ECAI 2012
Luc De Raedt et al. (Eds.)

doi:10.3233/978-1-61499-098-7-534

Creating Features from a Learned Grammar in a Simulated Student

Nan Li and **Abraham Schreiber** and **William W. Cohen** and **Kenneth R. Koedinger**[1]

Abstract. Understanding and developing intelligent agents that simulate human learning has been a long-standing goal in both artificial intelligence and cognitive science. Although learning agents are able to produce intelligent behavior with less human knowledge engineering than in the past, intelligent agent developers are still required to manually encode much prior domain knowledge. We recently proposed an efficient algorithm that acquires representations of the world using an unsupervised grammar induction algorithm, and integrated this representation learner into a simulated student, *SimStudent*. In this paper, we use the representation learner to automatically generate a set of feature predicates based on the acquired representation, and provide the automatically generated feature predicates to SimStudent as prior domain knowledge. We show that with the automatically-generated feature predicates, the learning agent can perform at a level comparable to when it is given manually-constructed feature predicates, but without the effort required to create these feature predicates.

1 Introduction

One of the fundamental goals of artificial intelligence is to understand and develop intelligent agents that simulate human-like intelligence. Much research (e.g., [7, 26]) has been put toward accomplishing this challenging task. By modeling human-like intelligence, we get a better understanding of how human beings acquire knowledge, and how human students' learning abilities vary from individual to individual. Understanding human learning will ultimately lead to tools and educational processes that make human students better at learning, thus making education in the 21^{st} century more effective.

Therefore, a growing body of research (e.g., [14]) has been carried out in developing intelligent agents that model human learning of math and science, and has successfully demonstrated that such agents are able to produce intelligent behavior. In order to build a more effective and cognitively plausible learning agent, it is important to reduce the amount of **supplied** prior domain knowledge and to supplant it with **learned** domain knowledge. This more accurately reflects the way a student learns basic domain knowledge before solving problems in the domain. As an additional benefit, reductions in manually encoded prior domain knowledge represent a reduction in the time-consuming and error-prone process of producing this knowledge in a form applicable by the learning agent. An intelligent agent that requires only domain-independent prior knowledge as given inputs would be a better engineering tool that can be more effectively applied across multiple domains.

[1] Carnegie Mellon University, USA, email: nli1@cs.cmu.edu, abrahamjschreiber@gmail.com, wcohen@cs.cmu.edu, koedinger@cmu.edu

The hypothesis that substantial prior knowledge is necessary for skill learning is supported by previous work in cognitive science [3]. It was shown that one of the key factors that differentiates experts and novices in a field is their different prior knowledge of world state representation. Experts view the world in terms of deep functional features (e.g., coefficient and constant in algebra), while novices only view it in terms of shallow perceptual features (e.g., integer in an expression). We [10] have recently developed an unsupervised learning algorithm that acquires deep features (e.g., what is a coefficient) automatically with only domain-independent knowledge (e.g., what is an integer) as input. Specifically, the deep feature learner acquires a probabilistic context-free grammar (pCFG) that parses input strings (e.g., *-3x*), and models the deep features as nonterminal symbols in the learned pCFG (e.g, *SignedNumber* in *Expression* $\rightarrow$ *1.0, SignedNumber Variable*).

We present both a new method for discovering perceptual feature predicates in an unsupervised way and an evaluation of this method in the context of complex skill learning. This method creates feature predicates from non-terminals in the parse tree and from relationships between non-terminals expressed in the grammar rules. We provide these automatically generated feature predicates as prior knowledge to the skill learning component of *SimStudent* [14]. More specifically, we automatically generate, from the acquired deep features, a set of predicates that can be used by the inductive logic programming (ILP) component that learns when to apply a skill. It is important and interesting that this integration of an unsupervised deep feature learner and a supervised skill learner makes it possible, for the first time, for a computer to learn a complex skill without domain-specific feature or representation engineering. Prior skill learning efforts have always required such engineering. We evaluate the quality of the automatically generated feature predicates in the algebra equation solving domain, and report the results in the experiment section.

2 A Brief Review of SimStudent

Before introducing the deep feature learner, let's first briefly describe the basic learning mechanisms of SimStudent. For full details, please refer to authors [19]. SimStudent is an intelligent learning agent that acquires skill knowledge from worked out solutions and problem solving experience. It is an extension of programming by demonstration [9] using inductive logic programming [18] as an underlying learning technique. Figure 1 shows a screenshot of the interface used to tutor SimStudent to solve algebra equations.

