Appears in the *Proceedings of the 14th International Conference on Inductive Logic Programming* (2004)

Learning an Approximation to Inductive Logic Programming Clause Evaluation

Frank DiMaio and Jude Shavlik

Computer Sciences Department, University of Wisconsin - Madison,
1210 W. Dayton St., Madison, WI 53706
{dimaio,shavlik}@wisc.edu

Abstract. One challenge faced by many Inductive Logic Programming (ILP) systems is poor scalability to problems with large search spaces and many examples. Randomized search methods such as stochastic clause selection (SCS) and rapid random restarts (RRR) have proven somewhat successful at addressing this weakness. However, on datasets where hypothesis evaluation is computationally expensive, even these algorithms may take unreasonably long to discover a good solution. We attempt to improve the performance of these algorithms on datasets by learning an approximation to ILP hypothesis evaluation. 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 for learning an approximate hypothesis evaluator. We outline three techniques that make use of the trained evaluation-function approximator in order to reduce the computation required during an ILP hypothesis search. We test our approximate clause evaluation algorithm using the popular ILP system Aleph. Empirical results are provided on several benchmark datasets. We show that the clause evaluation function can be accurately approximated.

1 Introduction

Inductive Logic Programming (ILP) systems [1] have been widely used in classification, data mining, and information extraction tasks. Their natural treatment of relational data, harnessing the expressive power of first-order logic, makes them useful for working with databases containing multiple relational tables. ILP systems combine background domain knowledge and categorized training data in constructing a set of rules in the form of first-order logic clauses. Formally, given a training set of positive examples E^+, negative examples E^-, and background knowledge B, all as sets of clauses in first-order logic, ILP's goal is to find a hypothesis (a set of clauses in first-order logic) h, such that

$$B \cup h \Rightarrow E^+ \qquad B \cup h \not\Rightarrow E^- \tag{1}$$

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 typically relaxed, allowing h to explain *most* positive examples and *few* negative examples. ILP systems have been successfully employed in a number of varied domains including molecular biology [2,3], engineering design [4], natural language processing [5], and software analysis [6].

One challenge many ILP systems face is scalability to large datasets with large hypothesis spaces. We define a general framework for learning a function that *estimates* the goodness of a hypothesis without looking at actual data. We suggest a number of ways in which such an approximation may be employed. One possible application eliminates poor hypotheses without wasting time evaluating them. Another uses the approximate hypothesis evaluator to guide the generation of promising new candidate hypotheses. Yet another application mines the estimator function itself for rules that can be used to invent useful predicates.

The remainder of the paper is structured as follows. Section 2 provides a background and related work on scaling up ILP. Section 3 describes construction of the hypothesis evaluation estimator. Section 4 describes in detail possible uses of such an estimator function. Section 5 shows some results of estimator learning on benchmark datasets, and Section 6 presents future research directions.

2 ILP Background and Related Work

The algorithm underlying most ILP systems is basically the same – it treats hypothesis generation as a local search in the *subsumption lattice* [7]. 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 [8], so ILP systems use the weaker notion of *Plotkin's θ-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, as in Figure 1. ILP implementations perform some type of local search over this lattice when considering candidate hypotheses.

The major distinction separating various ILP implementations is the strategy used in exploring the subsumption lattice. Algorithms fall into two main categories (with

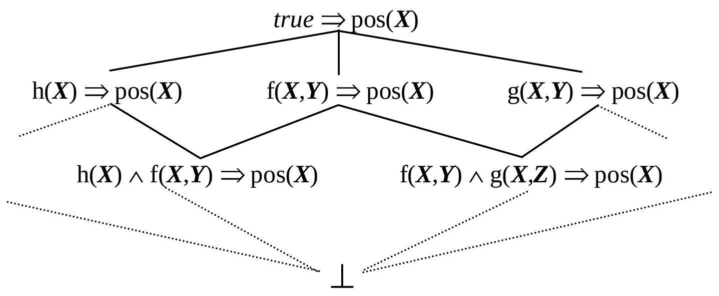


Figure 1. This illustrates an example of the subsumption lattice over which many ILP implementations search. The lattice is bounded above by *true*, and below by the bottom clause. Many ILP systems treat clause discovery as local search, moving along lattice edges.

some exceptions): general-to-specific ("top-down") [9] and specific-to-general ("bottom-up") exploration of the subsumption lattice [10]. Within these two frameworks, a variety of common local search strategies have been employed, including breadth-first search [11], depth-first search, heuristic-guided hill-climbing variants [10,11], uniform random sampling [12], rapid random restarts [13], and genetic algorithms [14]. Our work provides a general framework for increasing the speed of any ILP algorithm, regardless of the order candidate clauses are evaluated.

One challenge ILP systems face is that they scale poorly to large datasets. Srinivasan [12] investigated the performance of various ILP algorithms, and found that the running-time depends on two factors: (1) the size of the subsumption lattice and (2) the time required for clause evaluation, which in turns depends on the number of examples in the background corpus.

The first factor – the size of the subsumption lattice – mainly depends on the number of terms in a specific example's *saturation*. Saturation is used to put a lower bound on the subsumption lattice. The process is performed on a *single positive example*. 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 the background knowledge *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 (Figure 1) over which ILP searches. That is, all clauses considered by ILP (in the subsumption lattice) subsume $\perp$.

