STAND: Data-Efficient and Self-Aware Precondition Induction for Interactive Task Learning

Daniel Weitekamp and Kenneth Koedinger

Carnegie Mellon University, Pittsburgh, PA 15213, USA

Abstract. STAND is a data-efficient and computationally efficient machine learning approach that produces better classification accuracy than popular approaches like XGBoost on small-data tabular classification problems like learning rule preconditions from interactive training. STAND accounts for a complete set of *good* candidate generalizations instead of selecting a single generalization by breaking ties randomly. STAND can use any greedy concept construction strategy, like decision tree learning or sequential covering, and build a structure that approximates a version space over disjunctive normal logical statements. Unlike candidate elimination approaches to version-space learning, STAND does not suffer from issues of version-space collapse from noisy data nor is it restricted to learning strictly conjunctive concepts. More importantly, STAND can produce a measure called *instance certainty* that can predict increases in holdout set performance and has high utility as an active-learning heuristic. Instance certainty enables STAND to be self-aware of its own learning: it knows when it learns and what example will help it learn the most. We illustrate that *instance certainty* has desirable properties that can help users select next training problems, and estimate when training is complete in applications where users interactively teach an AI a complex program.

Keywords: Data-Efficient Learning · Self-Aware Learning · Active Learning · Interactive Task Learning

1 Introduction

In ecology, a stand is a contiguous region of trees that share similar characteristics. An ecological stand is a habitat that is typically mutually beneficial to the trees that comprise it. Analogous to its namesake concept, STAND is a method for learning a compact collection of classifiers embedded in a shared data structure. The compact representation learned by STAND holds every classifier that would be generated by a randomized greedy learning process. In our explanation, we'll use decision trees as an example for this underlying classifier but the same method can be applied to other greedy strategies as well, including sequential covering methods common in inductive logic programming [13].

STAND is particularly helpful when solving classification and concept induction problems common in an interactive task learning [9] setting. Interactive task learning (ITL) is a vision for AI systems that can learn robust programs interactively from the natural instructions of untrained (typically non-programmer) users. Programming-by-demonstration [5] is a central sub-problem within ITL, enabling users to demonstrate the behavior of a program instead of programming or naturally articulating the program's structure. Programming-by-demonstration is a desirable input approach because it only requires users to show what their program should do, but not necessarily how those capabilities should be implemented. When authoring large multi-faceted applications a user demonstration may represent only one special case of a more flexible space of desired behaviors that may vary depending on the context of the program's initial input and its interanl states. In these cases, it is helpful

for ITL systems to induce rule-based representation languages like production rules or hierarchical task networks (HTNs) [6], where individual rules or methods can generalize to serve broader roles beyond their appearance in users' initial demonstrated action sequences. In these cases, ITL systems must often induce preconditions for rules and methods to control how and when they are executed within a complex program.

We promote STAND as being particulary helpful for the problem of learning rule preconditions in an ITL setting from positive and negative examples of rule execution. This variety of supervised inductive precondition learning envisions that correctness labels are collected from users interactively as they verifying the induced behavior of the ITL agent. This approach has been taken, for instance, in authoring-by-tutoring systems [10, 14] where authors build educational technology by naturally tutoring an AI agent, initially with step-by-step demonstrations, and later by grading its problem-solving attempts. These kinds of precondition induction tasks typically involve learning conditions over noiseless features, and over data that is typically small, poorly balanced, and has a low sample-to-feature ratio, because the data was collected from interactive instruction. As we argue in the following sections STAND excels in these particular circumstances and outperforms methods like XGBoost that typically boast the highest accuracy at classification tasks over tabular features.

Moreover, STAND can directly support users in interactively training AI agents because it is self-aware of its learning. STAND produces a measure on unseen examples called *instance certainty* that can accurately estimate when new training examples produce increases in holdout set performance, and which has high active-learning utility in terms of identifying unlabelled examples that will help STAND learn the most. These two features can be used to aid users in picking good next training examples, and for estimating when training is complete—when the ITL system's induced program is correct and complete.

2 Overcoming the Limits of Supervised Learning by Embracing Ambiguity

Traditional supervised machine learning frames the problem of learning generalizations that perform well beyond a training set as a matter of mitigating overfitting and underfitting. In a small-data setting the properties of the distribution from which the data was sampled cannot be estimated precisely and thus overfitting becomes especially problematic. Overfitting is typically mitigated by methods of regularization that constrain how specialized or complex models are allowed to become when fit to training data. Regularization methods often sacrifice some training data performance to maximize performance on unseen data.

This framing of supervised learning where a theoretically optimal, yet fundamentally imperfect predictor is fit to noisy data works well when data is numerous and when there is some stochasticity in features or sample labels. However, in an interactive task learning (ITL) setting [9], data is small because it is generated interactively from a single user, and the aim is to enable the user to automate intended behaviors through various forms of instruction. In this context, the user may reasonably expect that they can produce a program that reliably works in every situation. In principle, this should be possible since in many cases we can expect that the features available within the task are noiseless and that the program that the user intends to produce has consistent well-defined behavior.

The best practices of traditional supervised learning do not translate well to ITL tasks. For instance, it is rarely feasible to collect enough well-balanced data that one could reliably model variations in a user's generated data as samples of a common distribution, without permitting major biases. For instance, training data generated from user's feedback on an agent's proposed actions may accumulate far more positive examples than negative examples as the agent improves in performance. Prior work along these lines [14] has reported that users typically only accidentally produce important edge cases, and rarely produce them as a result of employing good teaching strategies. Thus, important edge cases may be rarely covered during training, if at all. Many data-driven approaches that rely heavily on the relative frequencies at which patterns occur in data would perform poorly on this small, poorly balanced, and sparse data. For instance, data-hungry deep-learning methods are likely useless in this context.

Additionally, many of the tricks for reducing the overfitting of symbolic learning mechanisms, like subsampling and pruning for tree classifiers, are irrelevant by their very principle. After all, if one has set out to find conditions that are 100% accurate at separating correct and incorrect applications of rule for non-stochastic programs, then the training set must have 100% accuracy as well. No sacrifice in performance on the training set can be made in an effort to optimally generalize beyond it since one is always aiming for, and expect to achieve perfect performance. Of course, this expectation assumes that users catch all of their mistakes, which is not something that can be guaranteed but can certainly be supported with good interaction design choices, like affordances for reviewing past training interactions.

In a small-data setting fitting a classifier that has perfect performance on the training set is often so easy that any generated solution will be almost arbitrary. There are typically more perfect solutions than are feasibly generatable or even countable, and we can at best employ classifiers with inductive biases that select among the more parsimonious ones. In precondition learning finding a set of conditions that achieves 100% accuracy in all unseen cases is like finding a secret special needle in a pile of needles—we would not know if we happened to select the correct one even if it was right in front of us, and any attempt to guess would just be an arbitrary choice.

The key to solving this problem is to abandon the idea that we can rely upon statistical learning at all, and embrace that among the classifiers that could be learned from limited data, the choice of the right one is an ambiguous one. Each user interaction provides evidence of a program that the user intends to demonstrate to the AI agent. This intended program remains ambiguous until the user provides sufficient evidence to disambiguate the intended program from other possibilities. To support this perspective we can at best try to map out all of the programs that the user may be intending to demonstrate and cut out large swaths of possibilities as new evidence proves possibilities to be impossible. Approaching learning in this way allows us to do a great deal more with limited data, both because we are no longer arbitrarily guessing at solutions from limited evidence and because knowing all of the possibilities means we can support users in making decisions that can maximally reduce the ambiguity of selecting the right generalization.

