Appears in the Proceedings of the *ICDM Workshop on Foundations and New Directions of Data Mining*, November 2003

# Speeding Up Relational Data Mining by Learning to Estimate Candidate Hypothesis Scores

Frank DiMaio and Jude Shavlik
*Computer Sciences Department, University of Wisconsin – Madison,*
*1210 W. Dayton St., Madison, WI 53706*
*{dimaio, shavlik}@cs.wisc.edu*

## Abstract

*The motivation behind* multi-relational data mining *is knowledge discovery in relational databases containing multiple related tables. One difficulty relational data mining faces is managing intractably large hypothesis spaces. We attempt to overcome this difficulty by first sampling the hypothesis space. We generate a small set of hypotheses, uniformly sampled from the space of candidate hypotheses, and evaluate this set on actual data. These hypotheses and their corresponding evaluation scores serve as training data in learning an* approximate *hypothesis evaluator. We use this approximate evaluation to quickly rate potential hypotheses without needing to score them on actual data. We test our approximate clause evaluation algorithm using the popular* Inductive Logic Programming *(ILP) system* Aleph*. We use a neural network to approximate the hypothesis-evaluation function. The trained neural network replaces* Aleph*'s hypothesis evaluation on actual data, scoring potential rules in time* independent *of the number of examples. Our approximate evaluator can also be used in a heuristic search to help escape local maxima. We test the neural network's ability in learning the hypothesis-evaluation function on four benchmark ILP domains; the neural network is able to accurately approximate the hypothesis-evaluation function.*

## 1. Introduction and Background

Most data mining techniques assume that the data exists in a form that can be easily converted into a set of *fixed-length feature vectors* (where each example is converted into a fixed-size array of real numbers, integers, and nominal attributes). For many multi-relational datasets, such a conversion – when even possible – is inelegant and scales poorly. Conversely, Inductive Logic Programming (ILP) [1] natively handles multi-relational data. ILP's natural treatment of multi-relational datasets avoids the problems associated with converting examples into feature vectors. As a further advantage, its rules have the full expressive power of first-order logic, making for rich and human-readable hypotheses.

ILP systems have been proven quite successful in constructing a set of accurate rules, even on datasets with many relations. Such systems have been successfully employed in a number of varied domains, including molecular biology, engineering design, natural language processing, and software analysis.

ILP systems combine background domain knowledge and categorized training data in constructing a set of rules (hypotheses) in first-order logic. Formally, given a training set of positive examples $E^+$, negative examples $E^-$, and background knowledge $B$, all as set of clauses in first-order logic, ILP's goal is finding a hypothesis (a set of clauses in first-order logic) $h$, such that

$$B \cup h \Rightarrow E^+ \qquad B \cup h \not\Rightarrow E^- \qquad (1)$$

That is, given the background knowledge and the hypothesis, one can *deduce* all of the positive examples, and none of the negative examples. In real world applications, these constraints are usually relaxed somewhat, allowing $h$ to explain *most* positive examples and *few* negative examples.

The algorithm underlying most ILP systems is basically the same. It searches for a clause in the *subsumption lattice* [2], evaluating candidate clauses on the training data. The search begins with an initial candidate clause, and considers hypothesis generation as a local search problem in the subsumption lattice. The starting point for the search and the type of local search depends on the implementation of the ILP system.

The subsumption lattice is constructed based on the idea of specificity of clauses. Specificity here refers to implication; a clause $C$ is more specific than a clause $S$ if $S \Rightarrow C$. In general, it is undecidable whether or not one clause in first-order logic *implies* another [3], so ILP systems use the weaker notion of *Plotkin's $\theta$-subsumption*. Subsumption implies implication, but implication *does not* imply subsumption. Subsumption of candidate clauses puts a partial ordering on all clauses in hypothesis space. With this partial ordering, a lattice of clauses can be built. ILP implementations perform some type of local search over this lattice when considering candidate hypotheses.

Most ILP implementations also use a standard greedy covering algorithm. After completing a local search of the subsumption lattice, the best rule evaluated is accepted, and all the positive examples covered (explained) by the rule are removed from the dataset. The process is repeated until every positive example is covered.

The major distinction separating various ILP implementations is the strategy used in exploring the subsumption lattice. Algorithms fall into two main categories (with some exceptions): general-to-specific ("top-down") [4] and specific-to-general ("bottom-up") enumeration of the subsumption lattice [5]. Within this framework, a variety of common local search strategies have been employed, including breadth-first search [6], depth-first search, heuristic-guided hill-climbing variants [5,6], uniform random sampling [7], and rapid random restarts [8]. Our work provides a general framework for increasing the speed of any ILP algorithm, regardless of the order candidate clauses are evaluated.

One complaint levied against ILP systems is that they scale poorly to large datasets. Srinivasan [7] investigated the performance of ILP algorithms in general, and found that the worst-case running-time depends on both the size of the subsumption lattice and the time required for clause evaluation. The first factor – the search space size – depends on the maximum allowed clause length and the number of terms in an example's *saturation*.