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)
```

We are also given 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)
```

Clearly, the size of the subsumption lattice is directly related to the size of $\perp$. If we ignore multiple variablizations of a single ground literal and consider only hypotheses that contain less than *c* terms, then the size of the subsumption lattice − given a bottom clause $\perp$ − is at most $O(|\perp|^c)$ [12]. Taking into account multiple variablizations introduces an additional factor, exponential in the number of constants in the bottom clause.

The second factor – the evaluation time of a clause – is more complicated to analyze. Srinivasan simplifies the analysis by assuming that every clause can be evaluated on an example in constant time β; 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 $O(|\perp|^c|E|)$.

However, for most datasets clause evaluation is even worse than O(|E|). Srinivasan's work assumed that deducing each candidate hypothesis takes constant time. However, even with just one recursive rule and one background fact, deduction can be undecidable [15]. 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 [16]. 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. In other words, a long hypothesis will take significantly longer to test against the examples in the background knowledge than will a shorter hypothesis. For many large datasets, it is precisely these long hypotheses that are most interesting. As a result, approaches to scaling up ILP [9,10] have focused upon one of these two factors: reducing the number of clauses considered, or decreasing the time spent on clause evaluations.

In reducing the number of clauses considered, the simplest techniques employ general AI search strategies, such as **A***, iterative deepening, or beam search, to reduce the number of clauses in the subsumption lattice considered. For example, using a beam reduces the worst-case running time to $O(|\perp||E|)$. However, for extremely large datasets where $|\perp|$ may be in the thousands and $|E|$ in the hundred thousands, even this may take prohibitively long.

A novel approach at reducing the number of clauses in the subsumption lattice considered has been successfully employed by Srinivasan. It uses a random sampling strategy that considers sampling n clauses from the subsumption lattice, where the value of n chosen is independent of the size of the subsumption lattice. This gives worst-case running time of $O(|E|)$ for finding a single clause. However, Srinivasan's idea only works for domains where there are a sizable number of "sufficiently good" solutions. Recent work by Zelezny *et al.* [13] has coupled random clause generation method with heuristic search using the idea of *rapid random restarts* (RRR) to explore the subsumption lattice. They repeatedly generates random clauses followed by a short local search. Rückert and Kramer [17] have also had success using stochastic search for bottom-up rule learning, outperforming GSAT and WalkSAT.

Other ILP optimizations focus instead 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.* [18] consider reordering candidate clauses to reduce the number of redundant queries. Santos Costa *et al.* [19] developed several techniques for intelligently reordering terms within clauses to reduce backtracking. Srinivasan [20] 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. Sebag and Rouveirol [21] use stochastic matching to perform approximate inference in polynomial (as opposed to exponential) time. Maloberti and Sebag [22] provide an alternative to Prolog's SLD resolution for θ-subsumption. They instead treat θ-subsumption as a constraint satisfaction problem (CSP), then use a combination of CSP heuristics to quickly perform θ-subsumption.

Our work is distinct from all of these techniques. We describe a method for learning a function that *estimates* the clause evaluation function, which can be used in several different ways. It can reduce the evaluation time of a clause by quickly approximating the goodness of a clause, in an amount of time *independent of the number of training examples*. We can couple it with Zelezny *et al.*'s rapid random restart method in order to bias restarts toward better regions in the search space. We can use it in a manner similar to Boyan and Moore's STAGE algorithm [23] to escape local maxima in a heuristic search. Finally, we can extract hypotheses and perform predicate invention using the estimator itself.

3 Learning the Clause Evaluation Function

Heuristic approaches to exploring the subsumption lattice all make use of a scoring function to represent the *goodness* of a hypothesis at explaining the training data. Given a hypothesis (a candidate clause in first-order logic) *h*, a set of categorized training examples $E = \{E^+, E^-\}$, $\boldsymbol{\pi}^E_{evalfn}$ maps clause *h* to *h*'s score on training set *E* under scoring metric *evalfn*:

$$\boldsymbol{\pi}^E_{evalfn} : h \rightarrow \Re \qquad (2)$$

We use a multilayer, feed-forward neural network described in Section 3.1 to learn an approximate scoring function $\hat{\boldsymbol{\pi}}^E_{evalfn}$. Some preliminary testing revealed that other machine learning algorithms (e.g. naïve Bayes, linear regression, C4.5) were significantly less accurate at approximating the clause evaluation function. Furthermore, a neural network with a single hidden layer is capable of approximating any bounded continuous function with arbitrarily small error [24]. We use an online training algorithm detailed in the Section 3.2 to train the neural network.

3.1 Neural Network Topology

Before constructing our clause evaluation function approximator, we need a method for encoding clauses as neural network inputs. Our encoding is based 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 ($\perp$). 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 (our approach does *not* consider multiple – or *split* - variablizations of a single set of fully-ground literals). Approaches differ in how they select ground literals from the bottom clause.

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, when variablized, yield the same clause. This means there may be many different input representations for a single clause. However, we only use the input representation corresponding to the specific literals that *were actually chosen* when constructing 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:

$$x_k = \begin{cases} 1 & \text{if ground literal k chosen in constructing C} \\ 0 & \text{otherwise} \end{cases} \qquad \textbf{(3)}$$

This vector $\vec{x}$ is a subset 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 the subsumption lattice − has a unique input vector representation. However, the mapping does not work in the other direction: not every possible bit vector corresponds to a legal clause. In many cases, the majority of bit vectors correspond to *illegal* clauses, which contain unbound input variables. (Algorithms using the neural network to search the space of bit vectors, as in Section 4.2, need to be aware of this).