The notion of learning by updating possible spaces of classifiers is not a new one. The theory of version spaces [11,12] dates back to the early days of machine learning, and describes a general theory for efficiently representing spaces of all generalizations consistent with a set of examples. For a particular representation language, a version space bounds the set of all

consistent examples with two boundary sets. Version spaces' boundary sets can enclose a space of consistent generalizations far larger than can be feasibly enumerated and provide a means of updating the space to cut out large sets of possible generalizations as they prove to be inconsistent with new training examples. Version spaces require a definition of relative generality between generalization hypotheses. For instance, in the generalization language of conjunctive logical statements, a statement consisting of one predicate $X_5 = True$, is more general than a conjunction of two predicates $AND(X_5 == True, X_3 == False)$. The two boundary sets consist of a maximally general set G, and a maximally specific set S. The set of all hypotheses contained in the version space is the set of hypotheses in S or more general than S, but no more general than any hypothesis in G.

The seminal work on version spaces [11] describes them as "generalization as search". Although "search" is a somewhat misleading analogy. As the terms *searching* and *planning* are most commonly used, they evoke a process of looking through individual possibilities in pursuit of a known target, like finding a needle in a haystack. Version spaces do something far more powerful. They cut out vast sets of possibilities as new evidence proves them to be impossible. They systematically cut out possibilities in pursuit of the particular good needle in the needle stack, throwing out large sets of generalizations that are inconsistent with new examples. A version space may narrow down to a single possible generalization when its general set G and specific set S converge to a single generalization. However in practice, when data is limited, convergence to a single generalization cannot be guaranteed or even expected. This is not a weakness of version spaces, so much as a realistic treatment of the ambiguity of inducing correct generalizations from limited data.

One limitation of version spaces is that they are typically impractical or by some characterizations [7] even intractable to learn over disjunctive normal logical statements. A disjunctive normal logical statement is one that disjoins two or more conjunctions with an OR. For instance:

$$OR(AND(X_5 == True, X_3 == False), AND(X_1 == "1", X_2 == True)) \tag{1}$$

The classic candidate elimination approach to version-space learning is limited to learning just conjunctive statements.

STAND can learn a version-space-like structure with G and S boundary sets, that approximates a version space over this known intractable representation language. Unlike typical algorithms for learning version spaces, STAND's approximate version space learning does not fail catastrophically when it encounters noisy or mislabelled data; it does not suffer from version space collapse. More importantly, as I will demonstrate in the following sections, STAND is much more useful than a single classifier or even an ensemble of classifiers, both because it tends to achieve higher performance from less data, and because it can produce very reliable estimates of its learning progress that are useful for users in an ITL setting.

Symbolic classification methods that learn a single set of conditions can only accept or reject a new unlabeled example, but STAND can learn an approximate version space of possible condition sets that produce competing predictions about an example's correctness. Much like an ensemble, this allows STAND to estimate the certainty of its predictions based on a set of competing hypotheses. While many statistical machine-learning methods can make continuous probabilistic predictions about the class labels of unseen examples, they

are often too data-inefficient to be helpful in an ITL setting. STAND by contrast requires remarkably little data to accurately estimate the certainty of its predictions. Additionally, STAND's estimates of prediction certainty have a structurally meaningful counterfactual interpretation that differs from typical statistical estimates of class probability. If an example is accepted by a part of STAND's approximate version space but rejected by another, then one of the two parts will be eliminated when the label is revealed. This allows STAND to quantify how much of its approximate version space will change as a result of receiving feedback from a user. STAND can essentially predict which training examples will help it learn, and which will not.

3 STAND: Building A Complete Space of Classifiers for the Cost of One

In ecology, a stand is a contiguous region of trees that share similar characteristics. An ecological stand is a habitat that is typically mutually beneficial to the trees that comprise it. Analogous to its namesake concept, STAND is a method for learning a compact collection of classifiers embedded in a shared data structure. The compact representation learned by STAND holds every classifier that would be generated by a randomized greedy learning process. In our explanation, we'll use decision trees as an example for this underlying classifier but the same method can be applied to other greedy strategies as well, including sequential covering methods common in inductive logic programming [13].

To efficiently produce every classifier that would be produced by a greedy learning process, STAND expands every decision whenever an arbitrary choice would be made to break ties between nearly equally good options. For instance, when applied over decision trees, STAND is structurally similar to an option-tree [8]: a variation of decision trees where each node splits every feature with the highest split criterion reduction simultaneously, instead of splitting on just one (often randomly selected) best feature. Like an option-tree, instead of only expanding one feature that optimally splits data at each node, STAND expands every split that would decrease the impurity criterion nearly as well as the best split $\Delta C_{imp}(X_i) >= \alpha \Delta C_{imp}(X_*)$; where the parameter $\alpha \in (0, 1]$ varies the rejection rate for splits relative to the best split. Choosing $\alpha = 1.0$ to only accept splits with utility equal to the best split often works well for small non-noisy datasets. Like an option-tree, each node in STAND has $2n$ edges each leading to child nodes, where n is the number of best splits for that node. By contrast, normal decision trees typically have strictly 2 child nodes per non-terminal node.

While regular option-trees are unwieldy or intractable to compute without imposing constraints, such as limits on node depth or the number of expansions at each node, STAND can efficiently generate a complete compressed option-tree-like structure by caching nodes by the set of subsets of samples that they select. Since the set of expanded splits at each node depends entirely on the training samples selected by that node, we can route all edges that select the same subset of samples to the same shared node. Reusing nodes in this manner allows STAND to learn a complete space of possible decision trees, often in only a little more time than it takes to learn a single decision tree. This trick also makes it easy to support partial incremental learning over streams of examples. Highly ranked splits tend to remain highly ranked when new examples arrive, so when new examples are filtered into the tree only those nodes with changes in their set of best splits need to be refit.