The idea of saturation is used by a number of ILP systems to put a bound on the size of the subsumption lattice. Saturation involves first choosing a positive example from the training set. Using the background knowledge, saturation constructs the *most specific*, *fully-ground* clause that entails the chosen example. It is constructed by applying *all possible substitutions* for variables in *B* with ground terms in *B*. This clause is called the chosen example's *bottom clause*, and it serves as the bottom element ($\perp$) in the subsumption lattice over which ILP searches. That is, all clauses considered by ILP (in the subsumption lattice) subsume (and thus imply) the saturated example.

As a simple example, suppose we are given background knowledge (using Prolog notation where ground atoms are denoted with an initial lowercase letter and variables are denoted with an initial uppercase letter):

```
f(e,b)
g(b,c)
∀X,Y,Z  f(X,Y) ∧ g(Y,Z) ⇒ h(Y)
```

And the current positive example, `e`.

We first begin saturation by letting all ground atoms in *H* imply *e*:

```
f(e,b) ∧ g(b,c) ⇒ positive(e)
```

Then we apply all possible consistent substitutions, i.e., if we make the substitutions {`e/X`, `b/Y`, `c/Z`} (using the notation {*atom/Variable*} to indicate 'atom' is being substituted for 'Variable'), we can apply the rule given in the third line of our background knowledge, that is:

```
f(e,b) ∧ g(b,c) ⇒ h(b)
```

Finally, combining gives us the saturation of *e*:

```
f(e,b) ∧ g(b,c) ∧ h(b)  ⇒  positive(e)
```

Returning to the matter of runtime complexity, given maximum clauselength *c* and bottom clause $\perp$, the worst case size of the subsumption lattice over which the ILP algorithm will search is given by [7]:

$$\frac{|\perp|^{c+1}-1}{|\perp|-1} = O(|\perp|^{c}) \cdot \qquad (2)$$

The other factor affecting ILP's performance is the evaluation time of a clause. This aspect is more complicated to analyze. Srinivasan simplifies the analysis by assuming that every clause can be evaluated on an example in constant time $\beta$; thus, the evaluation of a clause against the entire training set occurs in time $\beta|E| = O(|E|)$ where *E* is the set of training examples. An exhaustive search of the subsumption lattice for a single clause, then, takes worst-case running time $O(|\perp|^{c}|E|)$.

It is important to note that the situation is a bit worse than the O(|E|) running time makes it seem. Srinivasan's work assumed that deduction of a goal clause against a set of background relations takes a constant amount time. However, even with just one recursive rule and one background fact, deduction can be undecidable [9]. Restricting ourselves to the simpler case where function symbols are not considered (i.e., Datalog) and not allowing recursive clauses, evaluating a candidate clause against a set of ground background facts is NP-complete [10]. Most ILP datasets fall into this simpler, function-free category, where evaluation time is exponential (unless P=NP) in the number of variables, which relates to the length of the expression. As more difficult problems are encountered, it seems likely longer hypotheses will be required in order to cover all the positive examples, resulting in an execution time worse than O(|E|) indicates.

Many improvements to ILP [4,5] have focused upon finding a better search strategy, thereby reducing the fraction of the search space explored. For example, using hill-climbing to explore the search space reduces the worst-case running time to $O(|\perp||E|)$. To this end, a number of heuristic functions have been used to guide ILP searches. Many of these attempts have proven quite successful. Srinivasan employs a random sampling strategy that considers sampling *n* clauses from the subsumption lattice. The value of *n* is chosen so one is reasonably sure the best clause found is in the top *k*% of *all clauses* in the subsumption lattice up to a specified maximum length. Interestingly enough, the value of *n* is independent of the size of the subsumption lattice. This gives a much-improved worst-case running time of O(|*E*|) for generating a single clause. However, Srinivasan's idea – based upon ordinal optimization [11] – only works for domains where there are a sizable number of "sufficiently good" solutions. His technique is not appropriate for needle-in-the-haystack problems.

Still more ILP optimizations focus on decreasing the time spent on clause evaluations: the $|E|$ term in ILP's running time. Several improvements to Prolog's clause evaluation function have been developed. Blockeel *et al.* [12] consider reordering candidate clauses to reduce the number of redundant queries. Santos Costa *et al.* [13] developed several techniques for intelligently reordering terms within clauses to reduce backtracking. Srinivasan [14] developed a set of techniques for working with a large number of examples that only considers using a fraction of all available examples in the learning process.

Our work is more closely related to the latter group, with our effort spent reducing the time used by clause evaluations. By learning a function that *estimates* the clause evaluation function, we can quickly approximate the goodness of a clause, in an amount of time *independent of the number of training examples*.