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

$$y_j = \#\text{ of ground literals in } C \text{ of predicate } j \qquad \textbf{(4)}$$

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

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

The neural network consists of one (fully-connected) hidden layer and a two output units. The output units correspond to P and N, the predicted positive and negative coverage of a clause (that is, the number of examples from E^+ and E^-, respectively, deduced from the hypothesis). Given these predicted values and a scoring function, computation of the predicted output $\hat{\boldsymbol{\pi}}^{E}_{evalfn}$ is trivial. For example, commonly used evaluation functions include *coverage* ($P-N$) and *accuracy* ($P/P+N$). Thus, we can

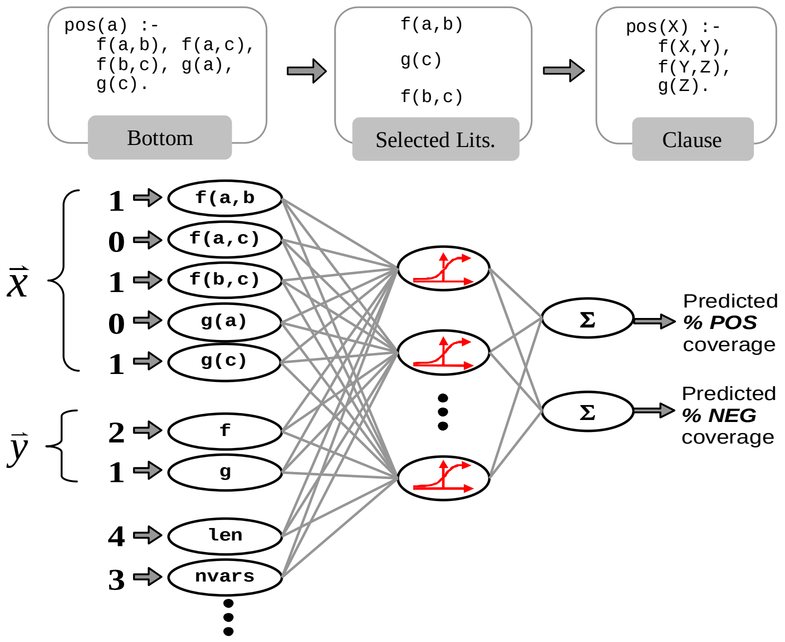


Figure 2. An overview showing the neural network's topology, and an example of input vector construction. Notice that the vector $\vec{x}$ is constructed by the literals chosen from the fully-ground bottom clause, not the candidate clause. It is quite possible for several different sets of selected literals to correspond to the same candidate clause; we only consider the set that was *actually chosen* in the clause's construction.

evaluate a clause on the neural network by converting it to the vector notation specified in Equations **(3)** and **(4)**, forward-propagating it on a trained neural network to estimate $\hat{P}$ and $\hat{N}$, and calculating $\hat{\boldsymbol{\pi}}^{E}_{evalfn}$ from $\hat{P}$ and $\hat{N}$. Figure 2 presents this network topology graphically.

3.2 Online Training

The neural network's initial training makes use of Srinivasan's random uniform sampling [12]. The user specifies a burn-in length *b*, and the algorithm uniformly randomly selects *b* clauses from the space of legal clauses (up to a given maximum clause length). We evaluate these clauses on the training data, thereby creating input/output pairs for training. 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 tends to bias the neural network's approximation toward some local region in the search space. Table 1a contains an overview of the algorithm used to initially train the neural network.

The methods we present in the Section 4 – that use our approximation to explore the subsumption lattice – continue to evaluate clauses (on actual data) once the relatively short burn-in period is concluded. It seems wasteful to just throw this potential training data for the network approximation away. Our algorithm uses an *online learning algorithm* to make use of these clause evaluations – that occur as part of ILP's regular search – to improve the accuracy of the approximation. This allows us to generate a virtually unlimited number of I/O pairs for our network by simply scoring clauses on actual data.

Our online training algorithm is shown in Table 1b. When a clause is evaluated by ILP, generating an I/O pair for training our neural network, our online learning algorithm adds the pair to a cache of recently evaluated clauses. The cache typically stores 1000 to 10000 recently evaluated clauses, and, once full, elements in the cache are randomly removed to make room for incoming elements. At regular intervals (typically every 50-100 insertions) the neural network is updated by backpropagation, using the entire cache for a fixed number of epochs (typically 10). The continually changing training set, relatively short training intervals, and small number of hidden units (typically 5-10) prevent overtraining.

While the goal of our approximation is to learn an approximation of the clause evaluation function over the entire subsumption lattice, we are *especially* concerned with high accuracy of this approximation in high-scoring regions of the subsumption lattice. To ensure this accuracy, we also maintain a cache of the *best* clauses seen so

Table 1: The Neural Network burn-in training and online training algorithms. (a) The burn-in training algorithm. Given bottom clause $\perp_i$, a set of training examples *E*, and the size of the training set *trainset_size*, train a neural network to learn the clause evaluation function $\boldsymbol{\pi}^{E}_{evalfn}$. We use early stopping to avoid overtraining, returning the learned network. **(b)** The online training algorithm, called for each I/O pair <*C*,{*pos*,*neg*}> that ILP generates. The algorithm keeps a cache of recent and best-scoring clauses. At some regular interval (every *arrivals_between_updates* arrivals), the algorithm updates trained network *NN* for a preset number of epochs (*epochs_per_update*). When a new arrival overflows the cache, it removes old items at random.

