Evaluating Planning Model Learning Algorithms Across Different Representations

Anonymous Authors

Abstract

Model-based planning is a fundamental approach to sequential decision-making in which a domain model is used to generate plans. The model is typically represented as a set of action models, which define the preconditions and effects of each action. Obtaining action models for planning is a challenging task, and various algorithms have been proposed to learn them from observations. Evaluating learned action models is challenging since there are no standardized benchmarks or evaluation metrics. Moreover, different action model learning algorithms ourput different representations of for states and actions. To address this gap in the literature, we introduce a paradigm for evaluating action model learning algorithms. In our evaluation paradigm, each action model learning algorithm is required to output an action model and an encoder-decoder pair. The encoder-decoder pair is used to bridge representation gaps and enable measuring the problem-solving capabilities of the learned action models. We propose an evaluation scheme that leverages the encoder-decoder pair to systematically compare learned action models, and describe several evaluation metrics. A benchmark suite is also provided to facilitate the evaluation of action model learning algorithms, based on existing domain models from the International Planning Competition (IPC). We demonstrate our evaluation paradigm by applying it to several action model learning algorithms, including SAM, ESAM, FAMA, and ROSAME.

1 Introduction

2 Background and Related Work

2.1 Automated planning

[Omer: START - automated planning] automated planning is ......

[Omer: START- formal languege-[pddl ....pddl2.1]]

the process of planning requers a formal represtantation of the problem and the domain' as in pddl ....

[Omer: END- formal languege-[pddl ....]]

[Omer: START- fast downward, planners]

the formal representation is givern tot the planner that.....

[Omer: END- fast downward, planners]

[Omer: END - automated planning]

2.2 Action Model Learning

[Omer: START- different learning configurations numeric(NSAM),, disorderd , and st.] .... [Omer: END-different learning configurations numeric(NSAM),, disorderd , and st.]

Different algorithms have been proposed for learning planning action models for single-agent planning. Most action model learning algorithms learn action models by processing a set of *trajectories*, which are given as input. A *trajectory* $T = \langle s_0, a_1, s_1, \dots a_n, s_n \rangle$ is an alternating sequence of states $(s_0, \dots, s_n)$ and actions $(a_1, \dots, a_n)$ that starts and ends with a state, where $s_i = a_i(s_{i-1})$ for $i = 1, \dots, n$. Some action model learning algorithms require access to all the actions and states in the given trajectories, while others can handle missing states and actions. For example, the ARMS algorithm **?** requires only the initial and final states of the provided trajectories, with optional use of intermediate states if available. For each observed action, ARMS lifts the action and constructs a set of constraints on the fluents involved in its preconditions and effects. These constraints are then resolved using a weighted MAX-SAT solver to generate the action model with the highest weight. The Simultaneous Learning and Filtering (SLAF) **?** learns lifted action models from trajectories even if some of the states in them are not observed. SLAF uses logical inference to filter inconsistent action models, and requires observing all actions in the trajectory. FAMA **?** can also handle missing observations and outputs a lifted planning domain model. It frames the task of learning an action model as a planning problem, ensuring that the returned action model is consistent with the provided observations. NOLAM **?** can learn lifted action models even from noisy trajectories. LOCM **?** and LOCM2 **?**, analyze only the sequences of actions in the given trajectories, ignoring any information about the states between them. LatPlan **?** and ROSAME-I **?** are conceptually different since they learn propositional action models from trajectories where the states in the given trajectories are given as images, as opposed to a conjunction of fluents.

The Safe Action Model Learning (SAM) learning algorithms **??????** are a family of action model learning algorithms that return action models that guarantee that plans generated with them are applicable in the actual action model. An action model having this property is referred to as *safe action model*. The SAM family of algorithms addresses learning safe action models under different settings: the action models are lifted **?**, with stochastic **?** or conditional effects **?**, and even action models containing numeric preconditions and effects (**?**). **?** even extended their approach to support learning safe action models in a partially observable environment. However, none of the presented algorithms supports learning action models in a multi-agent setting.

To the best of our knowledge, the only algorithm for learning action models in a multi-agent setting is Lammas **?**, which closely resembles ARMS but outputs an MA-STRIPS action model. Lammas adds constraints that define the interactions between the agents. These constraints, combined with the correctness and action constraints defined in ARMS, provide a better understanding of the agents' actions, resulting in better performance than ARMS. However, like all other action model learning algorithms discussed earlier, Lammas learns from traces in which agents' actions are executed sequentially. As we demonstrate later, learning from trajectories with concurrently executed actions significantly increases the complexity of the problem. To our knowledge, the algorithms introduced in this work are the first to address the challenge of learning action models in a multi-agent setting where agents perform actions concurrently.