We make use of a multilayer, feed-forward neural network in approximating the ILP scoring function $\pi^{E}_{evalfn}$:

$$\pi^{E}_{evalfn} : h \rightarrow \Re, \qquad (3)$$

with $h$ a candidate clause, $E$ the set of categorized training examples, and $\pi^{E}_{evalfn}$ mapping clause $h$ to $h$'s score on training set $E$ under scoring metric *evalfn*. This score represents the *goodness* of the hypothesis $h$ at explaining the training data.

We train the neural network until it approximates $\pi^{E}_{evalfn}$ with sufficient accuracy (Section 2 contains specifics of the network topology and the training process). The search is performed in the usual manner; however, instead of evaluating candidate clauses on the complete set of examples, we use the neural network to compute the *approximate* clause evaluation score $\hat{\pi}^{E}_{evalfn}$. We proceed with the algorithm as if the clause had been scored on the actual training data. Notice that this method allows clause evaluation in O(1) running time, not the $O(|E|)$ running time required to perform the actual evaluation on the training data.

Additionally, we can use the surface defined by the trained neural network to guide our search. The function encoded by a neural network *with fixed weights* defines a smooth surface in the space of network inputs. Figure 1 presents this surface graphically. Because of the smoothing nature of the approximation, the neural network can be used in order to escape from local maxima when using a heuristic to search the subsumption lattice.

This idea of using function approximation to intelligently guide a local search is not a new one. Though not in the domain of ILP, Boyan and Moore [15] use quadratic regression to approximate a function mapping points in feature space to the endpoint of a trajectory of some local search starting at that point. They use this approximation to escape local maxima in a heuristic search. Their algorithm ran in less time, and reported better test-set accuracy than solutions discovered using local search alone.

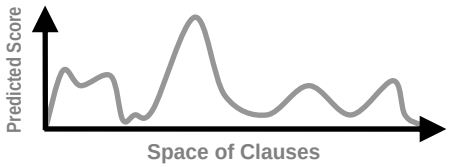


**Figure 1. A graphical representation of the function learned by the neural network. Exploring this surface may help escape local maxima in the surface defined by the clause evaluation function.**

It is important to note that empirically, this paper's only concern is with *learning* the approximation $\hat{\pi}^{E}_{evalfn}$. We have not yet thoroughly investigated using $\hat{\pi}^{E}_{evalfn}$ to speed up local searches over the subsumption lattice, or to intelligently escape local maxima when searching the hypothesis space.

## 2. Learning the Clause Evaluation Function

The first step in building our clause evaluation function approximator is construction of the neural network. This requires choosing the network inputs as well as the network topology. We base network construction on the top-down lattice exploration used by a number of popular ILP implementations. In such implementations, a positive example is chosen at random from the training set. The chosen example is then saturated, building a bottom clause $e$ :- $B$. (Note that we use Prolog notation where $e$ :- $B$ means $B \Rightarrow e$). Recall that this bottom clause consists of only *fully ground literals*. An ILP system constructs candidate hypotheses by choosing a subset of these fully-ground literals and "variablizing," replacing ground atoms with variables in a manner that replaces multiple instances of a single ground atom with a single variable. Approaches differ regarding how they select ground literals from the bottom clause. Figure 2 illustrates this process.

Our neural-network inputs are comprised of a set of features derived from the candidate clause both *before* and *after* variablization. When saturating an example, each literal in that example's bottom clause is associated with an input in the neural network. This input is set to **1** if the corresponding literal in the bottom clause was used in constructing the clause, and set to **0** otherwise. Notice that there may be multiple sets of literals from the bottom clause that, variablized, yield the same clause. However, we only set to 1 the input units for the specific literals that *were actually used* to construct the candidate clause.

Formally, let candidate clause $C$ be chosen by selecting some subset of literals from the most-specific bottom clause $\perp_i$ for current example $e_i$. We treat this clause as a vector $\vec{x} = \{x_1, \ldots, x_{|\perp_i|}\}$ in $|\perp_i|$-dimensional space, with:

$$x_k = \begin{cases} 1 & \text{if ground literal } k \text{ chosen in constructing } C \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

This vector $\vec{x}$ is part of the inputs to our neural network. One important aspect of the input vector is that every possible candidate clause - that is, every clause in Aleph's hypothesis space - has a unique input vector representation.

Additionally, we give each *functor* (function name) a specific input in the network, as well. Here, we consider a vector $\vec{y}$, in which each dimension corresponds to a functor appearing in $\perp_i$. Construction of $\vec{y}$ is based upon the number of times a particular functor is used in a candidate clause, that is:

$$y_j = \#\text{ of ground literals in } C \text{ with functor } j \quad (5)$$

Finally, a third set of inputs to the neural network comes from features extracted from the variablized clause $C'$. These features include

- **length**, the number of literals in $C'$.
- **nvars**, the number of *distinct* variables in $C'$.
- **nshared_vars**, the number of distinct variables appearing more than once in $C'$.
- **avg_var_freq**, the average number of times each variable appears in $C'$.
- **max_var_chain**, the longest variable chain appearing in $C'$, i.e., for the clause `f(A):-g(A,B),h(B,C)` the maximum chain length is 3 (A→B→C).

