Learning Markov Logic Networks via Functional Gradient Boosting

Tushar Khot*, Sriraam Natarajan†, Kristian Kersting‡ and Jude Shavlik§
*University of Wisconsin-Madison, USA. tushar@cs.wisc.edu
†Wake Forest University School of Medicine, USA. snataraj@wfubmc.edu
‡Fraunhofer IAIS, Germany. kristian.kersting@iais.fraunhofer.de
§University of Wisconsin-Madison, USA. shavlik@cs.wisc.edu

***Abstract*—Recent years have seen a surge of interest in Statistical Relational Learning (SRL) models that combine logic with probabilities. One prominent example is Markov Logic Networks (MLNs). While MLNs are indeed highly expressive, this expressiveness comes at a cost. Learning MLNs is a hard problem and therefore has attracted much interest in the SRL community. Current methods for learning MLNs follow a two-step approach: first, perform a search through the space of possible clauses and then learn appropriate weights for these clauses. We propose to take a different approach, namely to learn both the weights and the structure of the MLN simultaneously. Our approach is based on functional gradient boosting where the problem of learning MLNs is turned into a series of relational functional approximation problems. We use two kinds of representations for the gradients: clause-based and tree-based. Our experimental evaluation on several benchmark data sets demonstrates that our new approach can learn MLNs as good or better than those found with state-of-the-art methods, but often in a fraction of the time.**

INTRODUCTION

In recent years, there has been an increasing interest in addressing challenging problems that involve rich relational and noisy data. Fueled by this, several Statistical Relational Learning methods [1] have been proposed that combine the expressiveness of first-order logic and the ability of probability theory to handle uncertainty. These models range from directed models [2]–[4] to undirected models [5], [6] and sampling-based approaches [7], [8]. The advantage of these models is that they can succinctly represent probabilistic dependencies among the attributes of different related objects, leading to a compact representation of learned models.

While these models are highly attractive due to their compactness and comprehensibility, the problem of learning in these models is computationally intensive. This is particularly true for Markov Logic Networks (MLNs) [5]. MLNs extend Markov networks to the relational setting by expressing domain knowledge as a set of weighted logic formulas. One of the nice features of MLNs is that they allow the user to write many rules about the domain and then learn weights for the rules to perform inference.

But the task of learning the rules themselves is an important and challenging task and has received much attention lately [9]–[12]. Bottom-up structure learning [10] uses a propositional Markov network learning algorithm to identify paths of ground atoms. These form the templates that are generalized into first-order formulas. Hypergraph lifting [11] on the other hand clusters the constants and true atoms to construct a lifted (first-order) graph. Relational path-finding on this hypergraph is used to obtain the MLN clauses. Structural motif learning [12] uses random walks on the ground network to find symmetrical paths and cluster nodes with similar path distributions. All the methods obtain the candidate clauses first, learn the weights and modify the clauses accordingly.

We propose a different route. We present a MLN-learning approach that learns the weights and the clauses simultaneously. Specifically, we turn the problem of learning MLNs into a series of relational regression problems by using Friedman's functional gradient boosting algorithm [13]. The key idea in this algorithm is to consider the target potential function as a series of regression trees learned in a stage-wise manner. The functional gradient approach has produced state of the art results in building relational dependency networks over many domains [14] and has been employed in other relational learning problems such as relational CRFs [15], relational policies [16], relational sequences [17], etc.

We use two kinds of representations for the functional gradients on the pseudo-likelihood for MLNs: *clauses* and *trees*. The former version simply learns a set of clauses at each gradient step, each with an associated regression value, while the latter version views MLNs as a set of relational regression trees. The regression values are the weights on the MLN clauses. Finally, we compare them against state-of-the-art algorithms in four different standard SRL testbeds. The experimental results demonstrate the superior performance of boosting, both in terms of time and accuracy of the learned model. Our approach also has the advantage of learning more predictive rules than the current MLN structure-learning algorithms. As we show empirically, in spite of learning more rules, our algorithms have shorter running times compared to the state-of-the-art MLN algorithms.

Compared to existing MLN learning approaches, boosting MLNs has the following benefits: (1) The number of clauses grows with the number of training episodes, in turn increasing the size of the MLN only as needed; (2) It learns both the clauses and weights simultaneously; (3)

Because most off-the-shelf relational regression tree/clause learners can be used for this purpose, it is a flexible learning algorithm for MLNs. In fact, (4) viewing MLNs as a set of regression functions itself is a significant contribution. This view allows one to treat learning MLNs as boosting a set of conditional distributions. Intuitively, we view an MLN as a set of Horn clauses per query predicate. In turn, this suggests one can generalize efficient boosting approaches recently developed for rapid learning of Relational Dependency Networks (RDNs) [14] to MLNs.

The rest of the paper is organized as follows: we first introduce the necessary background on MLNs and functional gradients. Next, we derive the functional gradient for MLNs and present our two new learning methods. We then present our empirical evaluation in four different testbeds and finally conclude by outlining some areas for future research.

MARKOV LOGIC NETWORKS

One of the most popular and general SRL representations is *Markov Logic Networks (MLNs)* [5]. An MLN consists of a set of formulas in first-order logic and their real-valued weights, $\{(w_i, f_i)\}$. Together with a set of constants, we can instantiate an MLN as a Markov network with a node for each ground predicate (atom) and a feature for each ground formula. All groundings of the same formula are assigned the same weight, leading to the following joint probability distribution over all atoms:

$$P(X = x) = \frac{1}{Z} \exp\left(\sum_i w_i n_i(x)\right) \qquad (1)$$

where $n_i(x)$ is the number of times the ith formula is satisfied by possible world x and Z is a normalization constant (as in Markov networks). Intuitively, a possible world where formula f_i is true one more time than a different possible world is e^{w_i} times as probable, all other things being equal.

FUNCTIONAL GRADIENT BOOSTING OF MLNS

Functional gradient methods have been used previously to train conditional random fields (CRF) [18] and their relational extension (TILDE-CRF) [15], to learn relational policies [16], to learn to label relational sequences [17], and, as mentioned above, to train RDNs [14].

Assume that the training examples are of the form $(\mathbf{x}_i, y_i)$ for $i = 1, ..., N$ and $y_i \in \{0, 1\}$. The goal is to fit a model $P(y|\mathbf{x}) \propto e^{\psi(y,\mathbf{x})}$. A standard approach for learning such models is based on gradient-descent, where the learning algorithm starts with initial parameters θ_0 and computes the gradient of the likelihood function. Dietterich et al. [18] used a more general approach to train the potential functions based on Friedman's [13] gradient-tree boosting algorithm where the potential functions are represented by sums of regression trees that are grown stage-wise. More formally, *Functional Gradient Ascent* (FGA) starts with an initial potential ψ_0 and iteratively adds gradients Δ_i. After m steps, the potential is given by $\psi_m = \psi_0 + \Delta_1 + ... + \Delta_m$. Here, Δ_m, the functional gradient at episode m is

$$\Delta_m = \eta_m \times E_{x,y}[\partial/\partial\psi_{m-1}\; log\, P(y|x; \psi_{m-1})] \qquad (2)$$

η_m is the learning rate and $\partial/\partial\psi_{m-1}$ is used to represent the partial derivative with respect to ψ_{m-1}. Dietterich et al. suggested evaluating the gradient at every position in every example and fitting a regression tree to the derived examples.