2.1 Knowledge Representation

Skill knowledge is represented as production rules. The left side of Figure 2 shows an example of a production rule learned by SimStu-

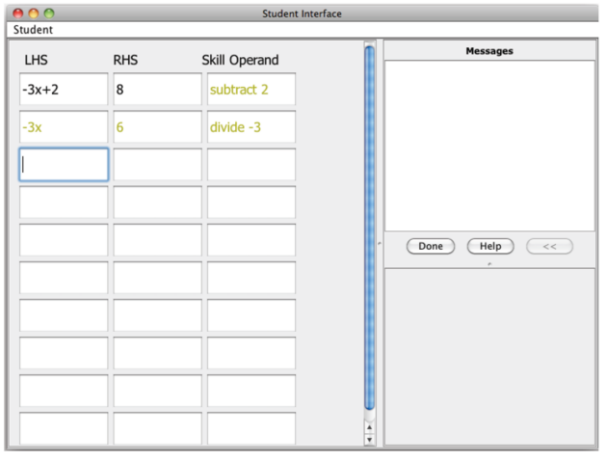


Figure 1. The interface where SimStudent is being tutored in an equation solving domain.

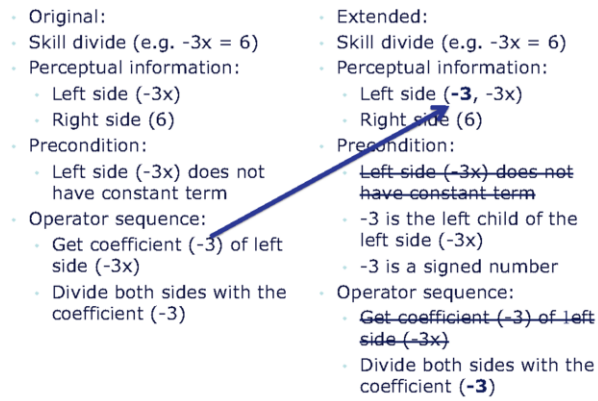


Figure 2. Original and extended production rules for divide.

dent in a readable format[2]. There are three parts in a production rule: the perceptual information part (***where*** to obtain the data needed), the precondition part (***when*** to apply the skill), and the operator function sequence part (**how** to perform the skill). The rule to "divide both sides of *-3x=6* by *-3*," shown at the left side of Figure 2, would be read as "given a left-hand side (i.e., *-3x*) and a right-hand side (6) of the equation, when the left-hand side does not have a constant term, then get the coefficient of the term at the left-hand side and divide both sides by the coefficient."

2.2 Learning by Tutoring

For each problem step (e.g., *-3x = 6*), SimStudent first tries to propose a next action based on the skill knowledge it has acquired so far. If it cannot find any applicable skill, a tutor[3] demonstrates the correct next action to SimStudent. SimStudent then updates its skill knowledge to incorporate this new training example. If SimStudent finds a next action and the tutor gives positive feedback, SimStudent directly continues to the next step without updating its skill knowledge. If the proposed next action is not correct, the tutor gives negative feedback, and demonstrates the correct next action to SimStudent. In this case, SimStudent modifies the acquired skill according to the proposed negative example and the demonstrated positive example.

2.3 Learning Mechanisms

The "where" learner acquires the perceptual information part by finding paths which identify useful information in the GUI. These pieces of useful information *percepts* are observed within the *GUI elements*, such as cells/textboxes. Elements in the interface are organized in a tree structure. For example, the table node has columns as children, and each column has multiple cells as children. Each element is *covered* by a set of paths ranging from specific to general. For instance, the possible paths to *Cell 21* are: 1) the exact path to the cell, P_{Cell21}, 2) the generalized path to any cell in row 2 or column 1, $P_{Cell2?}$ or $P_{Cell?1}$, and 3) the most general path to any cell in the table, $P_{Cell??}$. Each training example provides a list of GUI elements that are useful in generating the next action. For example, *(Cell 21, Cell 22)* is a list of cells from one training example for the skill "divide". The learning process proceeds from specific to general. The learner uses a brute-force depth-first search algorithm to find the most specific paths that cover all training examples. If we have received three training examples of skill "divide", *(Cell 21, Cell 22)*, *(Cell 11, Cell 12)* and *(Cell 51, Cell 52)*, the most specific paths that cover these training examples are *($P_{Cell?1}$, $P_{Cell?2}$)*.