The neural network consists of one (fully-connected) hidden layer and a single output unit. The output corresponds to predicted output $\hat{\pi}^E_{evalfn}$. Thus, we can evaluate a clause on the neural network by converting it to the vector notation specified in **(4)** and **(5)**, and forward-propagating it on a neural network trained to approximate $\pi^E_{evalfn}$. Figure 3 details the network topology.

## 2.1. Training the Neural Network

Table 1 contains the algorithm used to train the neural network. A training set size is specified, and a number of

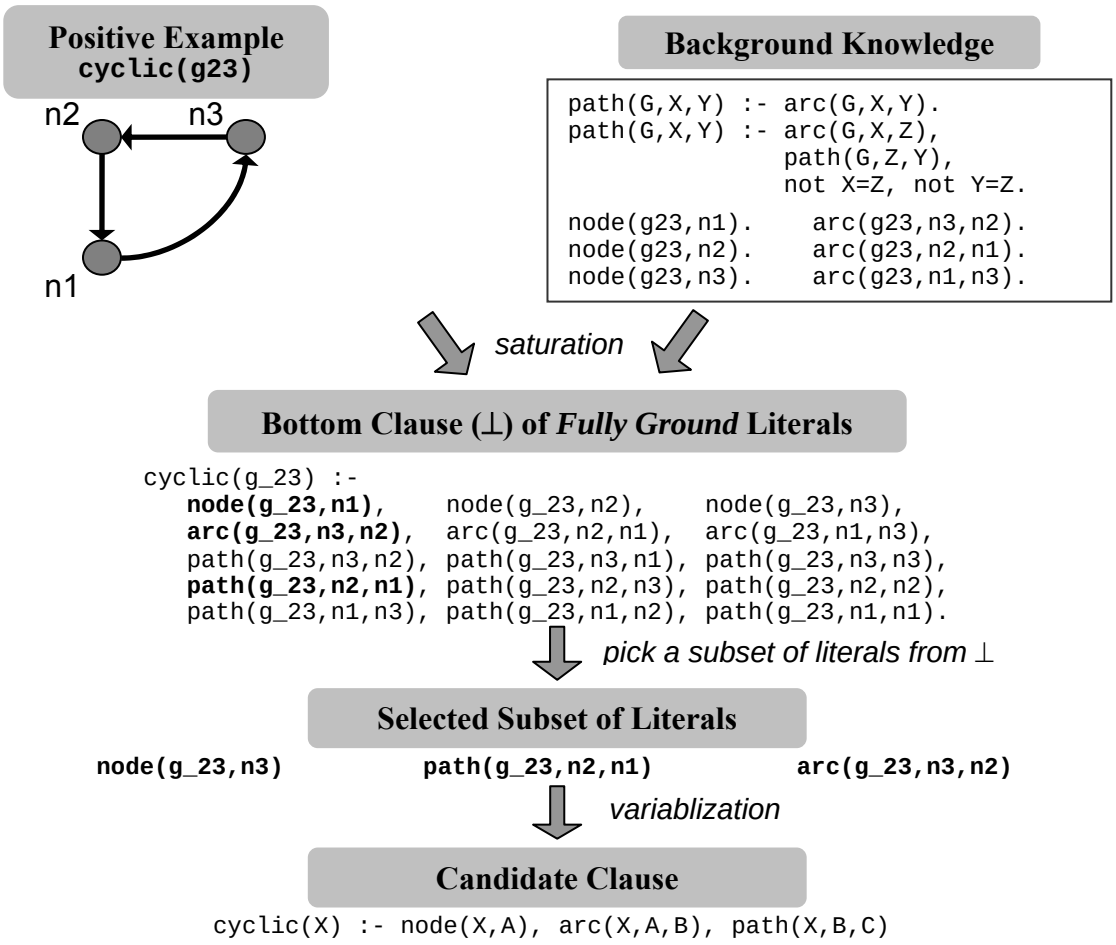


**Figure 2. An overview of the process by which many ILP algorithms construct clauses. The bottom clause is constructed by applying *all possible substitutions* of ground terms for variables in *B*. A subset of literals from the bottom clause is chosen, and through *variablization*, these literals are converted into a candidate hypothesis. Variablization replaces multiple instances of a single atom with a single variable. We use the Prolog notation for clauses, where *e* :- *B* means *B* ⇒ *e*, *ground atoms* are denoted with an initial *lowercase* letter, and *variables* are denoted with an initial *uppercase* letter.**

