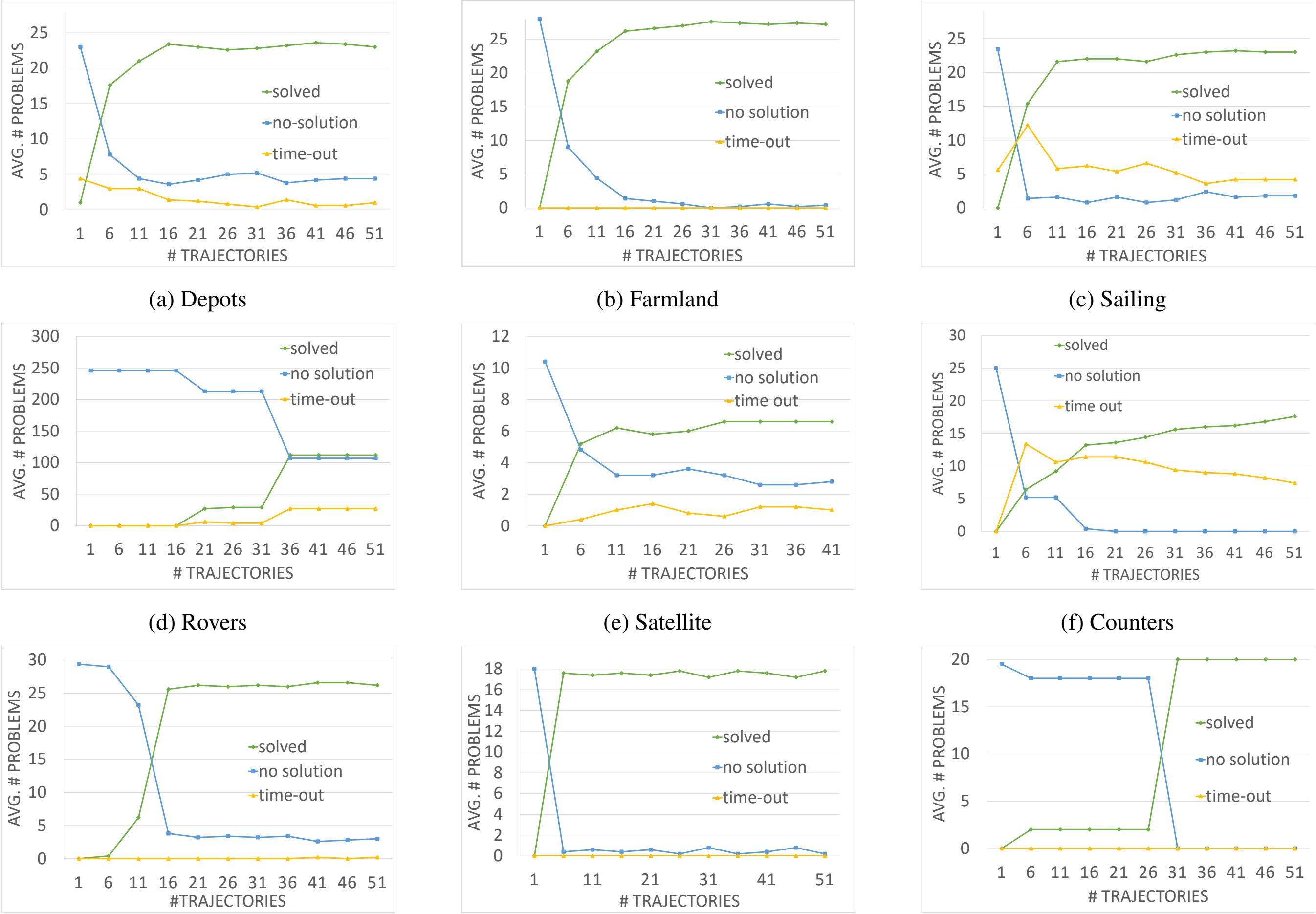
Table 2: N-SAM results for the max. # of trajectories in each domain.