The "when" learner acquires the preconditions of a production rule based on a set of *feature predicates*. Each feature predicate is a boolean function that describes relations among objects in the domain. For example, *(has-coefficient -3x)* means *-3x* has a coefficient. The feature test learner employs an inductive logic programming system, FOIL [21] to acquire a set of feature tests that describe the desired situation in which to fire the production rule. FOIL is a concept learner that acquires Horn clauses that separate positive examples from negative examples. For example, *(precondition-divide ?percept$_1$?percept$_2$)* is the precondition predicate to be learned for the production rule named "divide". *(precondition-divide -3x 6)* is a positive example since when we have -3x on one side of the equation and 6 on the other side, we would like to divide both sides by -3. *(precondition-divide 2x+4 6)* is a negative example since when we have 2x+4 on one side of the equation and 6 on the other, we would like to subtract 4. For all values that have appeared in the training examples (e.g., *-3x*, *6*, *2x+4*, *6*), we test the truthfulness of the feature predicates given all possible combinations of the observed values. For instance, for the feature predicate *(has-coefficient ?val0)*, *(has-coefficient -3x)* is true, and *(has-coefficient 2x+4)* is false. Given these inputs, FOIL will acquire a set of clauses formed by feature predicates describing the precondition predicate. In the case of skill "divide", the feature test learned is *(not (has-constant-term ?val0))*. The "when" learning process proceeds from general to specific, as FOIL starts from an empty feature test set, and grows the test set gradually until all of the training examples have been covered.

The "how" learner is given a set of basic transformations (e.g., add two numbers) called *operator functions* that can be applied to the problem. It seeks to find a sequence of operator functions that generates the correct next step using the percepts identified in the "where" part. For each training example i, the learner takes the percepts, $percepts_i$, as the input, and the step, $step_i$, as the output. We say an operator function sequence *explains* a percepts-step pair, $\langle\ percepts_i,\ step_i\ \rangle$, if the system takes $percepts_i$ as input and yields $step_i$ after applying the operator function sequence. The operator function sequence *(coefficient -3x ?coef) (divide ?coef)* is a possible explanation for $\langle$*(-3x, 6), (divide -3)*$\rangle$. Given all training examples for some skill, the learner attempts to find a shortest operator function

[2] Actual production rules follow the LISP format.

[3] Although other feedback mechanisms are also possible, in our case, the feedback is given by automatic tutors, that have been used to teach real students.

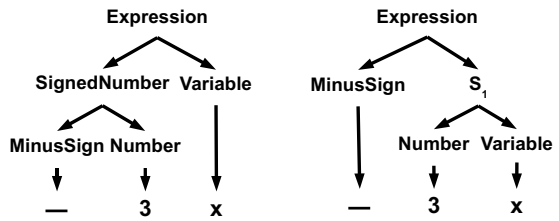


Figure 3. Correct and incorrect parse trees for $-3x$.

sequence that explains all of the ⟨*percepts, step*⟩ pairs using iterative-deepening depth-first search within some depth-limit.

Note that operator functions are divided into two groups, domain-independent operator functions and domain-specific operator functions. Domain-independent operator functions are basic skills used across multiple domains (e.g., adding two numbers, *(add 1 2)*, copying a string, *(copy -3x)*). Human students usually have knowledge of these simple skills prior to class. Domain-specific operator functions, on the other hand, are more complicated skills, such as getting the coefficient of a term, *(coefficient -3x)* and adding two terms, *(add-term 5x-5 5)*. Human students may not have enough domain expertise to perform these operator functions prior to taking a class in the domain. As we will see later, by integrating deep feature learning into SimStudent, the learning agent is able to achieve as good or better performance without domain-specific operator functions.

3 A Review of Deep Feature Learning

Having reviewed SimStudent, we move on to a review of deep feature learning, and how it was originally integrated into SimStudent. Deep feature learning is an important aspect in human knowledge acquisition. Modeling this learning procedure helps us in understanding how a novice becomes an expert. Due to the limited space, we only briefly review the propose approach, please refer to [19] for full details.

3.1 Deep Feature Learning as Grammar Induction

We [10] examined the nature of deep feature learning in algebra equation solving, and discovered that it could be modeled as a grammar induction problem given observational data (e.g. equations in algebra). As shown at the left side of Figure 3, the coefficient of *-3x* can be identified by extracting the signed number before a variable in the parse tree. Table 1 shows a context free grammar that generates the parse tree.[4] The deep feature "coefficient" then becomes a non-terminal symbol in one of the grammar rules. Viewing feature learning tasks as a grammar induction problem also explains many of the causes of student errors. For example, if the learning algorithm incorrectly learned the parse tree shown the right side of Figure 3, it will consider *3* as the coefficient, which is one of the most frequently observed errors in human student data.