FGA is different from the standard gradient ascent methods in one key aspect - it does not assume a linear parameterization for the potential function. The standard assumption is that the potential function ψ is represented as $\psi = \sum \beta_i f_i$ where $\{\beta_1, ..., \beta_n\} = \theta$ are the parameters of ψ. In FGA, the assumption is more general in that ψ is a weighted sum of functions (as shown earlier) and the gradient is given by Equation 2. As Dietterich et al. point out, the expectation $E_{x,y}[..]$ cannot be computed as the joint distribution $P(\mathbf{x}, \mathbf{y})$ is unknown. Since the joint distribution is unknown, FGA methods treat the data as a surrogate for the joint distribution. Hence, instead of computing the functional gradient over the potential function, the functional gradients are computed for each training example, i.e.,

$$\Delta_m(y_i; \mathbf{x}_i) = \nabla_\psi \sum_i log(P(y_i|\mathbf{x}_i; \psi))|_{\psi_{m-1}} \qquad (3)$$

These are point-wise gradients of the potential ψ computed in the model from the previous iteration using the potential ψ_{m-1}. Now this set of local gradients form a set of training examples for the gradient at stage m. The key step in *functional gradient boosting* (*FGB*) is the fitting of a regression function (typically a regression tree) h_m on the training examples $[(\mathbf{x}_i, y_i), \Delta_m(y_i; \mathbf{x}_i)]$ [13]. Dietterich et al. [18] point out that although the fitted function h_m is not exactly the same as the desired Δ_m, it will point in the same direction (assuming that there are enough training examples). So ascent in the direction of h_m will approximate the true functional gradient.

For the rest of the paper, we denote variables using capitalized letters, values as small letters and sets using bold-faced letters. Also, since there is a one-to-one mapping from x_i to y_i, we just use x_i to indicate an example and its label. P($x_i = 1$) is the probability of example x_i being true, whereas P(x_i) is the probability of the example having the same label as provided in the data.

Derivation of the Functional Gradient

The joint likelihood of MLNs (given in Equation 1) is hard to optimize due to the normalization constant. Hence, we take the standard approach of optimizing the pseudo-likelihood (PL) that is given by,

$$PL(\mathbf{X} = \mathbf{x}) \quad = \textstyle\prod_{x_i \in \mathbf{x}} P(x_i|\mathbf{MB}(x_i)) \qquad (4)$$

where $\mathbf{MB}(x_i)$ is the Markov blanket.

$$P(x_i = 1|\mathbf{MB}(x_i)) = \frac{\exp\left(\sum_j w_j n_j(x_i{=}1; MB(x_i))\right)}{\sum_{x'} \exp\left(\sum_j w_j n_j(x_i{=}x'; MB(x_i))\right)} = \frac{\exp\left(\sum_j w_j nt_j(x_i; \mathbf{MB}(x_i))\right)}{\exp\left(\sum_j w_j nt_j(x_i; \mathbf{MB}(x_i))\right)+1} \quad (5)$$

$$\text{where } nt_j(x_i; \mathbf{MB}(x_i)) = n_j(x_i = 1; \mathbf{MB}(x_i)) - n_j(x_i = 0; \mathbf{MB}(x_i)) \quad (6)$$

$n_j(\mathbf{x})$ is the number of groundings of clause C_j given $\mathbf{x}$. $nt_j(x_i; \mathbf{MB}(x_i))$ corresponds to the non-trivial groundings of an example (explained below), x_i given its Markov blanket [19]. Hence we can define the potential functions

$$\psi(x_i; \mathbf{MB}(x_i)) = \sum_j w_j nt_j(x_i; \mathbf{MB}(x_i)) \quad (7)$$

$\psi(x_i; \mathbf{MB}(x_i))$ is the potential function of x_i given all other $x_j \neq x_i$ and $(x_j \in \mathbf{x})$. Hence Equation 5 becomes,

$$P(x_i = 1|\mathbf{MB}(x_i)) = \frac{\exp\left(\psi(x_i; \mathbf{MB}(x_i))\right)}{\exp\left(\psi(x_i; \mathbf{MB}(x_i))\right) + 1}$$

As mentioned earlier, we optimize the pseudo-log-likelihood,

$$PLL(\mathbf{X} = \mathbf{x}) = \sum_{x_i \in \mathbf{x}} \log P(x_i|\mathbf{MB}(x_i)) \quad (8)$$

Taking the derivative of PLL w.r.t. the function ψ, we get

$$\frac{\partial PLL(\mathbf{X}{=}\mathbf{x})}{\partial \psi(x_i{=}1; \mathbf{MB}(\mathbf{x_i}))} = \frac{\partial \log P(x_i; \mathbf{MB}(\mathbf{x_i}))}{\partial \psi(x_i{=}1; \mathbf{MB}(\mathbf{x_i}))} = I(x_i = 1; \mathbf{MB}(\mathbf{x_i})) - P(x_i = 1; \mathbf{MB}(\mathbf{x_i})) \square \quad (9)$$

Note that the gradient at each example is now simply the adjustment required for the probabilities to match the observed value (x_i) for that example. This gradient serves as the weight for the current regression example at the next training episode.

The expression in Equation 9 is very similar to the one in Dietterich et al. [18]. The key feature of the above expression is that the functional gradient for each example is dependent on the observed value. If the example is positive, the gradient $(I - P)$ is positive indicating that the model should increase the probability of the ground predicate being true. On the contrary if the example is negative, the gradient is negative, implying that it will push the probability towards 0.

Our algorithm is restricted to learn only non-recursive Horn clauses, but extended to allow negation-by-failure. Despite this restriction, we are able to perform better than other structure-learning algorithms as shown in our experiments. For notational ease, we will represent a Horn clause, say,

$$p_1(\mathbf{X}_1) \wedge \ldots \wedge p_c(\mathbf{X}_c) \rightarrow target(\mathbf{X}')$$
$$\text{as} \quad \wedge_k\, p_k(\mathbf{X}_k) \rightarrow target(\mathbf{X}')$$

where $\mathbf{X}_k$ are the arguments for p_k and $\mathbf{X}' \subseteq \cup_k \mathbf{X}_k$.