Results

Table 2 presents the results of our experiments. The column "Dis." shows the ratio of actions in A observed by N-SAM. The columns "Solved" and "Var" show the average and variance, over the folds, of the ratio of problems solved using N-SAM. The DriverLog domain has two versions, one with only linear effects and the other with polynomial effects. We separated the domain accordingly, denoting the former as DriverLog Lin. and the latter as DriverLog Poly. The results show several trends. First, with only 51 trajectories for learning, N-SAM learned a safe action model that allowed solving more than half of the problems in all domains except Rovers.

Interestingly, this result was achieved even though some actions remained undiscovered in some domains, e.g., Satellite, Rover, Farmland, and Sailing. The limited success in Rovers may be attributed to the large number of fluents and actions in this domain (26 and 10, respectively). Since N-SAM returns a safe action model, the MSE of all actions' effects is constantly 0, i.e., N-SAM learned the numeric effects perfectly.

[5] For Metric-FF, we used BFS with no cost minimization and running configuration EHC+H. For ENHSP, we used Greedy Best First Search with the MRP heuristic and helpful actions.



(a) Depots

(b) Farmland

(c) Sailing

(d) Rovers

(e) Satellite

(f) Counters

(g) Zenotravel

(h) Driverlog Lin.

(i) Driverlog Poly.

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

The recall values for both numeric and Boolean preconditions (R_X and R_F) were almost perfect in all domains. Boolean precision results, while always higher than 0.5, were somewhat lower. This is explained by the fact that N-SAM learns *negative* preconditions that are missing from the PDDL representation.

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 unsolvable with the given action model, (3) the planner could not solve the problem within the given time limit. These outcomes are denoted as "solved", "no solution", and "timeout". Since the benchmark problems are all solvable by design, a "no solution" outcome indicates the learned action model is too restrictive, and a "timeout" outcome may only mean that the planner was not fast enough. Figure 2 shows how many problems reached each outcome as a function of the number of trajectories used for training. In all cases, the number of "no solution" outcomes decreases as we receive more trajectories, suggesting that the action model returned by N-SAM is becoming less restrictive (yet still safe) with more data. However, the number of "timeout" outcomes increases with the number of trajectories. This is because having more data results in a richer model, which allows the planner to explore a larger search space. That being said, the general trend in all domains is that increasing the quantity of data results in more "solved" outcomes, as desired.

Conclusions and Future Work

We explored the problem of learning a safe action model for numeric planning. Unlike the discrete case, we showed that guaranteeing safety for all numeric planning domains is impossible. Then, we proposed the N-SAM algorithm, which is capable of learning a safe action model under the assumption that the preconditions and effects are polynomials. The worst-case sample complexity of N-SAM does not scale gracefully, but we it works well in practice on standard benchmarks. Having fewer than 60 sample trajectories, N-SAM returns an action model sufficient to find plans for most problems in almost all benchmark domains. This suggests that N-SAM can be applied in practice in domains that satisfy our assumptions. Future work will extend N-SAM to support conditional effects, noisy input, and handle partial observability.

Acknowledgments

This research is partially funded by NSF awards IIS-1908287, IIS-1939677, and IIS-1942336 to Brendan Juba, BSF grant #2018684 to Roni Stern, and to the DARPA SAIL-ON program. This work was partially performed while Brendan Juba was at the Simons Institute for the Theory of Computing.

References

Aeronautiques, C.; Howe, A.; Knoblock, C.; McDermott, I. D.; Ram, A.; Veloso, M.; Weld, D.; SRI, D. W.; Barrett, A.; Christianson, D.; et al. 1998. Pddl| the planning domain definition language. *Technical Report, Tech. Rep.*

Aineto, D.; Celorrio, S. J.; and Onaindia, E. 2019. Learning action models with minimal observability. *Artificial Intelligence*, 275: 104–137.

Anthony, M.; Bartlett, P.; Ishai, Y.; and Shawe-Taylor, J. 1996. Valid generalisation from approximate interpolation. *Combinatorics, Probability and Computing*, 5(3): 191–214.