Therefore, we [10] extended an existing probabilistic context-free grammar (pCFG) learner [12] to support deep feature learning and transfer learning. The pCFG learner is a variant of another pCFG learner, *the inside-outside algorithm* [8]. The input to the pCFG learner is a set of observation sequences, $\mathcal{O}$. Each sequence is a string of characters obtained directly from user input (e.g., *-3x*). The output is a pCFG that can generate all input observation sequences with high probabilities. The system consists of two parts, a greedy structure hypothesizer (GSH), which creates non-terminal symbols and associated grammar rules as needed, to cover all the training examples, and a Viterbi training step, which iteratively refines the probabilities of the grammar rules.

[4] The nonterminal name was manually added to make the discussion more readable. It would normally be an arbitrary identifier generated by the deep feature learner.

Table 1. Probabilistic context free grammar for coefficient in algebra

Terminal symbols: $-, x$;
Non-terminal symbols: $Expression, SignedNumber, Variable, MinusSign, Number$;
$Expression \rightarrow 1.0,\ [SignedNumber]\ Variable$
$Variable \rightarrow 1.0,\ x$
$SignedNumber \rightarrow 0.5,\ MinusSign\ Number$
$SignedNumber \rightarrow 0.5,\ Number$
$MinusSign \rightarrow 1.0,\ -$

After learning the pCFG, to support feature learning, the system counts the number of times that a symbol in some grammar rule corresponds to the deep features, and picks the symbol that matches the most training records as the learned feature. For instance, if most of the input records match with *SignedNumber* in *Expression → 1.0, SignedNumber Variable*, this symbol-rule pair will be considered as the target feature pattern. To support transfer learning, the learner keeps record of the acquired grammar rules as well as their application frequencies from previous tasks, and adds new rules based on previously acquired grammar rules.

3.2 Integrating Deep Feature Learning by Extending Perceptual Learning

Having built the deep feature learner, to better evaluate how the learning algorithm could affect the performance of an intelligent agent, we next introduce how to integrate deep feature learning into SimStudent. As we have mentioned above, SimStudent is able to acquire production rules in solving complicated problems, but requires a set of domain-specific operator functions given as prior knowledge. In order to both reduce the amount of prior knowledge engineering needed for SimStudent and to build a better model of real students, we integrated the feature learner into SimStudent.

Recall that GUI elements are organized in a tree structure. To incorporate the problem representation acquired by the deep feature learner into SimStudent, we extended this element hierarchy to include the parse trees corresponding to the contents of the leaf nodes (cells). Then, SimStudent prunes out irrelevant contents in the parse trees based on the output of the "where" learner. For example, -, *3*, *x* are irrelevant contents in the parse tree of *-3x*. Only *-3* is useful in solving *-3x = 6*.

Figure 2 shows a comparison between production rules acquired by the original SimStudent and the SimStudent with deep feature learning. As we can see, the coefficient of the left-hand side (i.e., *-3*) is included in the perceptual information part in the extended production rule. Therefore, the operator function sequence no longer needs the domain-specific operator function, *(coefficient -3x)*. In past work, this has been shown to generate as good or better performance while requiring much less knowledge engineering of operator functions.

4 Generating Feature Predicates from the Learned Grammar

Having removed the dependency on domain-specific operator functions, we would like to further reduce the knowledge engineer-

ing required by eliminating SimStudent's dependency on manually-constructed feature predicates. As implied by its name, the deep feature learner acquires information that reveals essential features of the problem. It is natural to think that these deep features can be used in describing desired situations to fire a production rule. In this work, we automatically generate, from the acquired deep feature grammar, a set of predicates that can be used by the inductive logic programming (ILP) component. These automatically generated feature predicates can then replace manually constructed feature predicates.

Hence, we make use of the domain-specific information in the grammar acquired by the deep feature learner to automatically generate a set of feature predicates. There are two main categories of the automatically generated feature predicates: topological feature predicates, and nonterminal symbol feature predicates. A third category, parse tree relation feature predicates, considers a combination of the information used in the first two. Each of these types of predicates are applicable to a general pCFG and the parse trees it generates. The truthfulness of the feature predicates is decided by the most probable parse tree of the problem.

4.1 Topological Feature Predicates

Topological feature predicates evaluate whether a node with the value of its first arguments exists at some location in the parse tree generated from the second argument (e.g., *(is-left-child-of -3 -3x)*). There are four generic topological feature predicates: *(is-descendent-of ?val0 ?val1)*, *(is-nth-descendent-of ?val0 ?val1)*, *(is-tree-level-m-descendent-of ?val0 ?val1)* and *(is-nth-tree-level-m-descendent-of ?val0 ?val1)*. These four generic feature predicates are used to generate a wide variety of useful topological constraints based on different *n* and *m* values. An automatically-generated predicate is created for each *m* between *0* and *M-1*, where *M* is the maximum number of nonterminal symbols on the right side of the grammar rules, and for each *n* between *0* and *N-1*, where *N* is the maximum height of the parse trees encountered.