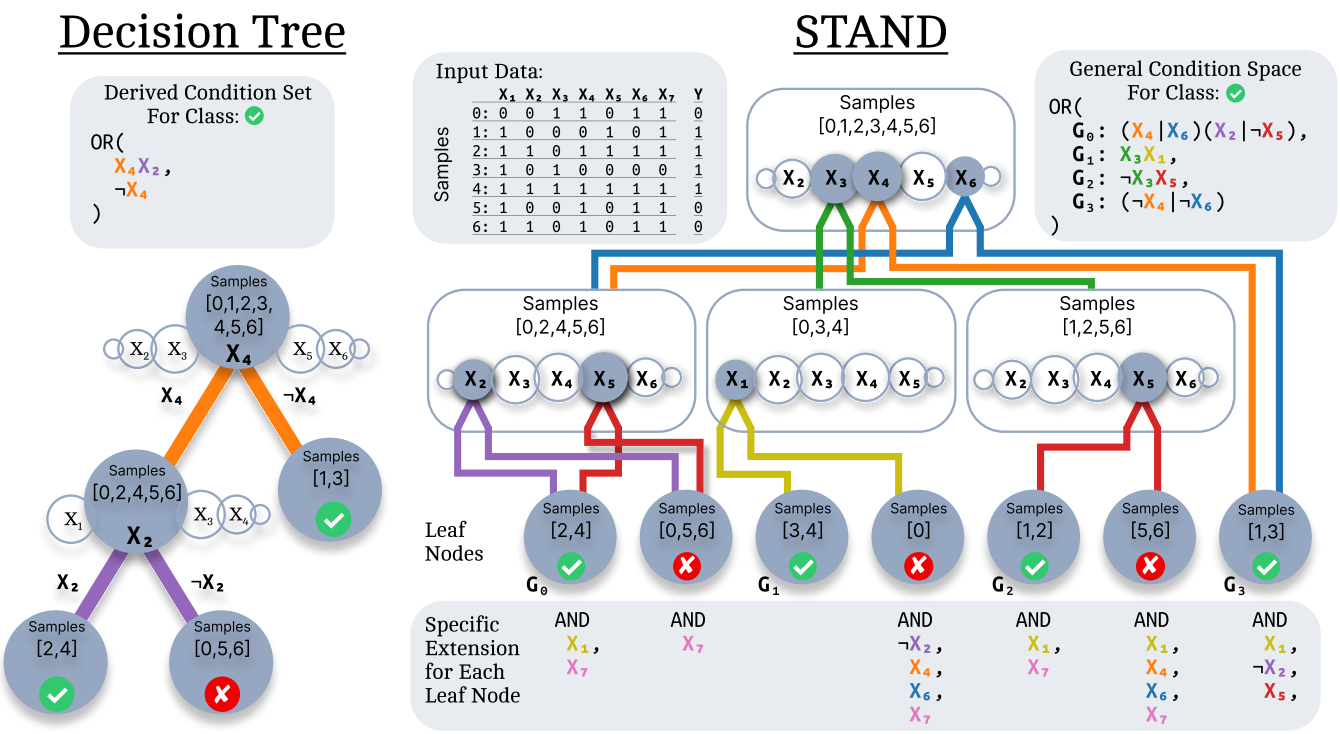

Fig. 1: An example of a Decision Tree and STAND fit to the same input data. In STAND multiple splits (filled grey circles) are expanded per node. STAND builds a general condition space (top-right) that is bounded below by a specific extension (bottom).

Figure 1 below shows an example of a decision tree and STAND fit on the same data. Each sample 0-6 in the training data has seven binary features $X_1, ..., X_7$. In the decision tree, X_4 is selected randomly for the root node from among the best features for splitting the data. $\neg X_4$ (i.e. $X_4 = 0$) selects a pure subset of two positive samples and X_4 (i.e. $X_4 = 1$) selects an impure subset that is then split further by X_2 into leaves with purely positive or negatively samples. By contrast, STAND splits X_4 at the root, but also X_3 and X_6 as well. STAND's sample caching trick makes it so that the 6 edges formed by these 3 splits only lead to 4 nodes (two of which are reused by two edges) instead of 6 nodes (one per edge) like in a normal option-tree. The 4 nodes downstream of the root are reached by following edges that select sample sets [0,2,4,5,6], [0,3,4], [1,2,5,6], and [1,3] respectively. The last one is a leaf because it only selects positive samples, while the others are still impure, and are further split into pure leaves. Note that unlike a typical decision tree, in STAND a single sample can filter into multiple leaves. For instance, the 3rd leaf [3,4] and last leaf [1,3] in Figure 1 both contain sample 3.

From a conventional decision tree, one can derive a disjunctive normal logical statement that only selects training samples of a particular class. If all leaves are pure, which can be expected for non-stochastic data, then edges along paths from the root of a decision tree to its positive leaves form conjunctions of literals that each only select positive training

examples. For instance, in Figure 1 the decision tree's derived statement for positive samples is $OR(X_4X_2, \neg X_4)$. In STAND, multiple edges can lead into the same node, meaning there are multiple paths of literal sequences (i.e. conjuncts) that select the same sub-samples. In the top-right of Figure 1 we represent these options in parentheses separated by the | symbol. For instance, the choice of literals $(X_4|X_6)$ corresponds to the two edges leading into the left-most node that selects [0,2,4,5,6]. STAND's caching trick helps account for alternative conjuncts that select the same subsets of training samples, and thus it is not just an optimization, but also a means of compressing sets of alternative generalizations. For instance, the left-most leaf in Figure 1 is reached by any conjunct in the Cartesian product of options represented by $\mathcal{G}_0$:

$$\mathcal{G}_0 = (X_4|X_6)(X_2|\neg X_5) = X_4X_2 \mid X_6X_2 \mid X_4\neg X_5 \mid X_6\neg X_5 \tag{2}$$

It is important to keep in mind that STAND's node caching trick is only helpful in limited circumstances. For instance, it would likely not be effective in many data-driven prediction settings where classifiers are learned over large datasets with noisy features or labels. In these cases, there would be very little consistency between the subsets of samples selected by different splits leading to limited potential for node reuse. Large training sets would also make subset hashing and comparison computationally expensive. However, this approach thrives in non-stochastic small-data environments, including many ITL applications. This includes precondition induction like when-learning where the target generalization is a set of hard requirements and not a probabilistic predictor. Preconditions after all must be expressible in a representation language of non-stochastic predicate-like features. It is admissible for stochastic features to be present in the dataset so long as they are not necessary for discriminating the target preconditions.

3.1 A Space of Classifiers for the Cost of One

Any AI approach used in an interactive setting should execute quickly to avoid subjecting users to considerable lag. In this section, we report STAND's average fit and predict times as they are used in an ITL tool for building educational programs. In this tool STAND's `.fit()` and `.predict()` sub-routines may be called tens to hundreds of times as a side effect of individual user interactions. Thus, keeping these subrountines to fewer than about 10 milliseconds is important for seamless usability.

In this context, STAND provides a great deal of benefits with almost no efficiency drawback. On average fitting and predicting with STAND takes only marginally more time than fitting and predicting with a single decision tree. Averaging over a long 100 problem training sequence refitting STAND takes about 5.30 ms whereas a single decision tree takes about 4.42 ms. On average for predicting the correctness of a new action STAND takes 0.35 ms and a decision tree takes 0.27 ms. With either of these precondition learning approaches the total time it takes to re-fit several classifiers and make several new predictions to refresh the display rarely exceeds 300ms. Replacing these classifiers with ensemble methods like random forests or XG Boost leads to update times of a second or more—long enough to be noticeable to users or even be disruptive to their training process.