Recall the definition of $nt_j(x_i)$ given in Equation 6. When x_i only appears in the head of the clause C_j, $nt_j(x_i)$ corresponds to the number of non-trivial groundings for x_i that satisfy the Horn clause C_j. Groundings of C_j that remain true irrespective of the truth value of x_i are defined as the *trivial* groundings for x_i that satisfy C_j, while others are called as the *non-trivial* groundings. So the non-trivial groundings for satisfying a clause $\wedge_k p_k(\mathbf{X}_k) \rightarrow target(\mathbf{X}')$ would correspond to the groundings for $\cup_k \mathbf{X}_k$ that satisfy the body of the clause i.e. $\wedge_k p_k(\mathbf{x}_k) = true$, after unifying the head of the clause ($target(\mathbf{X}')$) with the example, x_i. For a more detailed discussion of non-trivial groundings, see Shavlik and Natarajan [19].

We made two assumptions in our model: (1) Every ground literal does not have any other ground literal with the same predicate in its Markov blanket. (2) The Markov blanket is completely observed during training, i.e. there is no missing data. Also we consider only a single target predicate while learning a regression tree, but we learn a joint model that uses the regression trees learned for all the predicates. We explain the joint model learning in the Algorithm section.

Representation of Functional Gradients for MLNs

Our goal is to find $\hat{\psi}$ such that the squared error between $\hat{\psi}$ and the functional gradient is minimized. i.e.,

$$arg \min_{\hat{\psi}} \sum_{i=1}^{n} (\hat{\psi}(x_i; MB(x_i)) - \Delta(x_i))^2 \quad (10)$$

over all examples. We present our two representations of $\hat{\psi}$s: *trees* and *clauses*. For our first representation, we use a relational regression tree learner [20] to fit the gradients on each example. In order to do so, we modified the splitting criterion at each node to be the one presented below. Each path from the root to a leaf can be seen as a clause and the weigh ton the leaf correspond to the weight of the clause. As an example, let us consider the literal $q(\mathbf{X}'')$ to be added to the tree at a node N. Let the current clause formed by the path from the root to the node N be $\wedge_k p_k(\mathbf{X}_k) \rightarrow target(\mathbf{X}')$. So adding $q(\mathbf{X}'')$ would split the current clause to two clauses,

$$C_1 : \wedge_k p_k(\mathbf{X}_k) \wedge q(\mathbf{X}'') \rightarrow target(\mathbf{X}')$$
$$C_2 : \wedge_k p_k(\mathbf{X}_k) \wedge \forall \mathbf{X}_f, \neg q(\mathbf{X}'') \rightarrow target(\mathbf{X}')$$

where $\mathbf{X}_f$ are the free variables in $q(\mathbf{X}'')$. For all the examples that reach the node N, assume $\mathcal{I}$ to be the set of examples that satisfy $q(\mathbf{X}'')$ and $\mathcal{J}$ be the ones that do not. Let w_1 and w_2 be the regression values that would be assigned to C_1 and C_2 respectively. Let $n_{x,1}$ and $n_{x,2}$ be the number of non-trivial groundings for an example x with clauses C_1 and C_2. The regression value returned for an example would depend on whether it belongs to $\mathcal{I}$ or $\mathcal{J}$. Now,

$$\hat{\psi}(x_i) = n_{x_i,1} \cdot w_1 \cdot I(x_i \in \mathcal{I}) + n_{x_i,2} \cdot w_2 \cdot I(x_i \in \mathcal{J}) \quad (11)$$

and the squared error is

$$SE = \sum_{x\in\mathcal{I}}[n_{x,1}\cdot w_1-\Delta_x]^2+\sum_{x\in\mathcal{J}}[n_{x,2}\cdot w_2-\Delta_x]^2$$

$$\begin{aligned}\frac{\partial}{\partial w_1}SE &= \sum_{x\in\mathcal{I}}2\cdot[n_{x,1}\cdot w_1-\Delta_x]\cdot n_{x,1}+0=0\\ w_1 &= \frac{\sum_{x\in\mathcal{I}}\Delta_x\cdot n_{x,1}}{\sum_{x\in\mathcal{I}}n_{x,1}^2}\\ \frac{\partial}{\partial w_2}SE &= 0+\sum_{x\in\mathcal{J}}2\cdot[n_{x,2}\cdot w_2-\Delta_x]\cdot n_{x,2}=0\\ w_2 &= \frac{\sum_{x\in\mathcal{J}}\Delta_x\cdot n_{x,2}}{\sum_{x\in\mathcal{J}}n_{x,2}^2}\end{aligned}$$

When adding each literal to the clause, we greedily search for the literal that minimizes this squared error. The false branch at every node with condition $C(\mathbf{X})$, would be converted to $\forall\mathbf{X}_f,\neg C(\mathbf{X})$ which in its CNF form becomes $\exists\mathbf{X}_f,C(\mathbf{X})$, where $\mathbf{X}_f\subset\mathbf{X}$ are the free variables in $C(\mathbf{X})$. This can result in a large clique in the grounded Markov Network. To avoid this, we maintain an ordered list of clauses and return the weight for the first clause that has at least one grounding for a given example. We can then ignore the condition on a given node, if the false branch is picked in the path to the leaf. It is worth noting that $C(\mathbf{X})$ maybe a conjunction of literals depending on the maximum number of literals allowed at an inner node.

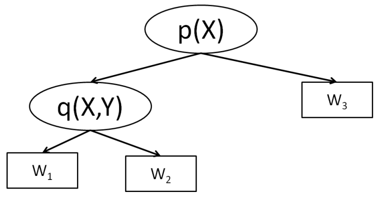

Figure 1. Example tree for target(X).

Figure 1 gives an example regression tree for $target(X)$. If we are scoring the node $q(X,Y)$, we would split all the examples that satisfy $p(X)$ into two sets $\mathcal{I}$ and $\mathcal{J}$. $\mathcal{I}$ would contain all examples that have at least one grounding for $q(X,Y)$ and $\mathcal{J}$ would contain the rest; $target(x_1)$ would be in $\mathcal{I}$ if $p(x_1)\wedge q(x_1,Y)$ is true and $target(x_2)$ would be in $\mathcal{J}$, if $p(x_2)\wedge(\forall Y,\neg q(x_2,Y))$ is true. The parameter $n_{x_1,1}$ corresponds to the number of groundings of $p(x_1)\wedge q(x_1,Y)$, while $n_{x_2,2}$ corresponds to the number of groundings of $p(x_2)\wedge(\forall Y,\neg q(X_2,Y)$. The corresponding ordered list of MLN rules is:

$$\begin{aligned}w_1 &: p(X),q(X,Y)\rightarrow target(X)\\ w_2 &: p(X)\rightarrow target(X)\\ w_3 &: target(X)\end{aligned}$$

For our second representation, we learn Horn clauses by using a beam search that adds literals to clauses that reduce the squared error. We maintain a (beam-size limited) list of clauses ordered by their squared error and keep expanding clauses from the top of the list. We add clauses as long as their lengths do not exceed a threshold and the beam still contains clauses. We recommend using clauses when the negation-by-failures introduced by the trees would make the inference step too slow.