The level specificity in the desired location varies from the most general topological predicate, *(is-descendent-of ?val0 ?val1)*, to the most specific *(is-nth-tree-level-m-descendent-of ?val0 ?val1)*. *(is-descendent-of ?val0 ?val1)* determines whether *?val0* exists anywhere in the subtree rooted at *?val1*. For example, since *3* is a grandchild of *-3x* in the parse tree shown in Figure 3, *(is-descendent-of 3 -3x)* is true. The next two topological feature predicates each encorporate one of the two pieces of information available about the location of a node: the depth of the node in the parse tree and in which subtree its located when the child nodes are ordered. *(is-nth-descendent-of ?val0 ?val1)* is slightly more specific than *(is-descendent-of ?val0 ?val1)*. It tests whether *?val0* exists anywhere in the subtree rooted at the n^{th} child of *?val1*. In the correct parse tree of *-3x*, *3* appears in the left subtree of *-3x*, therefore, *(is-Oth-descendent-of 3 -3x)* is true. *(is-tree-level-m-descendent-of ?val0 ?val1)* represents a similar level of specificity to *(is-nth-descendent-of ?val0 ?val1)*, in that it encorporates one of the two pieces of information available. It tests whether *?val0* appears at the m^{th} level in the subtree rooted at *?val1*. For instance, *3* appears at level two of the parse tree so *(is-tree-level-2-descendent-of 3 -3x)* is true. The last topological feature predicate *(is-nth-tree-level-m-descendent-of ?val0 ?val1)* considers both the tree level *m* and the descendent index *n*. It defines whether *?val0* exists *m-1*[5] levels down in the subtree rooted at the n^{th} child of *?val1*. If *m=1*, *n=0*, *(is-nth-tree-level-m-descendent-of ?val0 ?val1)* is equivalent to *(is-left-child-of ?val0 ?val1)*.

[5] We are considering nodes *m-1* levels down in the tree rooted at the n^{th} child, rather than *m*, because the n^{th} child is already 1 level down in the tree rooted at *?val1*.

4.2 Nonterminal Symbol Feature Predicates

Nonterminal symbol feature predicates are defined based on the nonterminal symbols used in the grammar rules. For example, *-3* is associated with the nonterminal symbol *SignedNumber* based in the grammar shown in Table 1. There are three generic nonterminal symbol feature predicates: *(is-symbol-x ?val0 ?val1)*, *(has-symbol-x ?val0 ?val1)*, and *(has-multiple-symbol-x ?val0 ?val1)* where *x* can be instantiated to any nonterminal symbols in the grammar.

(is-symbol-x ?val0 ?val1) describes whether *?val0* is associated with symbol *x* in the parse tree of *?val1*. For instance, *(is-symbol-SignedNumber -3 -3x)* tests whether *-3* is associated with *SignedNumber* in the parse tree of *-3x*. *(has-symbol-x ?val0 ?val1)* tests whether any node in the subtree of *?val0* is associated with symbol *x* in the parse tree of *?val1*. Although *-3* is **not** associated with symbol *Number*, it has a child, *3*, that **is** associated with symbol *Number*. In this case, *(is-symbol-Number -3 -3x)* is false, but *(has-symbol-Number -3 -3x)* is true. The last symbol feature predicate *(has-multiple-symbol-x ?val0 ?val1)* operates similarly to *(has-symbol-x ?val0 ?val1)*, but examines whether there are multiple **separate** nodes in the subtree of *?val0* which are associated with the symbol *x*. For the purposes of this predicate, two nodes *A* and *B* in a parse tree are separate iff *A* is not in *B*'s subtree and *B* is not in *A*'s subtree. In math and logic, an exact number is often less significant than whether a number falls into the category of zero, one, or many/infinite. The *(has-multiple-symbol-x)* predicate thus covers the category of many/infinite, without the need to create individual predicates for specific numbers of nodes which are associated with some symbol. *(has-multiple-symbol-Number 4-3 x+(4-3))* would return true because *4-3* has 2 nodes, *4 and 3*, each of which is associated with the symbol *Number* and neither is in the other's subtree.

4.3 Parse Tree Relation Feature Predicates

Topological feature predicates examine the position of a particular input in the overall parse tree and symbol feature predicates examine the symbol associated with a particular input. The third class of feature predicates, parse tree relation predicates, examine **both** the positions of nodes in the tree **and** their associated symbols. These allow SimStudent to examine the surrounding nodes in the parse tree and determine if they have a particular symbol from the grammar associated with them.