(a)

```
BurninTraining(⊥i, E, burnin)
  IOPairs ← ∅
  NN ← new NeuralNetwork
  minError ← +inf

  for i = 1 to burnin
    C ← rand. clause built from ⊥i
  {pos,neg} ← evaluate(evalfn, C, E)
    add <C,{pos,neg}> to IOPairs

  Split IOPairs into TrainSet and TuneSet

  for j = 1 to MAX_EPOCHS
    foreach <ex,{pos,neg}> in TrainSet
      run backprop on NN using <ex,{pos,neg}>
    error ← SSE of NN on TuneSet
    if (error < minError)
      minError ← error
    bestNN ← NN

  return bestNN
```

(b)

```
OnlineTrainingArrival(NN, <C,{pos,neg}>)
  if full(recent_cache)
   delete_random(recent_cache)
  insert <C,{pos,neg}> into recent_cache

  if score(pos,neg) > min(best_cache)
   insert <C,{pos,neg}> sorted into best_cache

  num_arrivals ← num_arrivals + 1
  if (num_arrivals = arrivals_between_updates)
   num_arrivals ← 0

   for j = 1 to epochs_per_arrival
    foreach <ex,{p,n}> in recent_cache
      run backprop on NN using <ex,{p,n}>
      foreach <ex,{p,n}> in best_cache
        run backprop on NN using <ex,{p,n}>

   return NN
```

far. This cache is typically 10% of the size of the recent-clauses cache, and when this cache is full, the lowest-scoring element is always removed to make room for incoming, higher-scoring clauses. When the neural network is updated, clauses in the best-scoring cache are also added to the training set and used to update the neural network as well.

4 Using the Clause Evaluation Approximation

This section describes three methods for using our clause approximator to scale ILP to larger datasets, and speed discovery of high-scoring hypotheses. These methods are:

(1) approximately evaluating clauses during the search of the subsumption lattice
(2) using the *evalfn* surface defined by the neural network to escape local maxima and to bias random restarts
(3) extracting hypotheses and performing predicate invention using the approximator function

4.1 Rapidly Exploring the Subsumption Lattice Using the Clause Approximator

This first method allows us to piggyback on just about any other local search method (though not stochastic methods). We perform our search in the usual manner; however, when we expand a node, instead of evaluating successor clauses on the complete set of examples, we use the neural network to compute the *approximate* clause evaluation score $\hat{\boldsymbol{\pi}}^{E}_{evalfn}$. We then choose the next node to expand depending on our search strategy and the approximate scores. If this next node was approximately scored on the network, we then score it on actual data (and cache it for future training). We expand this new node and repeat the process. Recall that approximate evaluation takes O(1) running time, not the $\mathrm{O}(|E|)$ running time required to perform the actual evaluation on the training data.

Interestingly enough, the behavior of this technique varies quite a bit depending on the search strategy employed. For a branch-and-bound search, this method serves to optimize the order in which clauses are evaluated – coupled with pruning, this could significantly reduce the total number of $\mathrm{O}(|E|)$ real evaluations required. With **A*** search, this instead lets one "throw away" clauses that don't seem promising without wasting time evaluating them on actual data. Clauses that the neural network predicts to score poorly will never reach the font of the open list and will never be evaluated on the actual data (Note that this does break the guaranteed optimality of **A***).

Nix and Weigend have developed a technique for using a neural network to predict not only a regression value, but also to place an error bar on its prediction [25]. Using their technique, we can instead approximately score clauses, storing them in the open list with a 95% confidence bound instead of simply their predicted score. This tends to favor evaluation of clauses that the neural network cannot accurately predict – areas that should probably be thoroughly explored (but still seem promising!).

4.2 Biasing Random Restarts towards Favorable Regions of Search Space

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. We can employ this neural-network designed surface in a stochastic search. For example, we can use this surface to perform "biased" rapid random restarts (hereafter referred to as biased-RRR): instead of randomly selecting literals, we perform stochastic gradient ascent on the neural-network defined surface. That is, starting from a random clause, we perform stochastic gradient ascent on this surface. The endpoint is our "random restart": the point from which we begin evaluating clauses on the actual training examples. These "guided" restarts bias search toward better regions of the search space.

One issue that arises is that the neural network contains two separate output units – one that predicts positive coverage and one that predicts negative coverage – and we want to perform gradient ascent over the surface of some scoring function that is a (possibly nonlinear) combination of the two. Fortunately, for all of the common scoring functions we can derive a simple expression relating the derivative of the scoring function to the derivative of the two output units. The derivatives of each output unit with respect to the input – $\partial P / \partial x_i$ and $\partial N / \partial x_i$ – are easily computed with a backpropagation variant (backprop computes $\partial Err_P / \partial w_{ij}$ and $\partial Err_N / \partial w_{ij}$). Table 2 summarizes these expressions for commonly used scoring functions.