Hence, we replace the tree with a set of clauses learned independently at each gradient-step. Since we do not have two branches when every new condition is added, the error function becomes:

$$\begin{aligned}SE &= \sum_{x\in\mathcal{I}}[n_{x,1}\cdot w-\Delta_x]^2+\sum_{x\in\mathcal{J}}\Delta_x^2\\ \Longrightarrow w &= \frac{\sum_{x\in\mathcal{I}}\Delta_x\cdot n_{x,1}}{\sum_{x\in\mathcal{I}}n_{x,1}^2}\end{aligned}$$

Note that the key change is that we do not split the nodes and instead just search for new literals to add to the current set of clauses. Hence, instead of an ordered list for each gradient step, we learn a pre-set number of clauses (C). We use a similar parameter for the regression-tree learner as well with a pre-set number of leaves (L). The values of C and L are fixed at 3 and 8 respectively for all our experiments. Hence, the depth of tree is quite small and so is the number of Horn clauses per gradient-step.

Before presenting the algorithmic details, we summarize our strengths. Apart from learning the structure and weight simultaneously, functional gradient boosting approach has other key advantages: (1) Our models are essentially weighted Horn clauses. This makes the inference process easier, especially given that we use the procedure presented in Shavlik and Natarajan [19] to keep track of the non-trivial groundings for a clause/predicate. (2) Our learning algorithms can use prior knowledge as an initial set of MLN clauses and learn more clauses as needed to minimize the error on a training set.

Algorithm for Learning MLNs

Functional gradient boosting of MLNs with both the tree and the clause learner is presented in Algorithm 1. In $TreeBoostForMLNs$, we iterate through M gradient steps and in each gradient step learn a regression tree for the target predicates one at a time. We create examples for the regression learner for a given predicate, P using the $GenExamples$ method. We use the function $FitRelRegressionTree(S,P,L)$ to learn a tree that best fits these examples. We limit our trees to have maximum L leaves and greedily pick the best node to expand. In our experiments, we set $L=8$ and $M=20$. $FitRelRegressionClause(S,P,N,B)$ can be called here to learn clauses instead. N is the maximum length

Algorithm 1 MLN-Boost: FGB for MLN's

```
1: function TreeBoostForMLNs($Data$)
2:     for $1 \leq m \leq M$ do                          ▷ M gradient steps
3:         $F_m := F_{m-1}$
4:         for P in T do ▷ Iterate through target predicates
5:             $S := GenExamples(Data; F_{m-1}, P)$
6:             $\Delta_m := FitRelRegressTree(S, P, L)$ ▷ FG
7:             $F_m := F_m + \Delta_m$                    ▷ Update models
8:         end for
9:     end for
10:    $P(x_i|\mathbf{MB}(x_i)) \propto \psi$ ▷ Obtained by grounding $F_M$
11: end function
12: function FitRelRegressionTree($S, P, L$)
13:    Tree := createTree($P(X)$)
14:    Beam := {root(Tree)}
15:    while $numLeaves(Tree) \leq L$ do
16:        Node := popBack(Beam)▷ Node w/ worst score
17:        C := createChildren(Node)      ▷ Create children
18:        BN := popFront(Sort(C))  ▷ Node w/ best score
19:        addNode(Tree, Node, BN)
20:                                   ▷ Replace Node with BN
21:        insert(Beam, BN.left, BN.left.score)
22:        insert(Beam, BN.right, BN.right.score)
23:    end while
24: return Tree
25: end function
26: function FitRelRegressionClause($S, P, N, B$)
27:    Beam := {P(X)}
28:    BC := P(X)
29:    while $\neg empty(Beam)$ do
30:        Clause := popFront(Beam)▷ Best scoring clause
31:        if $length(Clause) \geq N$ then
32:            continue           ▷ Clause cannot be expanded
33:        end if
34:        C := addLiterals(Clause)
35:        for c $\in$ C do
36:            c.score = SE(c)                    ▷ Squared error
37:            if c.score $\geq$ Clause.score then
38:                insert(Beam, c, c.score)
39:            end if
40:            if c.score $\geq$ BC.score then
41:                BC := c
42:            end if
43:        end for
44:        while length(Beam) $\geq$ B do
45:            popBack(Beam)
46:        end while
47:    end while
48: return BC
49: end function
```

of the clause and B is the maximum beam size. In $FitRelRegressionTree$, we begin with an empty tree that returns a constant value. We use the background predicate definitions (mode specifications) to create the potential literals that can be added ($createChildren$). We pick the best scoring node (based on square error) and replace the current leaf node with the new node ($addNode$). Then both the left and right branch of the new node are added to the potential list of nodes to expand. To avoid overfitting, we only insert and hence expand nodes that have at least 6 examples. We pick the node with the worst score and repeat the process.

The function for learning clauses is shown as $FitRelRegressionClause$ which takes the maximum clause length as the parameter, N (we set this to 3) and beam size, B (we set this to 10). It greedily tries to find the best scoring clause (BC) with $length \leq N$. This method only learns one clause at a time. Hence for learning multiple clauses, we call this function multiple times during one gradient step and update the gradients for each example before each call. In all our experiments we learn a maximum of 3 clauses in a single gradient step.

Learning Joint Models

One of the key features of SRL methods is the ability to learn and reason about predicates and examples jointly. To handle multiple target predicates, we learn a joint model by learning tree/clauses for each predicate in turn. We use the MLN's learned for all the predicates prior to the current iteration to calculate the regression value for the examples. We implement this by learning one tree for every target predicate in line 4 in Algorithm 1. For efficiency, while learning a tree for one target predicate, we do not consider the influence of that tree on other predicates.

Since we use the clauses learned for other predicates to compute the regression value for an example, we have to handle cases where the examples unify with a literal in the body of the clause. Consider the clause, C_j : $p(X), q(X,Y) \rightarrow target(X)$. If we learn a joint model over $target$ and p, this clause will be used to compute the regression value for $p(X)$ in the next iteration. In such a case, the number of non-trivial groundings corresponding to an example, say $p(x)$ for a given grounding ($X = x$) and the given clause would be the number of groundings of $q(x,Y) \wedge \neg target(x)$. Since $p(x)$ appears in the body of the clause, the difference

$$nt_j(p(x)) = [n_j(p(x) = 1) - n_j(p(x) = 0)] \qquad (12)$$

would be negative. As can be seen, $nt_j(p(x))$ is simply the negative of number of non-trivial groundings of $p(x)$ for the above clause. Computing $nt_j(x_i)$ this way allows us to compute the $\hat{\psi}$ values for every example quickly without grounding the entire MLN at every iteration as the number of groundings can be simply negated in some cases.