**Table 1: The Neural Network training algorithm. Given a bottom clause $\perp_i$, a set of training examples *E*, a heuristic function *evalfn*, and a number of training clauses *trainset_size*, train a neural network to learn the clause evaluation function $\pi^{E}_{evalfn}$. Use early stopping to avoid overtraining, and return the network learned.**

```
NeuralNetworkTraining(⊥i, E, evalfn, burnin)
    IOPairs ← ∅
    NN ← new NeuralNetwork
    minError ← +inf
    for i = 1 to trainset_size
        C ← random clause from the subsumption lattice of ⊥i
        n ← evaluate(evalfn, C, E)
        add <C,n> to IOPairs
    Split IOPairs into TrainSet and TuneSet
    for j = 1 to NUM_EPOCHS
        foreach <ex,score> in TrainSet
            run backpropogation algorithm on NN using <ex,score>
        error ← score NN on TuneSet
        if (error < minError)
            minError ← error
            bestNN ← NN
    return bestNN
```

clauses are *uniformly* randomly selected from the space of legal clauses (up to a given maximum clause length). We evaluate these randomly selected clauses on the training data, thereby creating input/output pairs for training.

The neural network's training process makes use of Srinivasan's random uniform sampling [7]. Using uniform sampling to generate I/O pairs ensures that the neural network approximation is reasonably accurate over the entire search space. Using other local search methods could bias the neural network's approximation toward some local region of the search space.

## 2.2. Speeding up Searching in ILP by Using a Trained Neural Network

Once training is complete, we can use the neural network to approximate clause scores whenever ILP's search requires a clause evaluation. When a solution is finally found, we score the best clause on actual data to determine the coverage of the solution the search located.

However, empirical evidence suggests that this problem might be better treated as an adaptive sampling problem. After the burn-in, instead of evaluating every clause on the network, we let some small percentage *k* of candidate clauses bypass the neural network. We compute these clauses' actual scores. Each clause that bypasses the network, together with its computed score, forms another I/O pair used to train the neural network. By storing some subset of recently-scored I/O pairs, every time a clause bypasses the neural network, another epoch of training occurs on all stored I/O pairs. To avoid overtraining the network on specific examples, after some time I/O pairs are removed from the set of stored pairs.

As most strategies explore more favorable regions of search space as the search progresses, adaptive sampling will learn with greatest accuracy the higher-scoring regions of the search space. Additionally, it will continue to improve the accuracy of the neural network as the search progresses. Assuming that the time required to evaluate a clause is much greater than the time required to approximate a clause on the neural network, this allows a factor $(1/k)$ speedup. Given enough training examples, this assumption is valid. Implementing and testing such usage of the approximator is a topic of our future work.

## 2.3. Using the Trained Neural Network to Escape Maxima in Local Search

We are also currently developing a search strategy to directly use the trained neural network in determining candidate clauses. The trained neural network defines some surface in the network's input space. The network's sigmoidal units will have a smoothing effect on the actual evaluation function $\pi^{E}_{evalfn}$. We can take advantage of this smoothing by using the neural-network approximation to escape local minima in hill-climbing search. This technique is suggestive of the approach taken by Boyan and Moore [15] in their STAGE algorithm. Like their approach, one can envision alternating iterations of hill-climbing using the actual evaluation function and hill-climbing using the neural-network approximation. The neural network's smoothing will tend to guide the search towards the global maximum in hypothesis space.

# 3. Results and Discussion

This section details empirical evaluation of the neural network learning task. Our goal is to ascertain whether a neural network can learn the ILP clause evaluation function. To simplifying the task, in our experiments we only consider a batch learning process, not the incremental learning process outlined in Section 2.

## 3.1. Datasets

We tested neural network learning on four standard ILP benchmarks. The tasks included predicting mutagenic activity [16] and carcinogenic activity [17] in compounds, predicting the smuggling of nuclear and radioactive materials [18], and predicting metabolic