For the algebra study, three such predicates were used which represent examining the nearest nodes in the parse which are not in the input's subtree: *(parent-is-symbol-x ?val0 ?val1)*, *(sibling-is-symbol-x ?val0 ?val1)*, and *(uncle-is-symbol-x ?val0 ?val1)* (or *aunt*). As their names imply, these predicates examine whether a parent/sibling/uncle(aunt) node of the input is associated with the symbol *x*, where *x* could be any nonterminal symbol in the grammar. As an example, referring again to Figure 3, consider the predicate *(sibling-is-symbol-MinusSign 3 -3x)*. This would return true because in the parse tree for *-3x*, the node representing the number *3* does have a sibling whose associated symbol is minus sign. In ongoing work, these types of predicates have been generalized to encompass arbitrary relations between nodes in the tree in much the same way that *(is-child-of ?val0 ?val1)* has been generalized to *(is-tree-level-1-descendent-of ?val0 ?val1)*. An arbitrary relationship representing the relative position of any two nodes in a parse tree can be described

by the predicate *(i-j-relation-is-symbol-x ?val0 ?val1)*. This represents examining whether the nodes reached by moving up *i* times in the tree, then down *j times* are associated with the symbol *x*. Using this notation, *(1-1-relation-is-symbol-x ?val ?val1)* is equivalent to *(sibling-is-symbol-x ?val0 ?val1)*.

5 Experimental Study

In order to evaluate whether the extended SimStudent is able to acquire correct knowledge with automatically generated feature predicates, we carried out an experiment in equation solving. Although not shown here, we have demonstrated that SimStudent can be used to discover models of human students that are better than those found by experts [11].

5.1 Experiment Design

The deep feature learner was first trained on a sequence of feature learning tasks (i.e., what is a signed number, what is a term, and what is an expression). Then, SimStudent was tutored by an automatic tutor, CTAT [1], which was used by 71 human students in a classroom study, to solve basic algebra problems. All of the training and testing problems were extracted from the same classroom study. There were four sets of training problems. Each set has 35 training problems. The testing problem set contains 11 problems. In this way, we have provided Simstuent with the same information and training as would be provided to human students.

We compared three SimStudents: one SimStudent given the manually constructed feature predicates known to be useful in solving algebra problems, one SimStudent given the automatically generated feature predicates, and one SimStudent given no feature predicates.

We evaluated the effectiveness of SimStudent in two aspects: the amount of knowledge engineering needed, and the speed of learning. To assess the knowledge engineering effort required, we counted the number of lines of Java code a developer needed to write for each feature predicate, and reported the total number of lines developed for all feature predicates used in the acquired rules.

To measure learning gain, we calculated a *first attempt accuracy* and an *all attempt accuracy* for each step in the testing problem. First attempt accuracy measures the percentage of the time SimStudent's first attempt is a correct action. For all attempt accuracy, we counted the number of proposed correct steps, and reported this number, divided by the total number of correct next steps plus the number of proposed incorrect next steps. For example, if there were four possible correct next steps, and SimStudent proposed three, of which two were correct and one was incorrect, then only two correct next steps were covered, and thus the all attempt accuracy is *2/(4+1)=0.4*.

5.2 Results

Since there is no manual encoding of domain knowledge needed for the automatically generated feature predicates, the number of lines of domain-specific code needed in equation solving is 0. On the other hand, the manually constructed feature predicates required 2093 lines of Java code, which is also a measure of the amount of knowledge engineering saved by automatic feature predicate generation.

The second study we carried out focused on evaluation of learning speed. The average learning curves for the three SimStudents are shown in Figures 4(a) and 4(b). As we can see, there is a huge gap between the SimStudents with and without manually constructed feature predicates (i.e., the two blue lines). The goal of our algorithm is to fill in the gap without requiring extra knowledge engineering.

As shown in the figures, the SimStudent with automatically generated feature predicates has a slower learning curve than the SimStudent with manually constructed feature predicates. It does, however, gradually catch up after being trained on more problems. This is to be expected because while the manually constructed feature predicate directly evaluate information which is known to be applicable in solving the problem, the automatically generated feature predicates evaluate a larger set of information obtained from the parse trees. Much of this information does not turn out to be relevant to solving the problem. It therefore takes more examples for the SimStudent with automatically generated feature predicates to learn to solve the problems because it must first determine which of the automatically generated predicates are relevant. As the results show, after being trained on 35 problems, the SimStudent with automatically generated feature predicates achieved comparable performance to that with manually constructed feature predicates. This is the case for both measurements (i.e., 0.77 vs. 0.75 for first attempt accuracy, 0.83 vs. 0.79 for all attempt accuracy). Taken together, we conclude that with automatic feature predicate generation, we are able to obtain nearly comparable performance while significantly reducing the amount of knowledge engineering effort needed.