Experiments

We next compare our two boosting algorithms - tree-based (MLN-BT) and clause-based (MLN-BC) to four state-of-the-art MLN structure learning methods: LHL [11], BUSL [10], Motif-S (short rules) and Motif-L (long rules) [12]) on four standard datasets. In order to make the comparison as fair as possible, we used the following protocol. We employed the default settings of Alchemy [21] for weight learning on all the datasets, unless mentioned otherwise. We set the $multipleDatabases$ flag to true for weight learning. For inference, we used MC-SAT sampling with 1 million sampling steps or 24 hours whichever occurs earlier. For learning structure using motifs, we used the settings provided by Kok and Domingos [12]. While employing LHL and BUSL for structure learning, we used the default settings in Alchemy. We set the maximum number of leaves in MLN-BT to 8 and maximum number of clauses to 3 in MLN-BC. The beam-width was set to 10 and maximum clause length was set to 3 for MLN-BC. We used 20 gradient steps on all the boosting approaches.

In all our experiments, we present MLN-BT and MLN-BC numbers in bold whenever they are statistically significantly better than all the other datasets (except each other). We used the paired t-test with p-value=0.05 for determining statistical significance. Since the Cora and IMDB datasets are much bigger than the UW dataset, we ran the experiments on a heterogenous cluster of machines and hence do not report the learning time for these datasets.

For the Alchemy-based structure-learning algorithms, we tried several different weight learning methods and present the ones with the best results. For instance, Motifs with generative weight learning yielded the best results in the UW dataset, while discriminative weight learning was better in Cora (in the other two domains, they were comparable). Hence, our Motif-S and Motif-L results correspond to generative weight learning in UW and discriminative in Cora. We also ran different weight-learning algorithms such as voted perceptron and conjugate gradient descent from the Alchemy package with every structure-learning algorithm and report the best results. We use the AUC-PR (Area under the Precision-Recall curve) and CLL (Conditional log-likelihood) values to compare the various approaches. We employ AUC-PR as it has been shown to be a more conservative estimate of the learning performance compared to AUC-ROC [22].

A key property of most relational data sets is the number of negative examples. This is also seen in Table I which shows the size of the various datasets used. Since most relations such as `actedIn, cancer, advisedBy` etc. are false in the real-world, the number of negatives can be order of magnitude more than the number of positives. In these cases, simply measuring CLL over the entire data set can be misleading. It can be shown easily that predicting all the examples as the majority class (when the number of examples in one class are far greater than the other) can have a very good CLL value, but a very low AUC-PR value (nearly 0). Hence, considering only CLL (which close to 0 indicates a very good performance) can be misleading in the case of skewed data sets. In fact, a major strength of PR curves is that they ignore the impact of correctly labeling negative examples and instead focus on the typically rarer and yet more important, positive examples. Hence, we not only present the CLL values, but also the AUC-PR values. In addition, we present results where the number of negative examples is twice the number of positives.

Dataset	Types	Predicates	Constants	True literals	Total literals
UW-CSE	9	12	929	2112	260,254
IMDB	3	5	306	1046	17,257
Cora	5	10	3,079	42,558	687,422
WebKB	3	6	1,700	2,065	688,193

Table I
DATASET SIZE

UW Dataset

The goal in the UW data set [23] is to predict the `advisedBy` relationship between a student and a professor. The data set consists of details of professors, students and courses from 5 different sub-areas of computer science (AI, programming languages, theory, system and graphics). Predicates include `professor, student, publication, advisedBy, hasPosition, projectMember, yearsInProgram, courseLevel, taughtBy, teachingAssistant` etc. Our task is to learn using the other predicates, to predict the `advisedBy` relation. We employ 5-fold cross validation where, we learn from four areas and predict on the other area. Apart from the methods describe above, we also compared against the handcoded MLN available on Alchemy's website with discriminative weight learning (shown as *A-D* in the tables). We were unable to get BUSL to run on this data set due to segmentation fault issues and hence we do not report BUSL for this testbed.

Table II presents the AUC and CLL values, along with the training time taken by each method averaged over five-folds. The training time does not change for the different test-sets. As can be seen, for the complete dataset both boosting approaches (MLN-BT and MLN-BC) perform significantly better than other MLN learning techniques on the AUC-PR values. Current MLN learning algorithms on the other hand are able to achieve lower CLL values over the complete dataset by pushing the probabilities to 0, but are not able to differentiate between positive and negative examples as shown by the low AUC-PR values.

When we reduce the negatives in the test set to twice the number of positives, the boosting techniques dominate on both the AUC-PR and CLL values, while the other techniques, which cannot differentiate between the examples, have poor CLL values. Also, there is no significant difference between learning the trees or the clauses in the case of boosting MLNs. We performed additional experiments on

Algo	2X negatives		All negatives		Time
	AUC-PR	CLL	AUC-PR	CLL	
BT	$\mathbf{0.94 \pm 0.06}$	$\mathbf{-0.52 \pm 0.45}$	0.21 ± 0.17	-0.46 ± 0.36	18.4 sec
BC	$\mathbf{0.95 \pm 0.05}$	$\mathbf{-0.30 \pm 0.06}$	0.22 ± 0.17	-0.47 ± 0.14	33.3 sec
M-S	0.43 ± 0.03	-3.23 ± 0.78	0.01 ± 0.00	-0.06 ± 0.03	1.8 hrs
M-L	0.27 ± 0.06	-3.60 ± 0.56	0.01 ± 0.00	-0.07 ± 0.02	10.1 hrs
A-D	0.31 ± 0.10	-3.90 ± 0.41	0.01 ± 0.00	-0.08 ± 0.02	7.1 hrs
LHL	0.42 ± 0.10	-2.94 ± 0.31	0.01 ± 0.01	-0.06 ± 0.02	37.2 sec

Table II
RESULTS ON UW DATA SET. BT = BOOSTING WITH TREES, BC=BOOSTING WITH CLAUSES, M-S=MOTIF WITH SHORT RULES, M-L=MOTIF LONG RULES, A-D=HAND-CODED RULES WITH DISCRIMINATIVE LEARNING, LHL=LIFTED HYPERGRAPH LEARNING

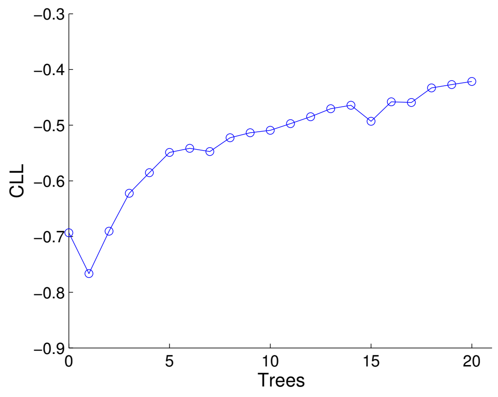

Figure 2. Number of trees used for inference vs. conditional log-likelihood over all examples in the UW-data set.

this data set to understand the impact of number of trees on the performance of the boosting algorithms. Figures 2 and 3 present the CLL and AUC-PR values averaged over 30 runs as a function of the number of trees. As can be seen, CLL values improve as the number of trees increase. This is due to the fact that adding more trees essentially amounts to moving the likelihood of the examples towards 1. On the other hand, the AUC-PR values increase for the first few trees. After a small amount of trees (in this case around 6), the value seem to attain a local optimum. In all our experiments, we observed that increasing the number of trees beyond 20 had no significant impact in AUC-PR values. Our results show that with a small number of trees, the boosting based methods are able to achieve reasonable predictive performance.