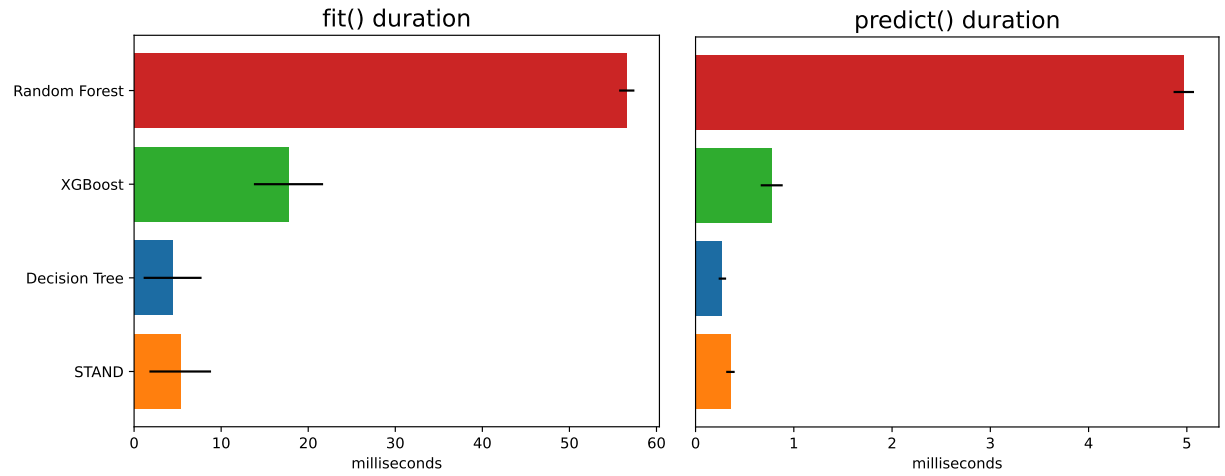

Fig. 2: Average `.fit()` and `.predict()` durations for Random Forest with 100 estimators, XG Boost, Decision Tree and STAND used as when-learning mechanisms for agents trained on 100 multi-column addition problems. Timed on an Ubuntu 22 laptop with an 11th generation Intel i7-1165G7 processor and 16 GB of RAM.

STAND's caching trick is a big part of why it is nearly as efficient as a single decision tree. Typically decision trees calculate the utility of every possible way of splitting each node's sub-samples. STAND simply hashes the indices of the two child subsets generated by each split and routes equivalent child subsets to the same node. When data is small and mostly noiseless STAND's total number of nodes is not more than a small factor greater than a normal decision tree's. Very little extra work is performed per node since each node only needs to calculate its split utilities once.

3.2 STAND as an Approximate Version Space

STAND's compressed option-tree-like structure of cached nodes is akin to the general set G of a version space. Since a true G over disjunctive normal logical statements is intractable to compute, it goes without saying that STAND's structure is only an approximation—a strict subset of a true general set G. Nonetheless, this structure shares important properties of a true version space's general set G:

1. The members of G are not any more specific than they need to be.
2. The members of G cover all of the most-general possibilities (by some definition).

The first property depends on the choice of greedy classifier underlying STAND but is certainly true of decision trees and sequential covering. These processes construct conditions one split or literal at a time and do not grow generalizations any more than necessary to separate training examples by label. The second property follows from the fact that STAND expands all options that could be randomly constructed by repeatedly rerunning one of these greedy construction processes. This is of course a much looser definition than the typical theoretical notion of a G set which includes all of the most-general consistent generalizations expressible within a representation language. Nonetheless, STAND's approximate G is useful in practice because it spans a well-defined space of *good* choices within a disjunctive representation language.

STAND's general set G is encoded in a distributed manner over its leaf nodes. Examples filter into one or more leaves because they satisfy one or more of the alternative literal statements associated with each of the ancestor nodes along some path leading from the root. The paths leading to each leaf node i form a set of alternative conjunctive statements $\mathcal{G}_i$ that select all of the training samples associated with that leaf node. As in the example above, the set of all conjunctions in $\mathcal{G}_i$ is the Cartesian product of each alternative. For instance, in Figure 1, leaf $i = 0$ forms a space of alternative conjunctive statements $\mathcal{G}_0$:

$$\mathcal{G}_0 = (X_4|X_6)(X_2|\neg X_5) = X_4X_2 \mid X_6X_2 \mid X_4\neg X_5 \mid X_6\neg X_5 \tag{3}$$

where $|$ represents an alternative choice. Let $\mathcal{L}_{cov}$ be the set of all minimal subsets of leaves that cover the positive training examples. For each covering set $L_{cov} \in \mathcal{L}_{cov}$ a portion of the general set $G_{L_{cov}}$ is formed by disjoining every combination of alternative conjunctions for all $i \in L_{cov}$. The set of disjunctive statements generated by L_{cov} is then:

$$G_{L_{cov}} = \{g_1 \vee \ldots \vee g_n | (g_1, \ldots, g_n) \in \mathcal{G}_0 \times \ldots \times \mathcal{G}_n\} \tag{4}$$

Where $\vee$ indicates disjunction and $\times$ indicates the cartesian product of all $\mathcal{G}_i$ associated with L_{cov}. The total set of most general disjunctive statements covered by STAND's effective G set is then simply:

$$G = \bigcup_{L_{cov} \in \mathcal{L}_{cov}} G_{L_{cov}} \tag{5}$$

STAND's specific set S is formed by extending the generalizations associated with each leaf node i so that they select any additional features that are common between the samples selected by each leaf. The *specific extension* s_i for each leaf i is a conjunction of literals selecting all common features in the leaf's subset of samples that do not overlap with any of the literals comprising $\mathcal{G}_i$. Each pair of $\mathcal{G}_i$ and s_i define a mini-version space of conjunctive statements within the whole. Any conjunction in $\mathcal{G}_i$ can be extended by adding literals from s_i to form a new conjunction that selects all of the training samples that filter into leaf i.

The classic candidate-elimination approach for learning conjunctive version spaces [11] can fail catastrophically when it encounters examples that are logically inconsistent with its enclosed generalizations. This is called version space collapse, and it makes traditional version space approaches brittle to training on noisy data. STAND does not suffer from this issue. It is as robust to noise as whatever greedy algorithm it is applied to. For instance, when applied to decision trees, as we have described above, a mislabelled example can prevent STAND from converging to perfect performance, but it will not cause STAND to break entirely. In an ITL setting, if a user mislabels the correctness of an example, then STAND will very likely introduce new disjunctions into its tree structure. In the best case the mislabelled example may filter into its own leaf isolated from the rest, which would produce minimal changes to model behavior. Or in the worst case it may filter into a leaf that captures several other properly labelled examples, which could alter the literals that select those examples, or cause the specific extension s_i for the leaf to over-generalize.

4 Estimating Model Ambiguity and Instance Certainty

Interpreting STAND as an approximate version space allows us to quantify various notions of ambiguity and certainty. We define *model ambiguity* as the size of the approximate version space. It captures how ambiguous the target generalization remains given all of the generalizations that are equally consistent with the current training data. *Instance certainty* captures how unambiguous the label prediction of an example is given all of the generalizations that capture the example. Low instance certainty indicates high disagreement between the predictions of alternative generalizations in a STAND model's version space.

Model ambiguity is loosely analogous to the inverse of the posterior distribution $P(\theta|X)$ of a Bayesian statistical model and instance certainty is loosely analogous to the posterior predictive distribution $P(y|x, X, Y)$. However, these probabilistic concepts are imperfect analogs since STAND is not nearly as sensitive to the distributional properties of data as a typical parameterized statistical model. Each element in STAND's space of generalizations is equally consistent with the training data, and possesses no notion of relative likelihood between them, nor do generalizations have parameters derived from the frequencies of patterns in the data. Consequently, STAND does not need to be trained on large sets of independent and identically distributed examples. STAND requires diverse examples to learn well, but not necessarily numerous or well-distributed ones.