6 Related Work

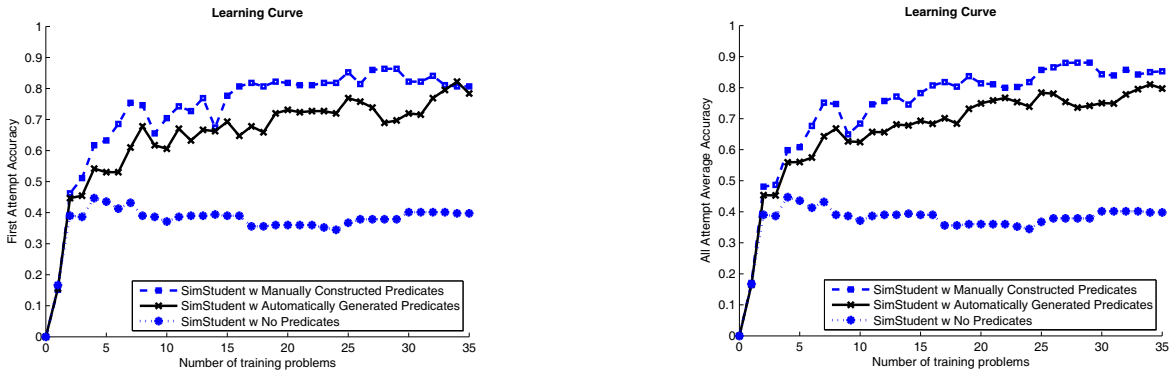
The main contribution of this paper is to reduce the amount of knowledge engineering required in building an intelligent agent by automatically generating feature predicates. Although there has been considerable work on representation change (e.g., [17, 13, 27, 5]) in machine learning, little has occurred in the context of deep feature learning. Additionally, research on deep architectures [2] and Markov logic networks [23] shares a clear resemblance with our work, but the tasks on which we work are different. These works are used more often in classification tasks whereas our work focuses on simulating human learning of math and science. Other research in cognitive science also attempts to use probabilistic approaches to model the process of human learning. Kemp and Tenenbaum [6] use a hierarchical generative model to show the acquisition process of domain-specific structural constraints, but did not integrate it an intelligent agent.

Research on ILP (e.g., [21, 22, 25]) is also closely related to our work, as SimStudent uses FOIL as its "when" learner. ILP systems acquire logic programs that separate positive examples from negative ones given an encoding of the known background knowledge. Our work differs from these systems in that it automatically generates the encoding based on a learned grammar, and calls an existing ILP algorithm to acquire the "when" part of the production rule.

Research on learning within agent architectures such as Soar [7], ACT-R [26] and so on mostly engage in speedup learning, whereas SimStudent engages in knowledge-level learning [20], and inductively acquires complex reasoning rules. Another closely related research area is learning procedural knowledge by observing others' behavior. Classical approaches include explanation-based learning [24, 16], learning apprentices [15] and programming by demonstration [4, 9]. None of these approaches tend to use a probabilistic model as a representation acquisition component in a learning agent.

7 Concluding Remarks

To sum up, one of the key challenges in building an intelligent agent is the requirement of manual encoding of prior domain knowledge. In this paper, we proposed a novel approach that takes advantage of



(a) (b)

Figure 4. Learning curves of three SimStudents in equation solving measured by, a) first attempt accuracy, b) all attempt accuracy.

the representation acquired by a deep feature learner to automatically generate a set of feature predicates, and integrate these predicates into an intelligent agent, *SimStudent*. We showed that the SimStudent with automatically generated feature predicates is able to achieve comparable performance without requiring any manually-constructed feature predicates as input.

SimStudent has been evaluated across multiple domains such as fraction addition and stoichiometry, but we have only evaluated the automatically generated features in equation solving. We would like to carry out more experiments on these feature predicates in other domains. Moreover, the deep feature learning process is currently carried out before the SimStudent knowledge acquisition. We would also like to further integrate the deep feature learner to acquire better representation knowledge during skill knowledge acquisition so that the two learning systems would mutually assist each other in achieving better performance. Lastly, we would like to compare SimStudent with human student data, and see how well does SimStudent fit with human student behavior.

REFERENCES

[1] Vincent Aleven, Bruce M. Mclaren, Jonathan Sewall, and Kenneth R. Koedinger, 'A new paradigm for intelligent tutoring systems: Example-tracing tutors', *International Journal of Artificial Intelligence in Education*, **19**, 105–154, (April 2009).

[2] Yoshua Bengio, 'Learning deep architectures for AI', *Foundations Trends in Machine Learning*, **2**, 1–127, (January 2009).