An interesting variant of this approach uses the network-defined surface to escape local maxima while performing a standard ILP best-first search. We can think of this as equivalent to the biased rapid random restart above; however, instead of some

Table 2. This table expresses the gradient of several common scoring functions π_{evalfn} in terms of the gradients of the two network output units – predicted positive and predicted negative coverage. Stochastic gradient ascent uses one of these equations to compute the network-surface gradient under some scoring function. In the equations below, *P* denotes positive coverage, *N* denotes negative coverage, and *L* denotes clause length.

Scoring Function	π	Gradient Ascent Equation
compression	$P-N$	$\frac{\partial\pi}{\partial x_i}=\frac{\partial P}{\partial x_i}-\frac{\partial N}{\partial x_i}$
coverage	$P-N-L+1$	$\frac{\partial\pi}{\partial x_i}=\frac{\partial P}{\partial x_i}-\frac{\partial N}{\partial x_i}$
accuracy	$\frac{P}{P+N}$	$\frac{\partial\pi}{\partial x_i}=\frac{1}{(P+N)^2}\left(N\cdot\frac{\partial P}{\partial x_i}-P\cdot\frac{\partial N}{\partial x_i}\right)$
Laplace	$\frac{P+1}{P+N+2}$	$\frac{\partial\pi}{\partial x_i}=\frac{1}{(P+N+2)^2}\left((N+1)\cdot\frac{\partial P}{\partial x_i}-(P+1)\cdot\frac{\partial N}{\partial x_i}\right)$
entropy	$-\frac{P}{P+N}\log\left(\frac{P}{P+N}\right)-\frac{N}{P+N}\log\left(\frac{N}{P+N}\right)$	$\frac{\partial\pi}{\partial x_i}=\frac{1}{(P+N)^2}\cdot\left(N\cdot\frac{\partial P}{\partial x_i}-P\cdot\frac{\partial N}{\partial x_i}\right)\cdot\left(\ln\frac{N}{N+P}-\ln\frac{P}{N+P}\right)$
GINI	$2\cdot\frac{P}{P+N}\left(1-\frac{P}{P+N}\right)$	$\frac{\partial\pi}{\partial x_i}=\frac{2(P-N)}{(P+N)^3}\cdot\left(P\cdot\frac{\partial N}{\partial x_i}-N\cdot\frac{\partial P}{\partial x_i}\right)$

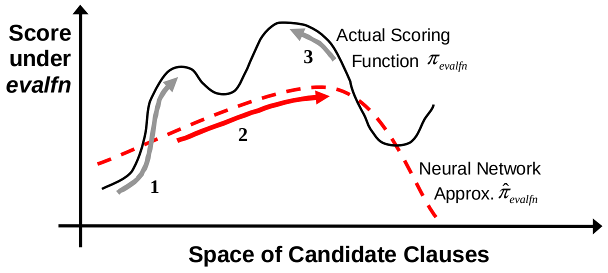

Score under *evalfn*
Actual Scoring Function π_{evalfn}
Neural Network Approx. $\hat{\pi}_{evalfn}$
1
2
3
Space of Candidate Clauses

Figure 3. This graphic illustrates our algorithm using stochastic gradient ascent on the surface defined by the neural network to escaping a local minima in ILP's standard best-first search. The search alternates between periods of ILP's best-first search (1 and 3), and stochastic gradient ascent on the network-defined surface (2). The only difference between this variant and our biased-RRR search is the *starting point* of the stochastic gradient ascent. Biased-RRR begins each period of stochastic gradient ascent at a *random* point in search space.

random point, the *starting* point for our network-guided gradient ascent is the *ending* point from the previous period of ILP's standard search (on real data). That is, in this variation we rapidly alternate between brief periods of ILP's standard (best-first) search and stochastic gradient ascent on the neural-network-defined surface. This variation is illustrated in Figure 3.

This idea of intelligent rapid random restarts to escape local maxima is not a new one. Though not in the domain of ILP, Boyan and Moore's STAGE algorithm [23] use quadratic regression to approximate search "trajectories." That is, they learn a function mapping points in feature space to the endpoint of a 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.

4.3 Extracting Concepts from the Function Approximation

Finally, we can extract concepts from the neural network itself. Craven and Shavlik [26] have developed a method to extract a decision tree from a trained neural network. Running their algorithm on the (thresholded) trained clause-evaluation approximator would produce a theory – a set of clauses – that we could variablize and score on the actual data set.

The neural network, in fitting a nonlinear surface to the scoring function, will hopefully find pairs and triplets of terms that – while individually not helpful – lead to a highly accurate rule when combined. Two terms that share one or more variables and are connected to the same single hidden unit via a highly-weighted edge that possibly have an impact on the accuracy of the rule when taken together. Such a pair of terms is a likely candidate for terms of an invented predicate. The neural network approximation could be used to find such predicates using only one or a few seeds; then the invented predicates could be added to the background knowledge for the search over the remaining seeds' subsumption lattices.

5 Results and Discussion

This section presents our results on several benchmark datasets. We first show that the neural network is indeed capable of learning an approximation to the clause evaluation function. We then use the network in a rapid-random-restart search to bias restarts towards more promising regions of search space, as described in Section 4.2.