4.1 Model Ambiguity

In practice, the true values of model ambiguity and instance certainty are prohibitively expensive to compute since they require generating G from all minimal spanning sets $\mathcal{L}_{cov}$. This runs the risk of combinatorial explosion. For practical purposes, it is more helpful to use heuristics that change throughout training in a manner reflective of changes in model ambiguity and instance certainty. As a simplification, we can calculate a heuristic A for the total model ambiguity by summing a heuristic A_i representing the size of each leaf's independent mini version space. The true total size of each leaf i's mini-version-space is on the order of:

$$size_i = (\prod_{g_{ji} \in \mathcal{G}_i} |g_{ji}|)(1 + |s_i|)! \tag{6}$$

Estimating this size precisely is not particularly useful, since the magnitude of $size_i$ is highly sensitive to the size of the specific extension s_i, leading to a number that can vary wildly between leaves. It is far more useful to simply sum the number of literals in $\mathcal{G}_i$ and s_i to make for an easy-to-compute, numerically stable heuristic that reflects immediate changes to the boundary sets G and S. We'll define A_i and A to be:

$$A_i = (\sum_{g_{ji} \in \mathcal{G}_i} |g_{ji}|) + |s_i| \tag{7}$$

$$A = \sum_i A_i \tag{8}$$

In the above, $|g_{ji}|$ represents the number of alternative choices of literals in ancestor node j of leaf i. Because of node caching each node may have multiple parents, and thus finding all $|g_{ji}|$ for leaf i involves traversing several branching possibilities back to the root.

4.2 Instance Certainty

To compute instance certainty we must consider how examples may be filtered into several leaves that disagree in label prediction. We must consider three varieties of disagreement that may occur when an example is compared to STAND's approximate version space.

First, an unlabelled example can simultaneously filter into multiple positive leaves and multiple negative leaves. This disagreement is similar to the prediction disagreement between classifiers within an ensemble.

Second, within each of the leaves that accept the example, if the true correctness label agrees with the leaf's label, the mini-version-space formed by $\mathcal{G}_i$ and s_i of leaf i may reduce in size to accommodate the new example. Subsets of the literals in each $g_{ji} \in \mathcal{G}_i$ and literals of the specific extension s_i may be inconsistent with the new example and thus will be dropped. This will reduce the total heuristic size A_i of leaf i's mini-version-space.

Third, if a user's stated correctness label disagrees with the leaf's label then refitting the option-tree structure will result in extensions or rearrangements of nodes in order to achieve purity in all leaves. These sorts of changes are largely unpredictable without speculatively refitting with alternative example labels and are likely to effectively increase the size of A. As these sorts of changes are difficult to characterize analytically it is better to ignore them. Part of the curse of trying to approximate a version-space over disjunctive concepts is that the space must grow as it entertains new disjunctions. Consequently, A is unlikely a useful heuristic of learning progress on its own. However, instance certainty is still useful if we only focus on how new examples may reduce the size of each known A_i.

Given these considerations, we can calculate the certainty that a new example x belongs to the positive class or negative class independently. For a set of positive leaves $L_+(x)$ that accept an unlabelled example x we can find the average disagreement of their mini-version spaces. If A'_i is the value of A_i after shrinking from accommodating the new example then the proportion of literals bounding i's mini-version-space that accepts the example is A'_i/A_i. Averaging over each leaf in $L_+(x)$ we get:

$$IC(x)_+ = \frac{1}{|L_+(x)|} \sum_{i \in L_+(x)} (A'_i/A_i) = \frac{1}{|L_+(x)|} \sum_{i \in L_+(x)} \frac{(\sum_{g'_{ji} \in \mathcal{G}'_i} |g'_{ji}|) + |s'_i|}{(\sum_{g_{ji} \in \mathcal{G}_i} |g_{ji}|) + |s_i|} \tag{9}$$

If we also compute $IC(x)_-$ from the negative leaves that accept x we can define $IC(x)$ as a value ranging from -100% to 100% that can be easily placed within an interface and interpreted by a user.

$$IC(x) = \begin{cases} IC(x)_+ & \text{if } IC(x)_+ \geq IC(x)_- \\ IC(x)_- & \text{otherwise} \end{cases} \tag{10}$$

Compared to other measures of model probability, instance certainty is particularly informative to a user in an ITL setting since it captures prediction certainty, example-by-example learning utility, and indirectly indicates learning completion when $IC(x) = 100\%$. Many

statistical models can produce continuous probability predictions from the contributions of either a single classifier's internal weights or the competing outcomes of multiple models in an ensemble. By contrast, $IC(x)$ accounts for all of the predictions of alternative consistent generalizations within STAND's version space. Relative to STAND, most methods for estimating prediction probabilities do not rely on a particularly complete account of alternative predictors. Bayesian estimations of posterior predictive distributions $P(y|x, X, Y)$ by integration over a posterior $P(\theta|X, Y)$ are a notable exception, although in practice these measures are hard to compute, and lack some of $IC(x)$'s desirable properties as a helpful signal to users.

For instance, when $IC(x) = 100\%$ on an unseen example x that means that STAND's version space encloses no positively labeled generalizations that reject x, meaning that nothing can be learned by verifying that x is correct. $IC(x) = 100\%$ may be a false positive if the users' examples are very similar to one another or if they have not yet provided many negative examples.

5 Evaluating Learning Performance and Certainty Estimation Quality

To assess STAND's capabilities as a when-learning mechanism in an authoring setting, we evaluate its performance against several alternative when-learning mechanisms in two tutoring system domains. We evaluate each model on its overall predictive performance per problem, and performance on measures of prediction quality.

5.1 Methods

Using an authoring-by-tutoring system for ITL-based authoring of educational technology we train several agents with several competing methods of precondition learning. We utilize a system similar to the one reported by Weitekamp et. al [14], and use an automated training system that mimics the demonstrations and feedback that an ideal user would provide while authoring. We apply this special *authoring* training approach in the two domains: multicolumn addition and fraction arithmetic.

In this setup, each agent receives ideal on-demand demonstrations and correctness feedback. At each state, all of the agent's proposed actions are given correctness feedback. If an action is missing then it is demonstrated to the agent with annotations that make the underlying reason for the action unambiguous. Each demo is annotated with the formula for producing the action's value, and the arguments used. These additional demonstration annotations support mechanisms for inducing the then-part of each rule's if-then structure (readers should refer to prior work [14] for a description of those mechanisms), and this essentially makes it so that all errors in our simulations can be attributed to the various precondition learning approaches that we employ.

We compare several classifiers for precondition learning with STAND:

1. **Decision Tree**: A decision tree using gini impurity [3] as the impurity criterion. We use STAND's implementation, expanding just one random split at each decision point.

2. **Random Forest**: Scikit-learn implementation of random forest ensemble [2] of 100 decision trees. Random forests use bagging [1] to independently train several decision trees on subsets of the data.
3. **XG Boost**: An ensemble method that trains multiple decision trees one at a time. This method uses gradient-based sampling to re-weight the samples for subsequent trees [4].

These tree-based methods are chosen because they excel at learning from small datasets of structured data. In all models no limits are set on tree depth or leaf size since for these condition-learning tasks the agents are provided with features that are sufficient for separating correct and incorrect candidate skill applications perfectly. Since condition learning is noiseless the trees will already tend to not become more complex than the ideal solution, and limiting their depth could only prevent the ideal solution from being discovered.

The two ensemble methods are included for comparison with STAND's comprehensive version-space-based approach, and to compare the utility of their prediction probabilities with instance certainty $IC(x)$. Each model is re-trained on 40 repetitions on a sequence of 100 randomly generated problems. However, in the active-learning conditions described below, agents self-select their next training problems. After each completed training problem each agent is evaluated on a holdout set of 100 problems. All agents are evaluated on the same holdout set for each domain.