[3] Michelene T. H. Chi, Paul J. Feltovich, and Robert Glaser, 'Categorization and representation of physics problems by experts and novices', *Cognitive Science*, **5**(2), 121–152, (June 1981).

[4] *Watch what I do: programming by demonstration*, eds., Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky, MIT Press, Cambridge, MA, 1993.

[5] Tom Fawcett, 'Knowledge-based feature discovery for evaluation functions', *Computational Intelligence*, **12**(1), (1996).

[6] Charles Kemp and Joshua B B. Tenenbaum, 'The discovery of structural form.', *Proceedings of the National Academy of Sciences of the United States of America*, (July 2008).

[7] John E. Laird, Paul S. Rosenbloom, and Allen Newell, 'Chunking in soar: The anatomy of a general learning mechanism', *Machine Learning*, **1**, 11–46, (1986).

[8] K. Lari and S. J. Young, 'The estimation of stochastic context-free grammars using the inside-outside algorithm', *Computer Speech and Language*, **4**, 35–56, (1990).

[9] Tessa Lau and Daniel S. Weld, 'Programming by demonstration: An inductive learning formulation', in *Proceedings of the 1999 International Conference on Intelligence User Interfaces*, pp. 145–152, (1998).

[10] Nan Li, William W. Cohen, and Kenneth R. Koedinger, 'A computational model of accelerated future learning through feature recognition', in *Proceedings of 10th International Conference on Intelligent Tutoring Systems*, pp. 368–370, (2010).

[11] Nan Li, William W. Cohen, Noboru Matsuda, and Kenneth R. Koedinger, 'A machine learning approach for automatic student model discovery', in *Proceedings of the 4th International Conference on Educational Data Mining*, pp. 31–40, (2011).

[12] Nan Li, Subbarao Kambhampati, and Sungwook Yoon, 'Learning probabilistic hierarchical task networks to capture user preferences', in *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, Pasadena, CA, (2009).

[13] Mario Martín and Hector Geffner, 'Learning generalized policies from planning examples using concept languages', *Applied Intelligence*, **20**, 9–19, (January 2004).

[14] Noboru Matsuda, Andrew Lee, William W. Cohen, and Kenneth R. Koedinger, 'A computational model of how learner errors arise from weak prior knowledge', in *Proceedings of Conference of the Cognitive Science Society*, pp. 1288–1293, Austin, TX, (2009).

[15] Tom M. Mitchell, Sridhar Mahadevan, and Louis I. Steinberg, 'Leap: a learning apprentice for vlsi design', in *Proceedings of the 9th international joint conference on Artificial intelligence*, pp. 573–580, San Francisco, CA, (1985).

[16] Raymond J. Mooney, *A General Explanation-Based Learning Mechanism and its Application to Narrative Understanding*, Morgan Kaufmann, San Mateo, CA, 1990.

[17] S. Muggleton and W. Buntine, 'Machine invention of first-order predicates by inverting resolution', in *Proceedings of the Fifth International Conference on Machine Learning*, pp. 339–352. Morgan Kaufmann, (1988).

[18] Stephen Muggleton and Luc de Raedt, 'Inductive logic programming: Theory and methods', *Journal of Logic Programming*, **19**, 629–679, (1994).

[19] William W. Cohen Kennenth R. Koedinger Nan Li, Noboru Matsuda, 'Integrating representation learning and skill learning in a human-like intelligent agent', Technical Report CMU-MLD-12-1001, Carnegie Mellon University, (January 2012).

[20] Allen Newell, 'The knowledge level', *Artificial Intelligence*, **18**(1), 87–127, (1982).

[21] J. R. Quinlan, 'Learning logical definitions from relations', *Machine Learning*, **5**(3), 239–266, (1990).

[22] Luc De Raedt and Luc Dehaspe, 'Clausal discovery', *Machine Learning*, **26**(2), 99–146, (1997).

[23] Matthew Richardson and Pedro Domingos, 'Markov logic networks', *Machine Learning*, **62**(1-2), 107–136, (2006).

[24] Alberto Segre, 'A learning apprentice system for mechanical assembly', in *Proceedings of the Third IEEE Conference on AI for Applications*, pp. 112–117, (1987).

[25] A. Srinivasan, *The Aleph Manual*, 2004.

[26] Niels A. Taatgen and Frank J. Lee, 'Production compilation: A simple mechanism to model complex skill acquisition', *Human Factors*, **45**(1), 61–75, (2003).

[27] Paul E. Utgoff, *Shift of Bias for Inductive Concept Learning*, Ph.D. dissertation, Department of Computer Science, Rutgers University, New Brunswick, NJ, 1984.