5.1 Benchmark Dataset Overview

We tested clause evaluation function approximation on four standard ILP benchmark datasets. The tasks included predicting mutagenic activity [27] and carcinogenic activity [28] in compounds, predicting the smuggling of nuclear and radioactive materials, and predicting metabolic 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.

5.2 Learning the Clause Evaluation Function

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 online learning process outlined in Section 3.2.

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 standard scoring metric, a *variant of Aleph's compression* heuristic; that is, a clause's score is given by

$$\text{score} = \frac{(\text{pos. exs. covered}) + (\text{neg. exs. covered}) - (\text{clauselength}) + 1}{(\text{total pos. exs.})} \qquad (\mathbf{6})$$

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 [29], 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. 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 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.)

For all four datasets, the hypothesis evaluation function $\boldsymbol{\pi}^{E}_{evalfn}$ 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.

So far, we have assumed no transfer of knowledge between seed examples, i.e., we learn a new neural network from scratch 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 $\vec{x}$ described in Section 3) 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 garnered from previous rules.

Consequently, we looked at the contribution of each subset of features on each of the four datasets. In particular, we wanted to see how instance-independent features contributed to the learning task. As before, we used WEKA to construct two learning curves for each dataset. These two learning curves correspond to training the network on (1) only instance-independent features, and (2) only instance-dependent features.

As Figure 4 illustrates, 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.

Although these graphs illustrate that we are capable of *learning* the clause evaluation function, they do not show the degree to which the function is learned. Figure 5 compares the RMS error of the network approximation to the RMS error obtained by using a random sampling of *training examples* to approximately score clauses. This provides an alternate method for computation reduction against which we compare our method. It also allows us to determine the number of evaluations the neural network is "worth." This number varies significantly across the four datasets, ranging from between 25% and 50% sampling to well beyond 90% sampling. As these are all fairly small benchmark datasets, it remains an open question how our method will compare to sampling the training examples in larger problems (with both larger hypothesis spaces as well as datasets). This includes large problems that often arise in the biological sciences and text extraction [30].

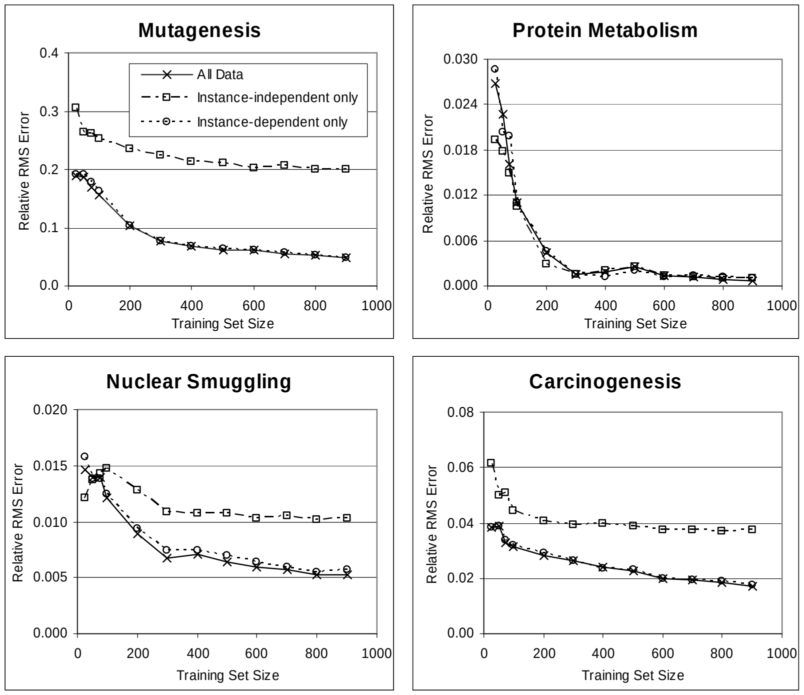


Figure 4. Learning curves showing *test-set* accuracy over four domains comparing the roles of instance-dependent versus instance-independent features. Learning curves were generated only using a subset of the complete set of features, and the results were compared to the case where all features were used to train the network.

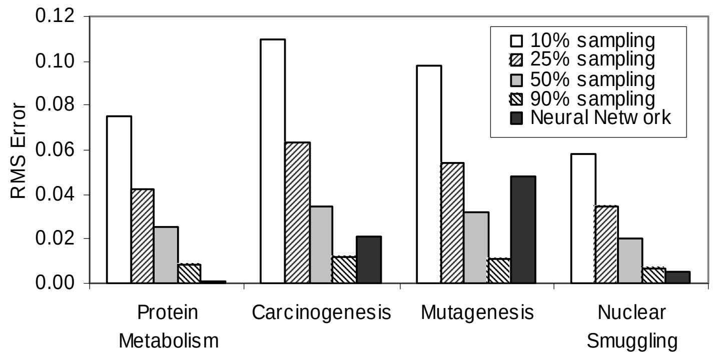


Figure 5. Comparing the RMS error of the neural-network approximation with that obtained by using a random sampling of *training examples* to approximate clauses. The error of the neural-network approximation varies widely, but in all cases does better than a 25% sampling of examples, and for two of the four datasets, does better than a 90% sampling.