In the active learning conditions agents assign a certainty score to each problem in the random problem pool. For each candidate next problem the agent rolls out every sequence of actions that it predicts to be correct along every possible diverging solution path. For each problem each action produced along this rollout is given a certainty value—$IC(x)$ for STAND and prediction probability for the other models. The certainty score for a problem is the minimum certainty value assigned to all actions produced in this rollout that are predicted to be correct. The minimum is used instead of the average so that problem selection is not biased by the number of problem steps. After each training problem, the problem in the pool with the lowest certainty value is selected as the next problem. Then the selected problem and the highest certainty 50% of problems are replaced with randomly regenerated problems. This resampling ensures that the pool tends to contain a high proportion of uncertain problems.

5.2 Evaluating Prediction Performance and Stability

STAND's raw predictive performance compared to alternative methods is only one element of evaluating its usefulness in an interactive task learning setting. Ideal precondition learning should rapidly converge to a state of 100% holdout set accuracy and alter its prediction behavior conservatively—changing predictions only when new examples provide evidence to suggest the change. Methods like decision trees that randomly pick among alternative choices when fitting can produce an issue where predictions on unlabeled examples change dramatically between training events. In an ITL setting, users may find that the actions suggested at each problem state change spontaneously to include new incorrect actions or exclude correct ones. Thus, in addition to raw model performance, it is important to evaluate the stability of the precondition learning algorithm's predictions. With these considerations in mind we evaluate STAND's prediction behavior to the comparison models on per-problem completeness, per-problem errors by type, and error re-occurrence rate.

Per-Problem Completeness Below we report each model's completeness performance on a holdout set of 100 problems after each training problem. We define completeness here as "model-tracing completeness" [14]: the average number of problem states along all correct solution paths where the agent would only suggest every correct action and no incorrect actions. Completeness reflects the proportion of problem states in the holdout set where when-learning produces 100% correct predictions.

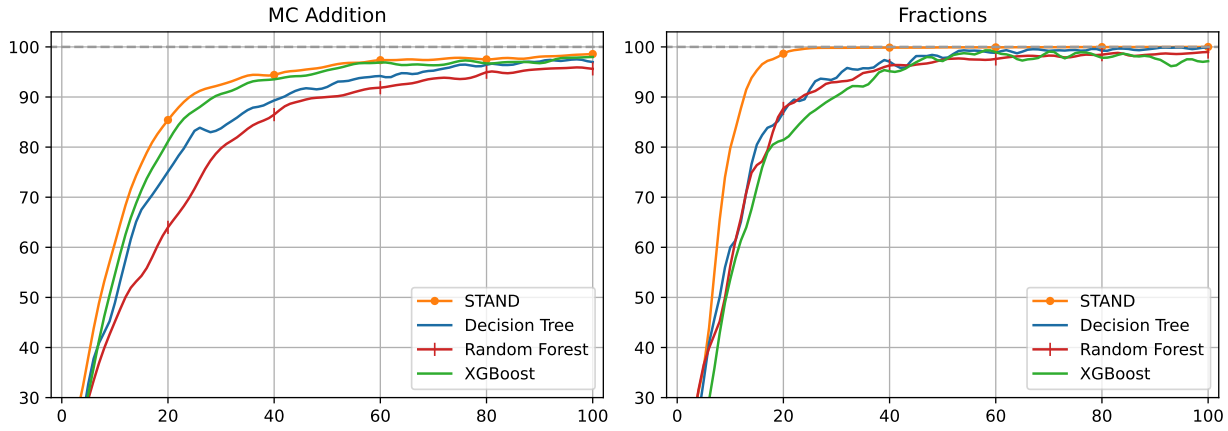

Fig. 3: Average holdout completeness by problem.

Table 1: Average Holdout Completeness at Problem N, and Number of 100% Complete Repetitions at problem 100.

	MC Addition				Fractions			
	N=20	N=50	N=100	100% Reps	N=20	N=50	N=100	100% Reps
STAND	**85.45%**	**96.10%**	**98.62%**	**19/40**	**98.72%**	**99.91%**	**99.99%**	38/40
Decision Tree	75.75%	91.86%	96.97%	10/40	88.15%	97.24%	99.88%	38/40
Random Forest	64.16%	90.02%	95.53%	0/40	88.13%	97.44%	98.97%	11/40
XG Boost	81.20%	95.40%	98.01%	3/40	81.12%	96.20%	97.34%	27/40

In both domains, STAND's average completeness is higher than the competing models throughout the training sequence. This implies that STAND has better overall model performance, and data efficiency since it can achieve greater levels of completeness with fewer training problems. In 19 of 40 multicolumn addition repetitions STAND achieved 100% completeness after training on a sequence of 100 problems compared to 10 of 40 repetitions for decision trees. In fractions, 38 of 40 repetitions achieved 100% completeness with STAND and decision trees. The relative performance of the decision tree, random forest, and XG Boost varies between domains. Notably the random forest was the worst in multicolumn addition, likely because its bagging approach of sampling subsets of the data had the effect of dropping important edge cases, which are particularly important in this domain.

The relatively poor performance of random forests highlights that models that tend to work well in a data-driven machine learning setting do not necessarily work well in an interactive task learning setting. When fitting an imperfect predictor to a large noisy dataset fitting on sub-samples can create helpful diversity in an ensemble. In our case however fitting

on sub-samples most likely discarded important edge cases that could have provided valuable evidence about the true preconditions, thus many of the tree instances in the random forest most likely underfit the data.

5.3 Per-problem Errors by Type

A when-learning mechanism can make errors of omission where a correct action is considered incorrect (i.e. a false negative), and errors of commission where an incorrect action is predicted to be correct (i.e. a false positive). Errors of commission are easy for users to fix and hard to miss: the user must simply mark a proposed action as incorrect. Errors of omission are harder to notice, and take slightly longer to fix: the user must demonstrate a correct action missing from the set of proposed actions.

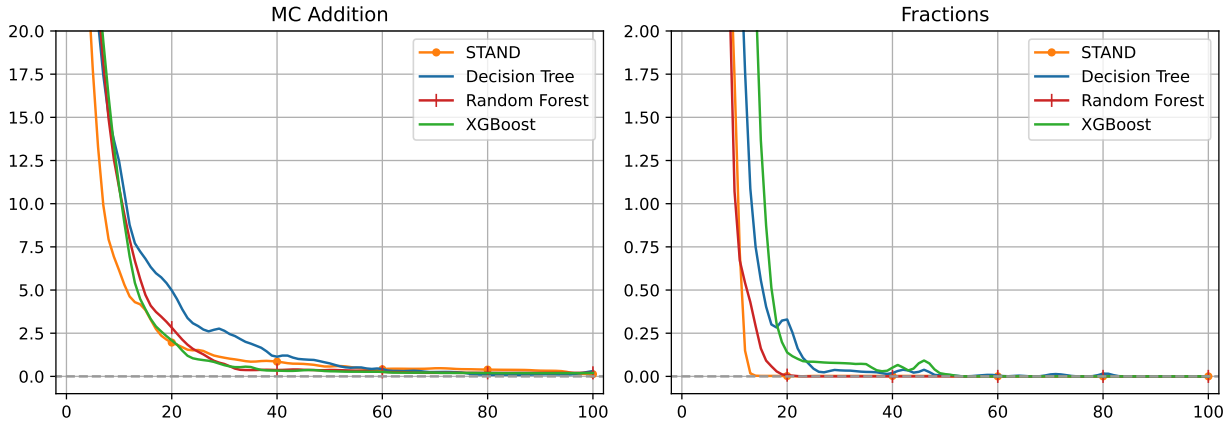

Fig. 4: Average omission errors by problem.