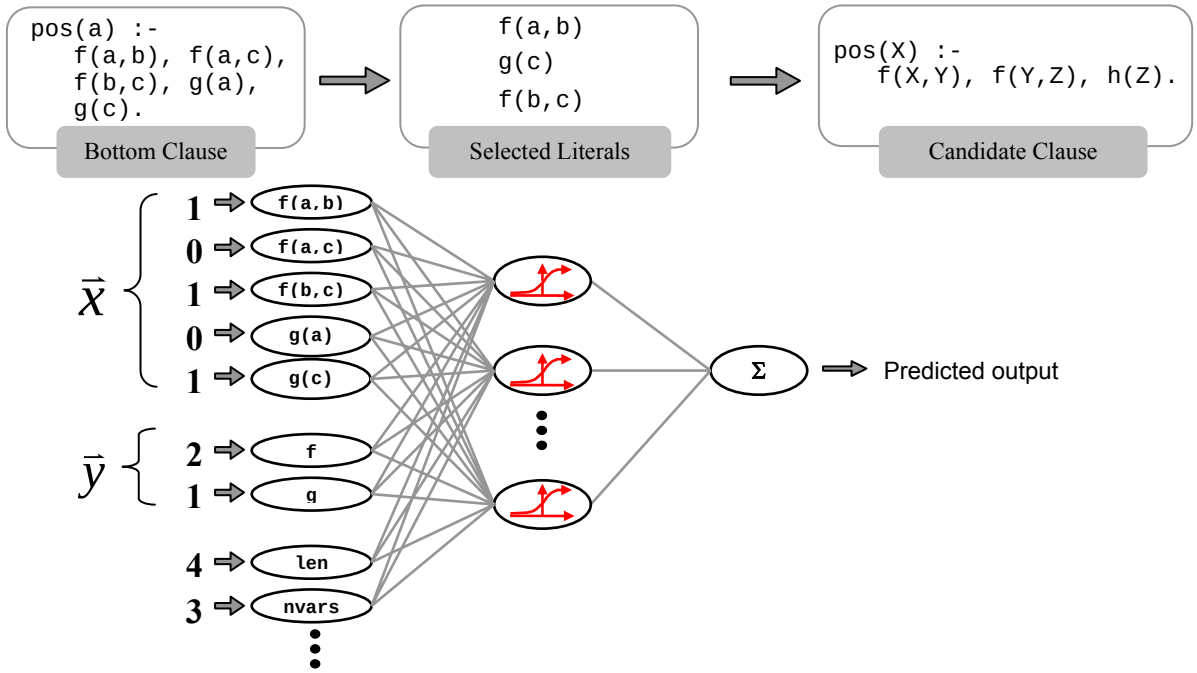


**Figure 3. An overview showing the neural network's topology, and an example of input vector construction. Notice that the vector $\vec{x}$ is constructed by the literals chosen from the fully-ground bottom clause, not the candidate clause. It is quite possible for several different sets of selected literals to correspond to the same candidate clause; we only consider the set that was *actually chosen* in the clause's construction.**

activity of proteins. A brief description of the four datasets follows.

**Mutagenesis.** This task is concerned with predicting the *mutagenicity* of certain compounds. The ILP learner is provided background knowledge consisting of the chemical properties of 188 compounds, as well as general chemical knowledge in the form of first-order logic relations. The dataset is a popular benchmark, and explores a reasonably large search space.

**Carcinogenesis.** Similar to the mutagenesis task, but an inherently more difficult problem, this task's main concern is predicting *carcinogenic* activity compounds from potential carcinogenic compounds. The database for this problem consists of 332 labeled examples, of which about half are carcinogenic.

**Nuclear Smuggling.** This dataset, based on reports of Russian nuclear materials smuggling, is interesting in its highly-relational nature, with over 40 relational tables. The task is concerned with predicting when two smuggling events are *linked*. The dataset we use is a subset of the complete dataset, 192 examples split evenly into positive and negative examples.

**Protein Metabolism.** This task is taken from the gene function prediction task of the 2001 KDD Cup challenge (www.cs.wisc.edu/~dpage/kddcup2001/). While the challenge involves learning 14 different protein functions, our sub-task is only concerned with predicting which proteins are responsible for *metabolism*. Here we also use a subset of the complete dataset, 230 examples split evenly between positives and negatives.

### 3.2. Learning the Clause Evaluation Function

We use the ILP system *Aleph* (web.comlab.ox.ac.uk /oucl/research/areas/machlearn/Aleph/aleph_toc.html) to generate 10 sets of 1000 randomly sampled clauses for each of the four datasets, corresponding to 10 different positive examples that were used in construction of the bottom clause. These 10 "seed examples" were chosen randomly. We considered a maximum clauselength $c$=6 for all but the Nuclear Smuggling task; we considered a larger value of $c$=10 for this task. Clauses were scored using a *variant of Aleph's compression* heuristic; that is, a clause's score is given by

$$\text{Score} = \frac{\left(\binom{\text{positive exs.}}{\text{covered}} + \binom{\text{negative exs.}}{\text{covered}} - (\text{clauselength}) + 1\right)}{(\text{total positive examples})} \quad (6)$$

Unlike Aleph's compression (which does not include the term in the denominator), we convert scores into a good range for neural networks by dividing by the total number of positive examples. This also allows comparison of scores across datasets.

For each dataset, these clauses and their corresponding scores were used to train the neural network. Using the machine learning package Weka [19], we generated learning curves using 10-fold cross-validation. For all datasets, the neural network was constructed with 10 hidden units. The learning rate was fixed at 0.2, with the

momentum set to 0. We added *early stopping* to Weka to avoid overtraining. For each cross-validation fold, we set aside 33% of each training set as a tuning set. Then, after 200 epochs, we *kept the neural network that performed best on the tuning set*. Weka's numeric feature normalization was enabled for all numeric input features.

The learning curves for each of the four datasets appear in Figure 4. The "All Data" curves show the *mean* root-mean-squared (RMS) error over the 10 different sets of examples. (Section 3.4 explains the other two curves in each of these graphs.)