Cora Entity Resolution

The Cora dataset, now a standard dataset for citation matching, was first created by Andrew McCallum, later segmented by Bilenko and Mooney [24], and fixed by Poon and Domingos [25][1]. In citation matching, a group is a set of citations that refer to the same paper, and a nontrivial group contains more than one citation [25]. The

[1]Available for download at http://alchemy.cs.washington.edu/papers/-poon07

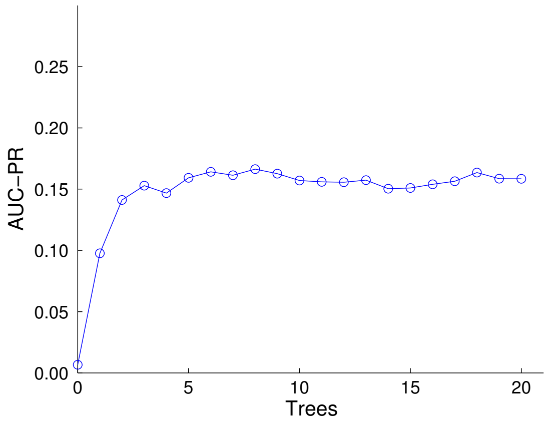

Figure 3. Number of trees used for inference vs. AUC-PR over all examples in the UW-data set.

Cora dataset has 1,295 citations and 134 groups where almost every citation in Cora belongs to a nontrivial group; the largest group contains 54 citations. It contains the predicates: `HasWordAuthor`, `HasWordTitle`, `HasWordVenue`, `Title`, `Venue`, `Author`.

For Cora, we learn a joint model over `SameBib`, `SameVenue`, `SameTitle` and `SameAuthor`. Since this dataset is large, to speedup learning we sampled 25% of the examples during every gradient step for MLN-BT. Similar to the UW dataset, we used a handcoded MLN(B+N+C+T) for Cora presented by Singla et al. [26]. We evaluated all the models jointly over the four target predicates given the evidence predicates. We used the queryEvidence flag for Alchemy weight learning and inference. As with the previous case, we could not get BUSL to run on this data set.

We performed 5-fold cross-validation and averaged the results over all the folds. The AUC-PR and CLL values are presented in Table III. MLN-BT has a slightly lower performance compared to MLN-BC since we need longer rules to accurately cluster entities. The entity-resolution task requires rules such as `Title(B1,T1)`, `Title(B2,T2)`, `SameBib(B1,B2)` $\rightarrow$ `SameTitle(T1,T2)` which the greedy approach used in boosting may never find. Since any subset of the given rule would have little impact on the squared error, MLN-BT never learn such rules. MLN-BT scores two literals at a time for a given node and as a result learns short rules that only capture common words between the titles. MLN-BC on the other hand searches for clauses of length 3 and hence may learn longer rules. Nevertheless, both methods are significantly better than other MLN learning methods. While structural Motifs and LHL methods are comparable when predicting the `SameAuthor` relationship, our boosting-based methods are significantly better for all the other relationships.

	AUC-PR				CLL			
Algorithm	SameBib	SameVenue	SameTitle	SameAuthor	SameBib	SameVenue	SameTitle	SameAuthor
MLN-BT	$\mathbf{0.96 \pm 0.02}$	0.56 ± 0.17	0.71 ± 0.20	0.96 ± 0.04	-0.39 ± 0.04	-5.32 ± 1.88	-8.09 ± 2.97	-0.29 ± 0.14
MLN-BC	$\mathbf{0.96 \pm 0.02}$	0.68 ± 0.09	$\mathbf{0.82 \pm 0.13}$	0.98 ± 0.02	-0.33 ± 0.06	-5.12 ± 3.86	-11.18 ± 7.28	-0.60 ± 0.39
Alch-G	0.63 ± 0.17	0.45 ± 0.11	0.54 ± 0.14	0.90 ± 0.05	-5.58 ± 1.49	-4.27 ± 0.96	-5.14 ± 1.39	-8.87 ± 0.37
Alch-D	0.63 ± 0.17	0.48 ± 0.12	0.58 ± 0.16	0.92 ± 0.06	-4.95 ± 0.06	-4.08 ± 1.14	-4.34 ± 0.82	-3.32 ± 1.82
Motif-S	0.63 ± 0.16	0.45 ± 0.10	0.61 ± 0.17	0.93 ± 0.09	-2.54 ± 1.45	-1.80 ± 1.57	-2.79 ± 1.36	-1.57 ± 1.63
LHL	0.63 ± 0.17	0.45 ± 0.10	0.52 ± 0.15	0.91 ± 0.04	-5.99 ± 1.60	-4.20 ± 0.97	-5.11 ± 1.41	-8.80 ± 0.34

Table III
RESULTS ON THE CORA TESTBED.

	AUC-PR			CLL		
Algorithm	workedUnder	genre	gender	workedUnder	genre	gender
MLN-BT	0.90 ± 0.07	0.94 ± 0.08	0.45 ± 0.06	-0.18 ± 0.06	-0.20 ± 0.09	-0.62 ± 0.05
MLN-BC	1.00 ± 0.00	1.00 ± 0.00	0.39 ± 0.07	-0.11 ± 0.04	-0.12 ± 0.08	-0.84 ± 0.21
RDN-B	0.99 ± 0.02	0.91 ± 0.12	0.46 ± 0.18	-0.88 ± 0.20	-0.25 ± 0.22	-0.76 ± 0.16
BUSL	0.89 ± 0.11	0.94 ± 0.08	0.44 ± 0.08	-0.56 ± 0.05	-0.27 ± 0.09	-0.69 ± 0.01
LHL	1.00 ± 0.00	0.37 ± 0.09	0.39 ± 0.12	-0.02 ± 0.01	-1.13 ± 0.23	-0.73 ± 0.05
Motif-S	0.56 ± 0.16	0.52 ± 0.29	0.48 ± 0.08	-2.73 ± 1.66	-3.99 ± 2.70	-0.71 ± 0.08
Motif-L	0.48 ± 0.27	0.39 ± 0.03	0.46 ± 0.08	-2.30 ± 1.16	-2.32 ± 1.15	-0.69 ± 0.06

Table IV
RESULTS ON IMDB DATA SET

IMDB

The IMDB dataset was first used by Mihalkova and Mooney [10] and contains five predicates: `actor`, `director`, `genre`, `gender` and `workedUnder`. We do not evaluate the `actor` and `director` predicates as they are mutually exclusive facts in this dataset and easy to learn for all the methods. Also since gender can take only two values, we convert the `gender(person,gender)` predicate to a single argument predicate `female_gender(person)`. Following [11], we omitted the four equality predicates. We performed five-fold cross-validation using the folds generated by Mihalkova and Mooney [10] and averaged the results across all the folds. We perform inference over every predicate given all other predicates as evidence.