Coles, A.; Coles, A.; Fox, M.; and Long, D. 2009. Temporal planning in domains with linear processes. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

Cresswell, S.; and Gregory, P. 2011. Generalised domain model acquisition from action traces. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 42–49.

Cresswell, S.; McCluskey, T.; and West, M. 2013. Acquiring planning domain models using LOCM. *The Knowledge Engineering Review*, 28(2): 195–213.

Fikes, R. E.; and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4): 189–208.

Fox, M.; and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20: 61–124.

Fox, M.; and Long, D. 2006. Modelling mixed discrete-continuous domains for planning. *Journal of Artificial Intelligence Research*, 27: 235–297.

Ghallab, M.; Nau, D.; and Traverso, P. 2016. *Automated planning and acting*. Cambridge University Press.

Goldberg, P. W. 1992. *PAC-learning geometrical figures*. Ph.D. thesis, University of Edinburgh.

Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26: 191–246.

Hoffmann, J. 2001. FF: The fast-forward planning system. *AI magazine*, 22(3): 57–57.

Hoffmann, J. 2003. The Metric-FF Planning System: Translating "Ignoring Delete Lists" to Numeric State Variables. *Journal of Artificial Intelligence Research*, 20: 291–341.

Juba, B.; Le, H. S.; and Stern, R. 2021. Safe Learning of Lifted Action Models. In *International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 379–389.

Juba, B.; and Stern, R. 2022. Learning Probably Approximately Complete and Safe Action Models for Stochastic Worlds. In *AAAI Conference on Artificial Intelligence*.

Kearns, M.; Li, M.; and Valiant, L. 1994. Learning boolean formulas. *Journal of the ACM (JACM)*, 41(6): 1298–1328.

Kidambi, R.; Rajeswaran, A.; Netrapalli, P.; and Joachims, T. 2020. MOReL: Model-Based Offline Reinforcement Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*.

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

Levine, S.; Kumar, A.; Tucker, G.; and Fu, J. 2020. Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *arXiv preprint arXiv:2005.01643*.

Li, D.; Scala, E.; Haslum, P.; and Bogomolov, S. 2018. Effect-Abstraction Based Relaxation for Linear Numeric Planning. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 4787–4793.

Long, D.; and Fox, M. 2003. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research*, 20: 1–59.

Maris, F.; and Régnier, P. 2008. TLP-GP: Solving temporally-expressive planning problems. In *International Symposium on Temporal Representation and Reasoning (TIME)*.

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

Piotrowski, W.; Fox, M.; Long, D.; Magazzeni, D.; and Mercorio, F. 2016. Heuristic planning for PDDL+ domains. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

Sanner, S. 2010. Relational dynamic influence diagram language (RDDL): Language description. *Unpublished ms. Australian National University*.

Scala, E.; Haslum, P.; Magazzeni, D.; Thiébaux, S.; et al. 2017. Landmarks for Numeric Planning Problems. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 4384–4390.

Scala, E.; Haslum, P.; Thiébaux, S.; and Ramirez, M. 2016. Interval-based relaxation for general numeric planning. In *European Conference on Artificial Intelligence (ECAI)*, 655–663.

Segura-Muros, J. Á.; Fernández-Olivares, J.; and Pérez, R. 2021. Learning Numerical Action Models from Noisy Input Data. *arXiv preprint arXiv:2111.04997*.

Segura-Muros, J. Á.; Pérez, R.; and Fernández-Olivares, J. 2021. Discovering relational and numerical expressions from plan traces for learning action models. *Applied Intelligence*, 1–17.

Stern, R.; and Juba, B. 2017. Efficient, Safe, and Probably Approximately Complete Learning of Action Models. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 4405–4411.

Yang, Q.; Wu, K.; and Jiang, Y. 2007. Learning action models from plan examples using weighted MAX-SAT. *Artificial Intelligence*, 171(2-3): 107–143.

Yu, T.; Thomas, G.; Yu, L.; Ermon, S.; Zou, J. Y.; Levine, S.; Finn, C.; and Ma, T. 2020. MOPO: Model-based Offline Policy Optimization. In *Advances in Neural Information Processing Systems (NeurIPS)*, 14129–14142.