## 3.3. Discussion of Results

As Figure 4 illustrates, for all four datasets, the hypothesis evaluation function $\pi^{E}_{evalfn}$ was learned with reasonable accuracy. In all four datasets, as more data is added to the training set, the neural network more accurately learns the evaluation function. It is interesting to note, however, that the number of examples required to accurately learn the approximator, and the accuracy of the final classifier varies amongst the datasets.

The absolute accuracy of the approximator varies across the datasets as well. For *protein metabolism*, the fully-trained network averages 0.005 RMS error; for *mutagenesis*, the results are an order of magnitude worse, at 0.05. Still, it seems promising that the worst performing approximator saw an RMS error of just 0.05.

### 3.4. Instance-independent vs. Instance-dependent Features

So far, our concern has been with learning the clause evaluation function on a rule-by-rule basis, i.e., learning a new neural network for *each* saturated example. However, several of the features we employ are independent of the example selected for saturation. In particular, every feature *except* the ground literals selected (the vector $\bar{x}$ described in Section 1) is instance-independent (or at least has an instance independent representation). These features can be shared when generating different rules from different seed examples, and, for all rules after the first, this allows us to bootstrap an initial classifier based on knowledge generated when learning previous rules.

Therefore, we looked at the contribution of each subset of features on each of the four datasets. In particular, we wanted to see how much the instance-independent features contributed to the learning task. Using the same methodology as in the Section 3.2, we used Weka to construct two additional learning curves for each of the four datasets. These two learning curves correspond to

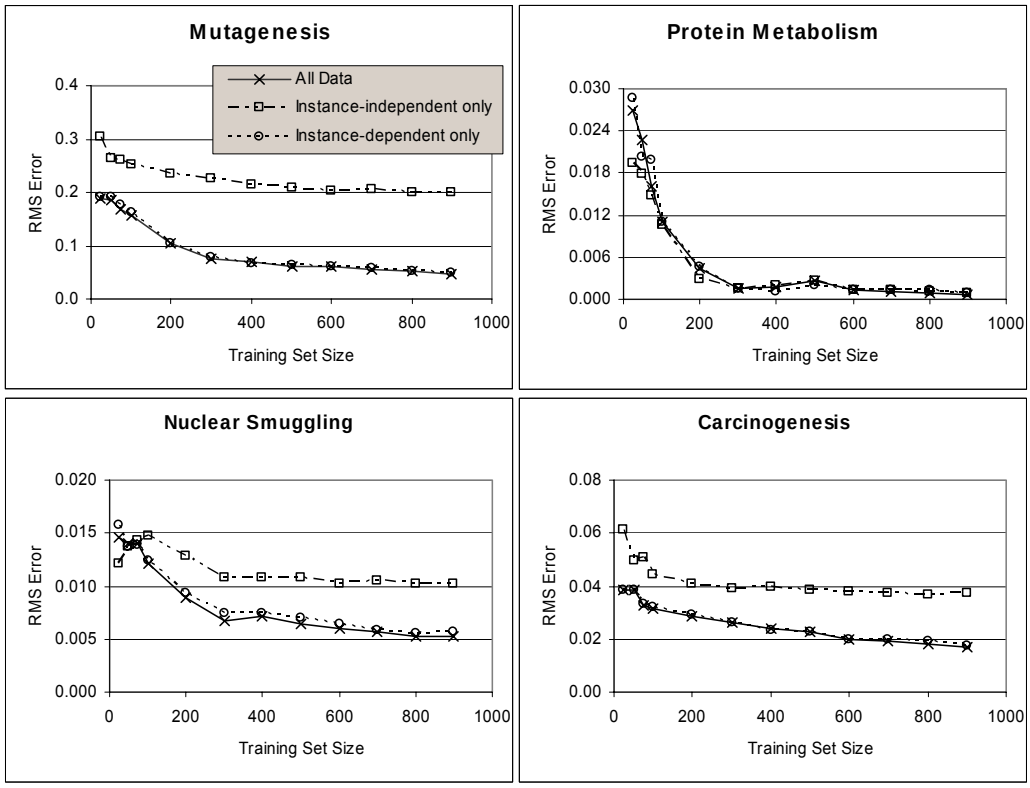


**Figure 4. Learning curves showing *test set* accuracy over four domains. The plots show how the root-mean-squared (RMS) distance between predicted and actual *score* improved as more examples were added to our training set. All four plots also show learning when we only consider instance-dependent features or instance-independent features.**

training the network on (1) only instance-independent features, and (2) only instance-dependent features.

With the exception of *protein metabolism*, training on the instance-independent features alone did not produce as accurate a classifier as training on the instance-dependent features alone, or on the complete set of features. Furthermore, *on all four datasets*, using the complete set of features did not produce a significantly more accurate network approximator than using the instance-dependent features alone did. This suggests that the instance-independent features are unlikely to help transfer learning for one seed example to the next seed example, and that better approaches need to be developed.