5.3 Using the Evaluation Function Approximator to Guide Random Search

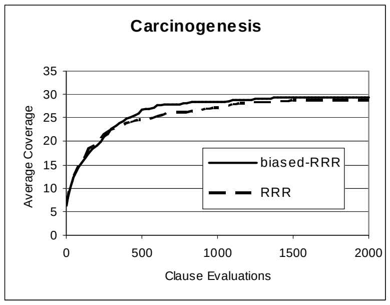
This section details the use of trained neural network to bias the random restarts in a rapid random restart search. Our goal here is to find the best-scoring clause in the subsumption lattice using as few clause evaluations as possible. Thus, results in this section are only concerned with maximizing some evaluation function over the *training data*. Assuming a well-designed evaluation function, this corresponds with good test-set performance.

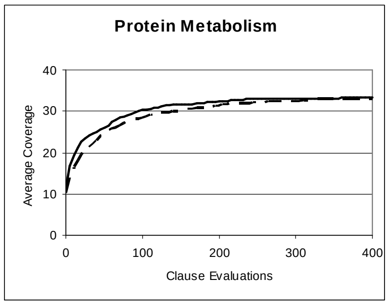
We implemented the previous-described online learning algorithm in *Aleph*. To enable biased random restarts, we also implemented a stochastic gradient ascent algorithm. Our gradient ascent implementation, at each step, only considered flipping an input bit on or off, and did not allow flipping a bit on if the clause length was already at its maximum. The probability of a bit flip of input x_i is given by:

$$P(\text{flip } x_i) = \frac{1}{Z}\exp\left(\frac{1}{\sigma_{\mathbf{x}}^2}\cdot\frac{\partial \hat{\pi}_{evalfn}^{E}}{\partial x_i}\right)\cdot(-1)^{x_i} \qquad \textbf{(7)}$$

In this formula, σ^2 determines the "softness" of the gradient ascent. For our results, it was set such that we were 100 times more likely to flip the "best" literal than the "worst." The $(-1)^x$ term simply flips the sign of the gradient when we consider flipping a bit *off* (since this is a move in the negative direction).

In order to test the performance of our algorithm, we attempt to find the clause that maximizes the *coverage* scoring function, defined as the number of positive examples covered minus the number of negative examples covered. We used stochastic gradient ascent to bias RRR search towards with 1000 restarts and 10 steps per restart, and compare the biased-RRR versus normal RRR with the same parameters. For the biased-RRR, the "burn-in period" consisted of a single random restart and the local moves following. We report results on three of the four datasets from the previous





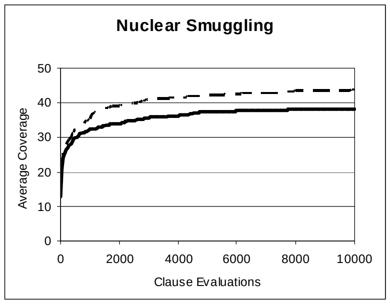


Figure 6. Performance of the biased-RRR search versus a traditional RRR search. The *x*-axis shows the number of clauses evaluated, and the *y*-axis displays the average coverage of the best clause found at that *x* value. For carcinogenesis and protein metabolism, the biased-RRR performs better, but for nuclear smuggling it is clearly outperformed.

section, omitting mutagenesis as it too quickly converges: over 80% of seeds found their best clause in the very first restart. For each dataset we explored the subsumption lattices of 100 different seed examples. Our neural network consisted of 10 hidden units. Finally, each rapid random restart began at the endpoint of the previous local search and finished after a fixed number of random steps. *Aleph* search parameters are left at default whenever possible.

Figure 6 shows the results for each of the three datasets. In each of the three graphs, the *x*-axis shows the number of clauses evaluated, and the *y*-axis shows the average coverage over all seeds of the best example found thus far. As the plots show, for two of the datasets – carcinogenesis and protein metabolism – biased-RRR found a better clause quicker than did traditional RRR. However, in the third task, nuclear smuggling, biased-RRR did worse than the default implementation. The reasons for this are unclear, as the neural network was clearly able to learn the evaluation function approximator in this domain.

6 Conclusion and Future Work

We demonstrated that the use of a neural network for clause evaluation is a useful 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 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) 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. This strategy can be used to decrease the runtime of ILP systems on large tasks.

The most pressing work that remains is implementing and evaluating the other strategies for taking advantage of the clause-evaluation approximator outlined in Sections 4.1 and 4.3. Clearly accuracy is lost in approximating the clause-evaluation function, but it is difficult to determine how it affects solutions generated by using it to quickly evaluate clauses in a typical ILP search. Another open question is whether useful information can be extracted from the trained neural network itself [26].

Also, Botta *et al.* [31] have characterized hypothesis space, discovering a critical region they have named the *phase transition*. In this critical region, the computational complexity of inference increases, and clauses generated in this region tend to have poor generalization to unseen test examples. This phase transition is a difficult region for ILP algorithms; our algorithm's performance here specifically needs exploration.

Finally, we have discussed learning the evaluation approximation in a least-squared-error sense. However, what may be more important for ILP is the relative *ranking* of candidate clauses. Thus, an approach like Caruana and Baluja's *Rankprop* algorithm [32] – an alternative to backprop concerned with correctly predicting the ranking of the output variables – may be more natural.