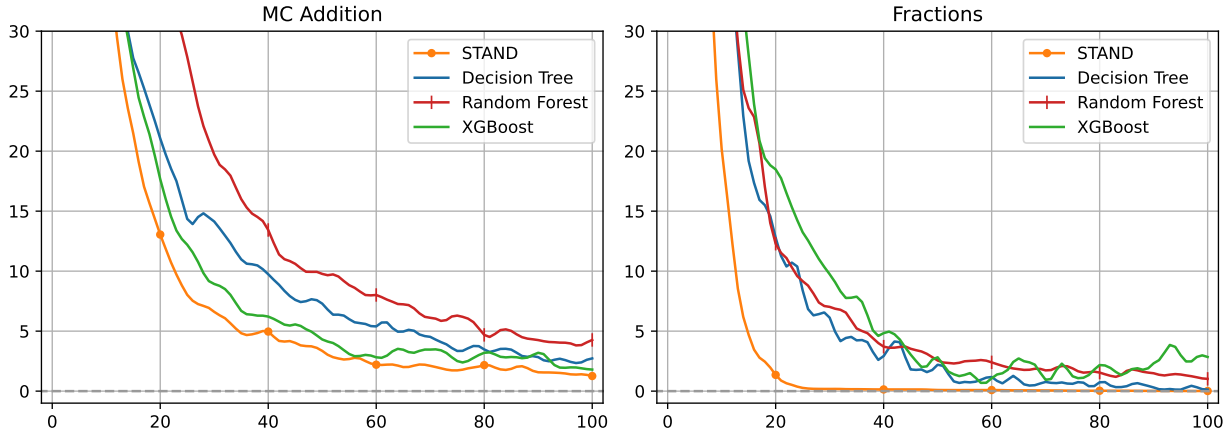

Fig. 5: Average commission errors by problem.

Our results show that STAND tends to make strictly fewer errors of omission and commission than the other models, except that in multi-column addition it does not make fewer errors of omission than the two ensemble methods.

Error Re-occurrence Rate Error re-occurrence rate is defined as the proportion of correct predictions prior to a training event that transition into being incorrect after the training event. A low error re-occurrence rate implies that errors strictly decrease with more training examples and do not spontaneously reappear in problem states that have already been given feedback. Error Re-occurrence Rates can also be broken down by type. The omission re-occurrence rate is the proportion of true positives that transition into false negatives, and the commission re-occurrence rate is the proportion of true negatives that transition into being false positives.

Table 2: Total Error Re-occurrence Rates

	MC Addition			Fractions		
	Total	Omission	Commission	Total	Omission	Commission
STAND	0.53%	0.43%	0.86%	**0.05%**	0.04%	0.08%
Decision Tree	**0.38%**	**0.00%**	1.48%	0.06%	**0.00%**	0.23%
Random Forest	1.28%	1.37%	0.98%	0.41%	0.52%	**0.07%**
XG Boost	0.74%	0.73%	**0.81%**	0.81%	0.96%	0.34%

Our results indicate that overall STAND does not succeed at reducing error re-occurrence rates over single decision trees, although it is slightly better in fractions. However, since STAND makes fewer errors overall this result is not necessarily an indication that STAND is less desirable on this front. STAND generally produces fewer error re-occurrence events than the two ensemble methods, and has a low ratio of omission re-occurrence events to commission re-occurrence events. A low rate of omission re-occurrence events is desirable since this implies that authors are less likely to find that proposed actions spontaneously disappear.

5.4 Evaluating Instance Certainty

To be informative to users, STAND's estimates of certainty must reflect actual learning progress and eventual completeness. Instance certainty $IC(x)$ should reflect STAND's learning trajectory: it should be low when receiving the correctness label of an example that would cause STAND to learn a lot, and high when STAND achieves a state of complete mastery. Additionally, increases in certainty estimates should reflect changes in performance on unseen problems.

We report several measures of desirable properties along these lines: precision at high certainties, productive monotonicity, and normalized active learning utility. We compare STAND with only those models that can produce prediction probabilities (i.e. the two ensembles), and where applicable each comparison model's prediction probabilities are negated when an action is predicted to be incorrect. This maps their values into to $IC(x)$'s range of $[-100\%, 100\%]$.

Precision at High Certainties If a when-learning classifier predicts that an action is correct with a high certainty of 90%-100% then there should be a very low probability that the user must inform the agent that the proposed action is actually incorrect.

Table 3: Total Precision at High Certainties

	MC Addition		Fractions	
	$\geq 90\%$	$= 100\%$	$\geq 90\%$	$= 100\%$
STAND	93.19%	99.81%	95.70%	100.00%
Random Forest	97.15%	94.79%	95.19%	93.72%
XG Boost	98.35%	100%	99.39%	100.0%

Our simulations show that XG Boost has the highest precision at high certainties. For predictions of 100% STAND is nearly as precise as XG Boost in multicolumn addition and equally 100% precise in fractions. For predictions of $\geq$ 90% STAND's precision is closer to 90%, which is arguably a desirable property—as it indicates some alignment of instance certainty $IC(x)$ with actual ground-truth precision.

Productive Monotonicity Productive monotonicity is defined as the proportion of changes in certainty estimates for actions in a holdout set that move toward 100% when the action is correct and -100% when the action is incorrect. High productive monotonicity reflects the degree to which changes in certainty estimates mirror actual learning gains.

Table 4: Total Productive Monotonicity

	MC Addition	Fractions
STAND	**56.74%**	**78.54%**
Random Forest	51.26%	50.61%
XG Boost	52.58%	50.90%

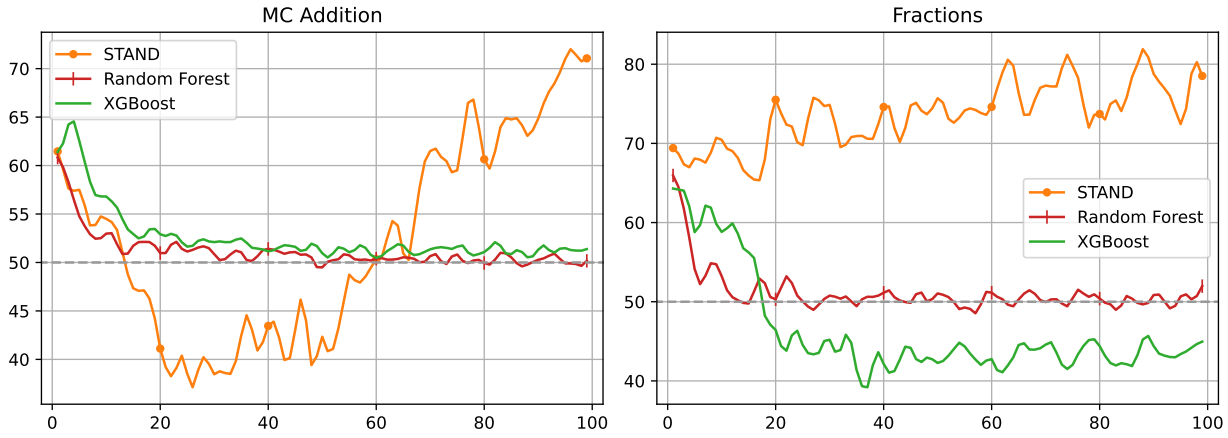

Fig. 6: Productive Monotonicity By Problem

Our results show that STAND's instance certainty $IC(x)$ has considerably higher overall productive monotonicity than the two ensemble methods' prediction probabilities. The random forest and XG Boost's prediction probabilities align with actual changes in holdout performance not much more than 50% of the time—they are not much better than chance.