2.3 Review of classical action model learning evaluation metrics

Historically, the evaluation of learned action models has been primarily based on ***syntactic*** metrics. These metrics typically compare the intersection or difference of the lifted predicates in preconditions and effects between the learned and original models to quantify true positives (TP), false positives (FP), false negatives (FN) and true negatives (TN). Notable works employing this syntactic evaluation include FAMA and related algorithms such as [Omer: list learning algorithm/papers that used this comparison]. The metric is formally described as follows for an equivalently object-bound action a, a^* in the learned and original domains respectively:

- True Positives (TP): $|(pre(a) \cap pre(a^*)|$
- False Positives (FP): $|((pre(a) \setminus pre(a^*)|$
- False Negatives (FN): $|(pre(a^*) \setminus pre(a))|$
- True Negatives (TN): let L_a be all parameter bound literals for an action a. then $TN = |L_a \setminus (pre(a) \cup pre(a^*))|$

While straightforward and easy to compute, the syntactic metric often does not accurately reflect functional differences between learned and original models, particularly since certain predicates or negative conditions in one model might implicitly imply others, complicating direct comparisons and similarly there exists domains (rovers for example) that rely on add-delete of effects mechanism where it adds and deletes the same literal l in the same transition to simulate a continuous domain but it cannot be represented in the trajectory used as an input for the learning algorithms. Furthermore, another limitation arises when learned domain actions or predicates do not share identical signatures with those of the original domain, making direct syntactic comparisons even more challenging.

Semantic metrics address some of the limitations associated with syntactic evaluation by comparing the difference in post-state predicates after action execution and verifying action applicability in the pre-state for both the learned and original domains. This provides a more accurate and direct measure of functional correctness and applicability. Semantic evaluation is state- and action-specific, removing ambiguity and ensuring a precise assessment. The metric is formally described as follows:

Semantic applicability metric:

- True Positives (TP): Instances where a grounded action a is applicable from state s for both m and m^*.
- False Positives (FP): Instances where a grounded action a is applicable from state s for m but not for m^*, the original model.
- False Negatives (FN): Instances where a grounded action a is applicable from state s for m^*, the original model but not for m the learned model.
- True Negatives (TN): Instances where an action \$a\$ is not applicable in both the original and the learned domain pre-states.

Semantic effects difference metric: let s be the pre-state and s', s^* the states of the learned model and original model resulted by applying a grounded action a from state s and assume that for each state oof some grounded literal l is not in the state predicates then it's negation $\neg l$ is in the state.

- True Positives (TP): $|(s' \cap s^*) \setminus s|$
- False Positives (FP): $|(s' \setminus s^*) \setminus s|$
- False Negatives (FN): $|(s^* \setminus s') \setminus s|$
- True Negatives (TN): $|s \cap s' \cap s^*|$

Examples of works applying semantic metrics include [Omer: list papers] and recent variants focusing on detailed state-action analysis. Despite their advantages, semantic metrics face challenges when the signatures of actions, predicates, or object types differ significantly between the learned and original domains, complicating the identification of corresponding actions, predicates, or object types across domains. Additionally, the ***solving ratio*** metric offers a practical evaluation by determining TP, FP, FN, and TN based on planners' success in generating valid plans from learned versus original models. Specifically, a True Positive indicates that both models successfully produce a valid plan for a given problem; a False Positive occurs if only the learned model yields a solution; a False Negative is recorded when only the original model succeeds; and a True Negative when neither model generates a valid solution. Although highly reliable, this metric faces challenges in accurately representing state predicates and objects in the grounded domain to correspond precisely to the original domain's problem description. Another notable issue is reliance on planners that might exceed their time limits for certain complex problems. A possible solution is validating plans generated from one domain within the other, although this approach can be complicated when action signatures do not match.

3 Problem Definition

[Omer: TBD , i think it was discussed earlier. Maybe I need to restructure the paper.]

Input: - State representation - Action representation - Trajectories - Planner - Action model learning algorithm

How to compare performance of different algorithms?

4 A Paradigm for Comparing Action Model Learning Algorithms

Every action model learning algorithm can be viewed as a function that accepts a set of trajectories from the original domain and a learned domain. Therefore, we need to normalize the data of learned domains produced by different learning algorithms . Comparison of action model learning algorithms is challenging because different algorithms may output models with varied syntactic structures, representations, or levels of abstraction, making direct comparisons difficult. To mitigate this, we introduce an **encoder-decoder** mechanism: The **decoder** maps the components (i.e. object types, fluents, action naming, signature ordering of object and st.) in the learned model m back to the original domain language m^*. The **encoder** transforms the original domain m^* into the representation format of the learned model m. This process ensures that comparisons are done in a normalized format, reducing discrepancies caused by syntactic variations. Let us formally describe the encode and decode functions:

- $encodeAction : A_{m^*} \rightarrow P(A_m)$ and $encodeAction(s, a, m^*, m)$= $\{a' \in m | a' \textit{ is a proxy action of } a\}$
- $decodeAction : A_m \rightarrow A_{m^*}$ and $decodeAction(s, a', m^*, m) a^* \in A_{m^*}$ s.t. a^* is a proxy of a.
- $encodePlan : \{\pi_{m^*} \in \pi(A_{m^*})\} \rightarrow P(\pi_m \in \pi(A_m))$
- $decodePlan : \{\pi_m \in \pi(A_m)\} \rightarrow \{\pi_{m^*} \in \pi(A_{m^*})\}$
- $encodePredicates : P(P_{m^*}) \rightarrow P(P_m)$ and $encodePredicates(P' \subseteq P^*, m^*, m) = P'' \subseteq P_m$
- $decodePredicates : P(P_m) \rightarrow P(P_{m^*})$ and $decodePredicates(P' \subseteq P_{m^*}, m^*, m) = P'' \subseteq P_m$.

using these functions, we can now describe and use the earlier mentioned evaluation metrics to systematically compare the learned action model m with the ground truth model m^*.

semantic metric

True Positives (TP): The set of instances where for every pre-state s and action $a \in A_{m^*}$, if a is applicable in s, then at least one learned action $a \in encode(s, a, m^*, m)$ is applicable in s.

False Positives (FP): The set of instances where there exists a learned action $a \in A_m$ that is applicable in a pre-state s, but there is no corresponding action $a^* \in A_{m^*}$ $s.t.$ $a^* = decode(s, a', m^*, m)$ and a^* is applicable in s.

False Negatives (FN): The set of instances where for some pre-state s and action $a^* \in A_{m^*}$ $s.t.$ a^* is applicable in s, but no learned action $a \in encode(s, a, m^*, m)$ is applicable in s.

True Negatives (TN): The set of instances where for every pre-state s, if a proxy action $a \in A_m$ is not applicable in s, then there is no corresponding action $a* \in A_{m^*}$ such that a is applicable in s.

Action Applicability	Original Model m^*	Learned Model m
True Positive (TP)	Applicable	Applicable
False Positive (FP)	Not Applicable	Applicable
True Negative (TN)	Not Applicable	Not Applicable
False Negative (FN)	Applicable	Not Applicable

Table 1: Comparison of action applicability in the original and learned models.

These metrics allow us to quantify the *precision* and *recall*, where *precision* is $\frac{TP}{TP+FP}$ and *recall* is $\frac{TP}{TP+FN}$ This quantification will be useful when choosing the action model learning method based on restrictions and use cases of the learned model. These metrics provide a comprehensive evaluation of the learned model's ability to replicate the original model's behavior.

The *semantic* precision and recall of the preconditions and effects are computed with respect to a set of trajectories that we call the *test trajectories*. A reasonable choice for how to create these test trajectories is by running a planner on a set of test problems with the real action model. Alternatively, one may run a random walk with real action model, starting from random start states. To compute the defined above metrics, we compute the TP, TN, FP, and FN of every transition in the test trajectories.

solving ratio metric

syntactic metric

5 Case Study

How we implemented this paradigm on several algorithms

SAM: Encoder: identity Decoder: identity

FAMA - Leonardo L.?: Encoder: identity Decoder: identity - maybe change to parameter names?

ESAM: Encoder: identity for both state and actions Decoder: translate proxy actions to original actions

ROSAME: Encoder: encode to a numeric vector of ones and zeros Decoder: map action parameters

OBSERVER: Encoder: from binding to grounded Decoder: from grounded predicates and actions to the binding they represent isApplicable(state, groundedAction, model, encode) possible-grounded-action-list=[] if encode: possible-grounded-action-list = encode(groundedAction) else: possible-grounded-action-list = [groundedAction]

if any(isApplicable, possible-grounded-action-list) return true else false

compareApplicability(state, original-grounded-action, OriginalModel, LearnedModel)

get_possible_ecodings(real_action, model) /// return all actions in the model that my be decoded into real_action /// This is a helper function, that may or may not be provided

get_all_applicable(state, model) applicable_actions = [] for every action in model if pre(action) in state add action to applicable_actions return applicable_actions

IsApplicable(real_state, real_grounded_action, model) state = encode(real_state) for grounded_action in get_all_applicable(state, model) if decode(grounded_action)=real_grounded_action return True return False

Metric: TP,FP,FN,TN=0

for every (state,action) in data set If IsApplicable(state, grounded_action, learned_model) If IsApplicable(state, grounded_action, real_model) TP++ Else FP++ Else If IsApplicable(state, grounded_action, real_model) FN++ Else TN++

6 Benchmarks and Results

7 Conclusion Future Work