## 4. Conclusion and Future Work

The use of a neural network for clause evaluation seems to be a powerful tool for improving runtime efficiency when handling large search spaces in ILP. As ILP is confronted with increasingly larger problems, the need for methods like the ones we present in this paper grows. So far, we have treated the network learning and evaluation tasks as computationally "free" operations, which is not entirely true. However, it is true that the running time of neural network evaluation (and training as well) is independent of the number of ILP examples in the dataset. This means that *given enough examples in the ILP training set*, neural-network evaluation can be made virtually free. Thus, this strategy can be used to decrease the running time of ILP systems on very large tasks.

The most pressing work that remains is implementing and evaluating the strategies for taking advantage of the clause-evaluation approximator outlined in Sections 2.2 and 2.3. Clearly some accuracy is lost in approximating the clause-evaluation function, but it is difficult to determine how it will affect the quality of solutions generated by using it to quickly evaluate clauses in a typical ILP search. Another open question is whether or not we can use the network's approximation to better chose where in the hypothesis space to search.

Another possible direction for future work uses another approach idea from Boyan and Moore's STAGE algorithm [14]. Recent work in ILP search function has focused on using GSAT with rapid random restarts (RRR) to explore the subsumption lattice [8]. In this paper, we tried to learn to approximate the clause evaluation function, but perhaps, like Boyan and Moore, we should instead concentrate on learning GSAT trajectories starting in various locations of the subsumption lattice. As Boyan and Moore showed, the trajectory approximation function provides would provide even more of a smoothing effect over the search space than does the neural-network evaluation approximator, which could prove beneficial.

In conclusion, our experiments that suggest that ILP's clause-evaluation function can be approximated reasonably well using a neural network. It seems clear that as ILP problems with larger and larger datasets are encountered, strategies such as this will become increasingly important.

## 5. Acknowledgements

This work was supported by NLM grant 1T15 LM007359-01, DARPA EELD Grant F30602-01-2-0571, and NLM grant 1R01 LM07050-01

## 6. References

[1] N. Lavrac & S. Dzeroski (1994). *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood.

[2] S. Nienhuys-Cheng & R. de Wolf (1997). *Foundations of Inductive Logic Programming*. Springer-Verlag.

[3] M. Schmidt-Schauss (1988). Implication of clauses is undecidable. *Theoretical Computer Science,* 59:287-296.

[4] J. Quinlan (1990). Learning logical definitions from relations. *Machine Learning*, 5:239-266.

[5] S. Muggleton & C. Feng (1990). Efficient induction of logic programs. *Proc. 1st Conf. on Algorithmic Learning Theory*, 368-381.

[6] S. Muggleton (1995). Inverse entailment and Progol. *New Generation Computing,* 13:245-286.

[7] A. Srinivasan (2000). A study of two probabilistic methods for searching large spaces with ILP. *Tech. Report PRG-TR-16-00*. Oxford Univ. Computing Lab.

[8] F. Zelezny, A. Srinivasan & D. Page (2002). Lattice-search runtime distributions may be heavy-tailed. *Proc. 12th Intl. Conference on ILP,* 333-345.

[9] P. Hanschke & J. Wurtz (1993). Satisfiability of the smallest binary program. *Info. Proc. Letters*, 496:237-241.

[10] E. Dantsin, T. Eiter, G. Gottlob & A. Voronkov (2001). Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33:374-425.

[11] Y. Ho, R. Sreenivas & P Vakili (1992). Ordinal optimization in DEDS. *Discrete Event Dynamic Systems: Theory and Applications*, 2:61-68.

[12] H. Blockeel *et al.* (2002). Improving the efficiency of inductive logic programming through the use of query packs. *J. AI Research*, 16:135-166.

[13] V. Santos Costa *et al.* (2003). Query transformations for improving the efficiency of ILP systems, *J. Machine Learning Research,* 4:465-491.

[14] A. Srinivasan (1999). A study of two sampling methods for analysing large datasets with ILP. *Data Mining and Knowledge Discovery*, 3:95-123.

[15] J. Boyan & A. Moore (2000). Learning evaluation functions to improve optimization by local search. *J. Machine Learning Research*, 1:77-112.

[16] R. King, S. Muggleton, A. Srinivasan & M. Sternberg (1996). Structure-activity relationships derived by machine learning. *PNAS*, 93:438-442.

[17] B. Dolsak & S. Muggleton (1991). The application of ILP to finite element mesh design. *Proc. 1st Intl. Workshop on Inductive Logic Programming*, 225-242.

[18] S. McKay, P. Woessner & T. Roule (2001). Evidence extraction and link discovery seedling project, database schema description. *Veridian Technical Report 2862*.

[19] I. Witten & E. Frank (1999). *Data Mining*. Morgan Kaufmann Publishers.