In multi-column addition, STAND's productive monotonicity is $< 50\%$ for the first 60 training problems and $> 50\%$ thereafter. This may be the case in this domain because in the early stages of training STAND's version space is still growing from permitting new disjunctions. This growth introduces new leaves that reduce $IC(x)$ but increase prediction performance. In the later stages of training the version spaces enclosed by each leaf tend to gradually shrink as possible generalizations are eliminated. Fractions may not show a similar pattern because purely conjunctive preconditions tend to suffice in this domain, and so STAND's effective version space is covered by fewer leaves and tends to shrink monotonically.

Normalized Active Learning Utility If the agent proposes an action with low certainty then this should indicate high expected learning gains when the user verifies the action's correctness. Consequently, we can use instance certainty $IC(x)$ and estimates of prediction probability as heuristics in an active-learning scenario where the agent self-selects each next training problem from a pool of random problems. We define normalized active-learning utility as the average difference in completeness between agents that can and cannot self-select problems divided by the total completeness deficit of the agents that cannot:

$$\hat{U}_{active} = (C_{active} - C_{normal})/(1.0 - C_{normal}) \tag{11}$$

Active learning utility is a measure of the expected proportion of agent errors that can be eliminated by allowing the agent to self-select the problems it is instructed on. The denominator normalizes the completeness benefit of active learning by the total completeness deficit of the baseline model to control for differences in baseline model performance.

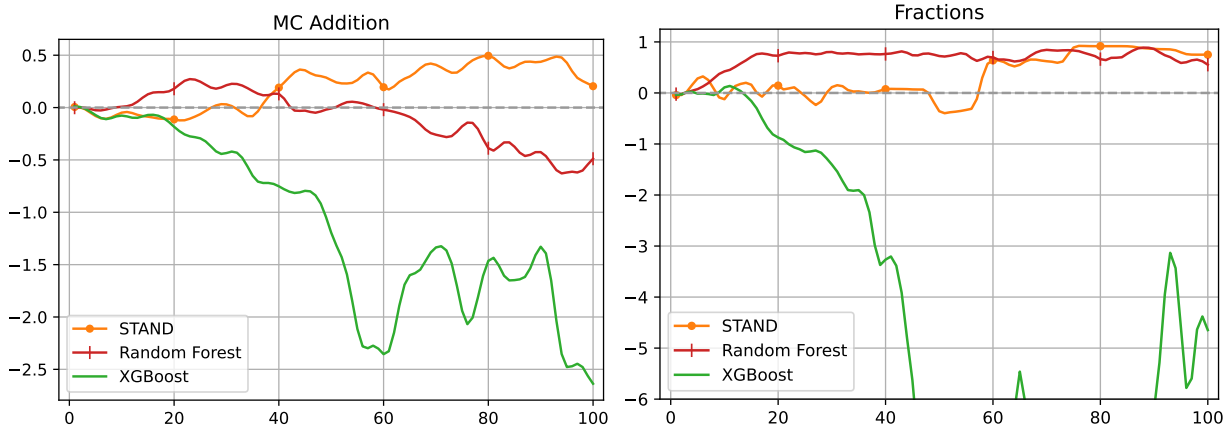

Fig. 7: Normalized Active Learning Utility by Problem

STAND shows positive active learning utility after problem 40 in multi-column addition with a peak of nearly .5 at problem 80. An active learning utility of .5 indicates that half of the remaining completeness deficit was made up by being able to self-select new problems with $IC(x)$. STAND shows high positive active learning utility in fractions after problem 60. The random forest shows some active-learning utility for early problems in multicolumn addition, and throughout training for fractions. XG Boost consistently shows negative utility in both domains.

STAND may only benefit from active learning in the latter stages of training in both domains because it is good for identifying edge cases to round out the final stages of training, but less effective when nearly any new example would be helpful.

6 Conclusion

This work reports on the implementation of a novel machine learning algorithm called STAND. We illustrated that STAND has benefits over go-to methods like XGBoost in situations like the induction of preconditions for rules in interactive task learning applications, where data tends to be small and unevenly distributed because it is generated from interactive instruction. STAND excels in situations like precondition learning where desirable features tend to be noiseless. Unlike candidate elimination approaches to version-space learning that are conceptually similar to STAND, STAND does not suffer from version space collapse and is not restricted to learning strictly conjunctive concepts. We show that STAND's unique *instance certainty* measure can predict actual changes in holdout set performance, and can be used to identify training instances that will help STAND learn most effectively. These properties address two critical problems with systems where AI can be taught bottom-up interactively: 1) users typically have no means of assessing when they are finished training an agent, and 2) they typically have no way of assessing what examples will be helpful for future training. STAND's unique self-awareness of its own learning takes a major step toward addressing these issues.

References

1. Breiman, L.: Bagging predictors. Machine learning **24**, 123–140 (1996)
2. Breiman, L.: Random forests. Machine learning **45**, 5–32 (2001)
3. Breiman, L.: Classification and Regression Trees. Routledge (2017)
4. Chen, T., Guestrin, C.: XGBoost: A scalable tree boosting system. In: Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining. pp. 785–794 (2016)
5. Cypher, A., Halbert, D.C.: Watch What I Do: Programming by Demonstration. MIT press (1993)
6. Erol, K., Hendler, J.A., Nau, D.S.: Semantics for hierarchical task-network planning. Citeseer (1994)
7. Hirsh, H.: Polynomial-time learning with version spaces. In: AAAI. pp. 117–122 (1992)
8. Kohavi, R., Kunz, C.: Option decision trees with majority votes. In: ICML. vol. 97, pp. 161–169 (1997)
9. Laird, J.E., Gluck, K., Anderson, J., Forbus, K.D., Jenkins, O.C., Lebiere, C., Salvucci, D., Scheutz, M., Thomaz, A., Trafton, G., Wray, R.E., Mohan, S., Kirk, J.R.: Interactive Task Learning. IEEE Intelligent Systems **32**(4), 6–21 (2017). https://doi.org/10.1109/MIS.2017.3121552, http://ieeexplore.ieee.org/document/8012335/
10. Matsuda, N., Cohen, W.W., Koedinger, K.R.: Teaching the teacher: Tutoring simstudent leads to more effective cognitive tutor authoring. International Journal of Artificial Intelligence in Education **25**(1), 1–34 (2015)
11. Mitchell, T.M.: Generalization as search. Artificial Intelligence **18**(2), 203–226 (1982)
12. Mitchell, T.M.: Version spaces: an approach to concept learning. Tech. rep., STANFORD UNIV CALIF DEPT OF COMPUTER SCIENCE (1978)
13. Quinlan, J.R.: Learning first-order definitions of functions. Journal of artificial intelligence research **5**, 139–161 (1996)
14. Weitekamp, D., Harpstead, E., Koedinger, K.: An interaction design for machine teaching to develop ai tutors. CHI (2020 in press)