7 Acknowledgements

This work was supported by National Library of Medicine (NLM) grant 1T15 LM007359-01, DARPA Grant F30602-01-2-0571, United States Air Force Grant F30602-01-2-0571, and NLM grant 1R01 LM07050-01. The authors would also like to thank the UW Condor Team and the anonymous reviewers.

References

1. N. Lavrac & S. Dzeroski (1994). *Inductive Logic Programming.* Ellis Horwood.
2. R. King, S. Muggleton & M. Sternberg (1992). Predicting protein secondary structure using inductive logic programming. *Protein Engineering*, 5:647-657.
3. A. Srinivasan, R. King, S. Muggleton & M. Sternberg (1997). The predictive toxicology evaluation challenge. *Proc. 15th Intl. Joint Conf. on Artificial Intelligence*, 1-6.
4. B. Dolsak & S. Muggleton (1991). The application of ILP to finite element mesh design. *Proc. 1st Intl. Workshop on ILP*, 225-242.
5. J. Zelle & R. Mooney (1993). Learning semantic grammars with constructive inductive logic programming. *Proc. 11th Natl. Conf. on Artificial Intelligence*, 817-822.
6. I. Bratko & M. Grobelnik (1993). Inductive learning applied to program construction and verification. *Proc. 3rd Intl. Workshop on Inductive Logic Programming*, 169-182.
7. S. Nienhuys-Cheng & R. de Wolf (1997). *Foundations of Inductive Logic Programming.* Springer-Verlag.
8. M. Schmidt-Schauss (1988). Implication of clauses is undecidable. *Theoretical Computer Science*, 59:287-296.

9. J. Quinlan (1990). Learning logical definitions from relations. *Machine Learning*, 239-266.
10. S. Muggleton & C. Feng (1990). Efficient induction of logic programs. *Proc. 1st Conf. on Algorithmic Learning Theory*, 368-381.
11. S. Muggleton (1995). Inverse Entailment and Progol. *New Generation Computing*, 13:245-286.
12. 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.
13. F. Zelezny, A. Srinivasan & D. Page (2002). Lattice-search runtime distributions may be heavy-tailed. *Proc. 12th Intl. Conf. on Inductive Logic Programming*, 333-345.
14. A. Giordana, L. Saitta & F. Zini (1994). Learning disjunctive concepts by means of genetic algorithms. *Proc. 11th Intl. Conf. on Machine Learning*, 96-104.
15. P. Hanschke & J. Wurtz (1993). Satisfiability of the smallest binary program. *Info. Proc. Letters*, 496:237-241.
16. E. Dantsin, T. Eiter, G. Gottlob & A. Voronkov (2001). Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33:374-425.
17. U. Rückert & S. Kramer (2003). Stochastic local search in k-term DNF learning. *Proc. 20th Intl. Conf. on Machine Learning*, 648-655.
18. H. Blockeel, L. Dehasp, B. Demoen, G. Janssens, J. Ramon & H. Vandecasteele (2002). Improving the efficiency of inductive logic programming through the use of query packs. *J. AI Research*, 16:135-166.
19. V. Santos Costa, A. Srinivasan, R. Camacho, H. Blockeel, B. Demoen, G. Janssens, J. Struyf, H. Vandecasteele & W. Van Laer (2003). Query transformations for improving the efficiency of ILP systems, *J. Machine Learning Research,* 4:465-491.
20. A. Srinivasan (1999). A study of two sampling methods for analysing large datasets with ILP. *Data Mining and Knowledge Discovery*, 3:95-123.
21. M. Sebag & C. Rouveirol (2000). Resource-bounded relational reasoning: induction and deduction through stochastic matching. *Machine Learning*, 38:41-62.
22. J. Maloberti & M. Sebag (2001). Theta-subsumption in a constraint satisfaction perspective. *Proc. 11th Intl. Conf. on Inductive Logic Programming*, 164-178.
23. J. Boyan & A. Moore (2000). Learning evaluation functions to improve optimization by local search. *J. Machine Learning Research*, 1:77-112.
24. K. Hornik, M. Stinchcombe & H. White (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2:359-366.
25. D. Nix & A. Weigend (1995). Learning local error bars for nonlinear regression. *Advances in Neural Information Processing Systems*. MIT Press.
26. M. Craven & J. Shavlik (1995). Extracting tree-structured representations of trained networks. *Advances in Neural Information Processing Systems.* MIT Press.
27. R. King, S. Muggleton, A. Srinivasan & M. Sternberg (1996). Structure-activity relationships derived by machine learning. *PNAS*, 93:438-442.
28. A. Srinivasan, R. King, S. Muggleton & M. Sternberg (1997). Carcinogenesis predictions using ILP. *Proc. 7th Intl. Workshop on Inductive Logic Programming*, 273-287.
29. I. Witten & E. Frank (1999). *Data Mining.* Morgan Kaufmann Publishers.
30. M. Goadrich, L. Oliphant & J. Shavlik (2004). Learning ensembles of first-order clauses for recall-precision curves: a case study in biomedical information extraction. *Proc. 14th Intl. Conf. on Inductive Logic Programming.*
31. M. Botta, A. Giordana, L. Saitta & M. Sebag (2003). Relational learning as search in a critical region. *J. Machine Learning Research*, 4:431-463.
32. R. Caruana & S. Baluja (1996). Using the future to 'sort out' the present. *Advances in Neural Information Processing Systems.* MIT Press.