Table IV shows the AUC values for the three predicates: `workedUnder`, `genre` and `gender`. The boosting approaches perform better on average, on both the AUC and CLL values, than the other methods. The BUSL method seems to exhibit the best performance of the prior structure-learning methods in this domain. Our boosting algorithms seem to be comparable or better than BUSL on all the predicates. For `workedUnder`, LHL has comparable AUC values to the boosting approaches, while it is clearly worse on the other predicates. There is no significant difference between the two versions of the boosting algorithms.

The other interesting question that we consider in this domain is: how do boosted MLNs compare against boosted RDNs [14]? To answer this question, we compared our proposed methods against boosted RDNs (RDN-B). As can be seen from Table IV, the MLN-based methods are marginally better than the boosted RDNs for predicting `workedUnder` predicate, while comparable for others. It should be noted that the goal of this work is not to justify the use of MLNs instead of RDNs, but to derive a new and effective learning algorithm for MLNs.

WebKB

The WebKB dataset was first created by Craven et al. [27] and contains information about department webpages and the links between them. It also contains the categories for each webpage and the words within each page. This dataset was converted by Mihalkova and Mooney [10] to contain only the category of each webpage and links between these pages. They created the following predicates: `Student(A)`, `Faculty(A)`, `CourseTA(C, A)`, `CourseProf(C, A)`, `Project(P, A)` and `SamePerson(A, B)` from these webpages. The textual information was ignored. We removed the `SamePerson(A, B)` predicate as it only had groundings with both the arguments being exactly same (i.e., `SamePerson(A,A)`). We evaluated all the methods over the `CourseProf` and `CourseTA` predicates since all other predicates had trivial rules such as `courseTA(C,A)` $\rightarrow$ `Student(A)`. We performed 4-fold cross-validation where each fold corresponds to one university. We do not present the performance of BUSL with default setting and Motif-S (short rules) in this domain because the algorithms were unable to learn any useful rules in our formulation and hence had a AUC-PR value of 0.

Table V presents the results of the different algorithms in this domain. As with UW data set we present two different cases here. First is the data set with all the negative examples in the test set and the second is the data set with twice the number of negatives as positives. Similar to the earlier case, in the test set with all negatives, current MLN methods such as LHL and Motifs exhibit good performance for the CLL evaluation measure for both the `courseTA` and `courseProf` predicates. On the other hand, the AUC-PR values are significantly lower than that of our boosting-based methods. This difference is magnified when we limit the number of negatives to twice the number of positives. In the latter case, even the CLL for the current MLN structure learning algorithms are significantly worse than our boosting methods. There is no statistically significant difference between the performance of the boosting methods. Our current results show that employing a test set with a reasonable distribution of the classes yields a better insight into the difference in the performance of the learning algorithms.

Precision-Recall curves

We also present the PR curves for the first fold on `workedUnder` predicate in IMDB in Figure 5, `genre` predicate in IMDB in Figure 6, `SameBib` predicate in Cora in Figure 7, the `SameVenue` predicate in Cora in Figure 8 and the `courseTA` predicate in Web-KB in Figure 9. We only show the curves for the best previously published structure-learning methods. Our algorithms exhibit a clear superior performance especially in the high-recall regions.

CONCLUSION

Since MLNs provide clear semantics and convergent inference approaches [28], they are among the most popular SRL methods. But learning the structure of MLNs remains one of the hardest and challenging problems. We address the problem of structure learning by using gradient-boosting with the added benefit of learning weights simultaneously. A similar approach has been taken in the propositional world for learning Markov Networks by Lowd and Davis [29]. In their work, they learn Markov network structure as a series of local models where each local model is a set of decision trees. Our proposed approach can be seen as generalizing their approach to learning MLNs by using functional gradient boosting and extending the work of Natarajan et al. [14], who learned RDNs as a series of first-order regression trees. Building upon the success of pseudo-likelihood methods for MLNs, we derived tree-based and clause-based gradient boosting algorithms. We evaluated the algorithms on four standard datasets and established the superior performance of the boosting method across all the domains and all the predicates. Our methods' restriction that the structure be only Horn clauses did not affect the results.

One future direction is to derive the functional gradients for the full likelihood instead of the pseudo-likelihood and learn the trees/clauses for jointly predicting several predicates. Another direction is to induce a simpler MLN that approximates the learned set of clauses/trees; this will ensure that the learned model is interpretable as well. An additional avenue for future work is learning with partially observed data or even in open-world domains. Finally, it is an interesting future direction to demonstrate the usefulness of boosting in additional real-world tasks.

ACKNOWLEDGEMENTS

Tushar Khot, Sriraam Natarajan and Jude Shavlik gratefully acknowledge support of the DARPA Machine Reading Program under the Air Force Research Laboratory (AFRL) prime contract no. FA8750-09-C-0181. Any opinions, findings, and conclusion or recommendations expressed in this material are those of the authors and do not necessarily reflect the view of the DARPA, AFRL, or the US government. Kristian Kersting was supported by the Fraunhofer ATTRACT fellowship STREAM and by the European Commission under contract number FP7-248258-First-MM.

REFERENCES

[1] L. Getoor and B. Taskar, *Introduction to Statistical Relational Learning*. MIT Press, 2007.

[2] L. Getoor, N. Friedman, D. Koller, and A. Pfeffer, "Learning probabilistic relational models," *Relational Data Mining, S. Dzeroski and N. Lavrac, Eds.*, pp. 307–338, 2001.

[3] K. Kersting and L. De Raedt, "Bayesian logic programming: Theory and tool," in *An Introduction to Statistical Relational Learning*, 2007.

[4] M. Jaeger, "Relational Bayesian networks," in *UAI*, 1997.

[5] P. Domingos and D. Lowd, *Markov Logic: An Interface Layer for AI*. San Rafael, CA: Morgan & Claypool, 2009.

[6] B. Taskar, P. Abeel, and D. Koller, "Discriminative probabilistic models for relational data," in *UAI*, 2002.

[7] T. Sato and Y. Kameya, "Parameter learning of logic programs for symbolic-statistical modeling," *JAIR*, pp. 391–454, 2001.

[8] D. Poole, "Probabilistic Horn abduction and Bayesian networks," *AIJ*, pp. 81–129, 1993.

[9] M. Biba, S. Ferilli, and F. Esposito, "Structure learning of Markov logic networks through iterated local search," in *ECAI*, 2008.

[10] L. Mihalkova and R. Mooney, "Bottom-up learning of Markov logic network structure," in *ICML*, 2007, pp. 625–632.

[11] S. Kok and P. Domingos, "Learning Markov logic network structure via hypergraph lifting," in *ICML*, 2009.

[12] ——, "Learning Markov logic networks using structural motifs," in *ICML*, 2010.

[13] J. Friedman, "Greedy function approximation: A gradient boosting machine," *Annals of Statistics*, vol. 29, 2001.

[14] S. Natarajan, T. Khot, K. Kersting, B. Gutmann, and J. Shavlik, "Gradient-based boosting for statistical relational learning: The Relational Dependency Network case," *MLJ*, To appear.

[15] B. Gutmann and K. Kersting, "TildeCRF: Conditional random fields for logical sequences," in *ECML*, 2006.

Algo	2x negatives				All negatives			
	AUC-PR		CLL		AUC-PR		CLL	
	courseTA	courseProf	courseTA	courseProf	courseTA	courseProf	courseTA	courseProf
MLN-BT	0.426 ± 0.027	0.738 ± 0.034	$\mathbf{-0.603 \pm 0.057}$	$\mathbf{-0.406 \pm 0.050}$	0.005 ± 0.003	0.029 ± 0.005	$\mathbf{-0.359 \pm 0.041}$	$\mathbf{-0.334 \pm 0.068}$
MLN-BC	0.379 ± 0.031	0.750 ± 0.110	$\mathbf{-0.656 \pm 0.012}$	$\mathbf{-0.357 \pm 0.045}$	0.004 ± 0.002	0.027 ± 0.007	$\mathbf{-0.479 \pm 0.041}$	$\mathbf{-0.304 \pm 0.010}$
LHL	0.350 ± 0.046	0.460 ± 0.036	-2.274 ± 0.102	-2.243 ± 0.104	0.004 ± 0.002	0.007 ± 0.002	-0.023 ± 0.011	-0.029 ± 0.005
Motif-L	0.332 ± 0.014	0.637 ± 0.219	-2.282 ± 0.110	-2.198 ± 0.105	0.003 ± 0.002	0.017 ± 0.009	-0.024 ± 0.011	-0.029 ± 0.005

Table V
RESULTS ON WEBKB DATA SET

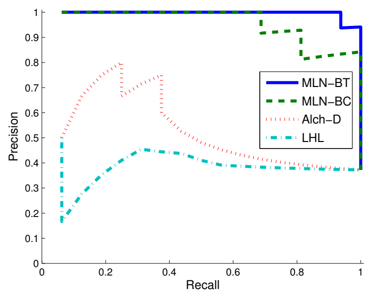

Fig. 4. PR Curve for UW

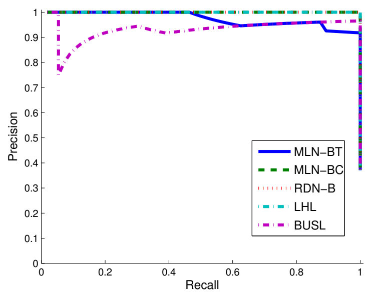

Fig. 5. PR Curve for workedUnder in IMDB

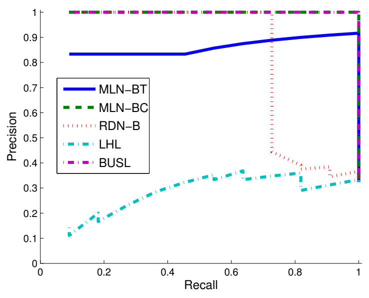

Fig. 6. PR Curve for genre in IMDB

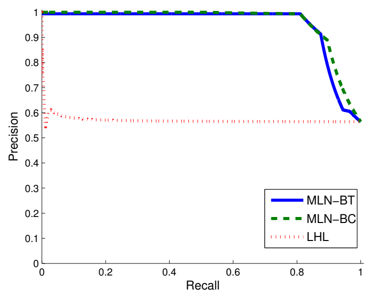

Fig. 7. PR Curve for SameBib in Cora

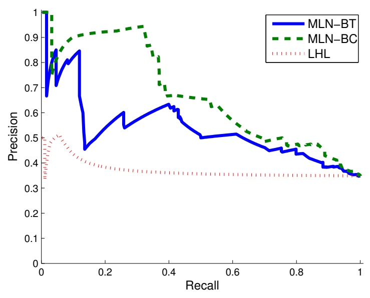

Fig. 8. PR Curve for SameVenue in Cora

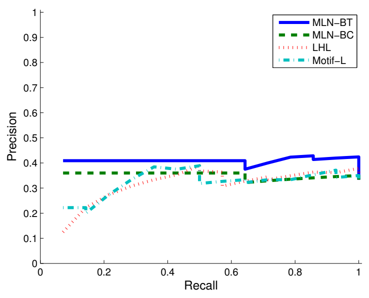

Fig. 9. PR Curve for courseTA in WebKB

[16] K. Kersting and K. Driessens, "Non–parametric policy gradients: A unified treatment of propositional and relational domains," in *ICML*, 2008.

[17] A. Karwath, K. Kersting, and N. Landwehr, "Boosting relational sequence alignments," in *ICDM*, 2008.

[18] T. Dietterich, A. Ashenfelter, and Y. Bulatov, "Gradient tree boosting for training conditional random fields," *JMLR*, pp. 2113–2139, 2008.

[19] J. Shavlik and S. Natarajan, "Speeding up inference in Markov logic networks by preprocessing to reduce the size of the resulting grounded network," in *IJCAI*, 2009.

[20] H. Blockeel and L. D. Raedt, "Top-down induction of first-order logical decision trees," *Artificial Intelligence*, vol. 101, pp. 285–297, 1998.

[21] S. Kok, M. Sumner, M. Richardson, P. Singla, H. Poon, D. Lowd, J. Wang, A. Nath, and P. Domingos, "The Alchemy system for statistical relational AI," Department of Computer Science and Engineering, University of Washington, Seattle, WA, Tech. Rep., 2010, http://alchemy.cs.washington.edu.

[22] J. Davis and M. Goadrich, "The relationship between Precision-Recall and ROC curves," in *ICML*, 2006.

[23] M. Richardson and P. Domingos, "Markov logic networks," *Machine Learning*, vol. 62, pp. 107–136, 2006.

[24] M. Bilenko and R. Mooney, "Adaptive duplicate detection using learnable string similarity measures," in *KDD*, 2003.

[25] H. Poon and P. Domingos, "Joint inference in information extraction," in *AAAI*, 2007, pp. 913–918.

[26] P. Singla and P. Domingos, "Entity resolution with Markov logic," in *ICDM*, 2006, pp. 572–582.

[27] M. Craven, D. DiPasquo, D. Freitag, A. McCallum, T. Mitchell, K. Nigam, and S. Slattery, "Learning to extract symbolic knowledge from the World Wide Web," in *AAAI*, 1998, pp. 509–516.

[28] P. Singla and P. Domingos, "Lifted first-order belief propagation," in *AAAI*, 2008, pp. 1094–1099.

[29] D. Lowd and J. Davis, "Learning Markov network structure with decision trees," in *ICDM*, 2010.