To appear in *Proceedings of the 24th Annual Intl. Conf. on Machine Learning (ICML-07)*. Corvallis, OR, June 2007.

Bottom-Up Learning of Markov Logic Network Structure

Lilyana Mihalkova LILYANAM@CS.UTEXAS.EDU
Raymond J. Mooney MOONEY@CS.UTEXAS.EDU
Department of Computer Sciences, The University of Texas at Austin, 1 University Station C0500, Austin TX 78712, USA

Abstract

Markov logic networks (MLNs) are a statistical relational model that consists of weighted first-order clauses and generalizes first-order logic and Markov networks. The current state-of-the-art algorithm for learning MLN structure follows a top-down paradigm where many potential candidate structures are systematically generated without considering the data and then evaluated using a statistical measure of their fit to the data. Even though this existing algorithm outperforms an impressive array of benchmarks, its greedy search is susceptible to local maxima or plateaus. We present a novel algorithm for learning MLN structure that follows a more bottom-up approach to address this problem. Our algorithm uses a "propositional" Markov network learning method to construct "template" networks that guide the construction of candidate clauses. Our algorithm significantly improves accuracy and learning time over the existing top-down approach in three real-world domains.

1. Introduction

Methods for unifying the strengths of first-order logic and probabilistic graphical models have become an important aspect of recent research in machine learning (Getoor & Taskar, to appear 2007). Markov logic networks (MLNs) are a recently developed statistical relational model that generalizes both full first-order logic and Markov networks (Richardson & Domingos, 2006). An MLN consists of a set of weighted clauses in first-order logic, and learning an MLN decomposes into *structure learning*, or learning the logical clauses, and *weight learning*, or setting the weight of each clause. Kok and Domingos (2005) have proposed a probabilistic method for learning MLN structure and shown that it produces more accurate sets of clauses than several previous inductive logic programming (ILP) methods. Like many existing methods for learning logic programs (Quinlan, 1990) and graphical models (Heckerman, 1995), they take a *top-down* approach, heuristically searching the space of increasingly complex models, guiding the search by scoring models using a statistical measure of their fit to the training data.

Such top-down approaches follow a "blind" generate-and-test strategy in which many potential changes to an existing model are systematically generated independent of the training data, and then tested for empirical adequacy. For complex models such as MLNs, the space of potential revisions is combinatorially explosive and such a search can become difficult to control, resulting in convergence to suboptimal local maxima. *Bottom-up* learning methods attempt to use the training data to directly construct promising structural changes or additions to the model (Muggleton & Feng, 1992). Many effective ILP algorithms use some combination of top-down and bottom-up search (Zelle et al., 1994; Muggleton, 1995).

We argue that, if properly designed, a more bottom-up statistical relational learner can outperform a purely top-down approach because of the large search space with many local maxima and plateaus that can be avoided by utilizing stronger guidance from the data. We present a more bottom-up approach to learning MLN structure that first uses a "propositional" Markov network structure learner to construct "template" networks that then guide the construction of candidate clauses. Our approach therefore follows the ILP tradition of "upgrading" a propositional learner to handle relational data (Van Laer & De Raedt, 2001). However, our approach can employ any existing Markov network structure learning algorithm as a subroutine. Experiments on three real-world relational data sets demonstrate that our approach significantly out-performs the current state-of-the-art MLN structure learner on two commonly-used metrics while greatly decreasing the training time and the number of candidate models considered during training.

2. Terminology and Notation

First-order logic distinguishes among four types of symbols—constants, variables, predicates, and functions (Russell & Norvig, 2003). Constants describe the objects in

Appearing in *Proceedings of the 24^th International Conference on Machine Learning*, Corvallis, OR, 2007. Copyright 2007 by the author(s)/owner(s).

a domain and can have types. Variables act as placeholders to allow for quantification. Predicates represent relations in the domain, such as $WorkedFor$. Function symbols represent functions of tuples of objects. The arity of a predicate or a function is the number of arguments it takes. We assume that the domains contain no functions, an assumption also made in previous research (Kok & Domingos, 2005). We denote constants by strings starting with lower-case letters, variables by single upper-case letters, and predicates by strings starting with upper-case letters.

Example: *As a running example, we will use a simplified version of one of our domains. The domain contains facts about individuals in the movie business, describing their profession ($Actor(A)$ or $Director(A)$), their relationships, and the movies on which they have worked. The $WorkedFor(A, B)$ predicate specifies that person A worked on a movie under the supervision of person B, and the $Movie(T, A)$ predicate specifies that individual A appeared in the credits of movie T. A, B, and T are variables. The domain has the constants $brando$ and $coppola$ of type person, and $godFather$ of type movieTitle.*

A term is a constant, a variable, or a function that is applied to terms. Ground terms contain no variables. An atom is a predicate applied to terms. A positive literal is an atom, and a negative literal is a negated atom. We will use the word *gliteral* to refer to a ground literal, i.e. one containing only constants, and *vliteral* to refer to a literal that contains only variables. A clause is a disjunction of positive and negative literals. The length of a clause is the number of literals in the disjunction. A definite clause is a clause with only one positive literal, called the head, whereas the negative literals compose the body. A world is an assignment of truth values to all possible gliterals in a domain.

3. Markov Logic Networks

An MLN consists of a set of first-order clauses, each of which has an associated weight (Richardson & Domingos, 2006). MLNs can be viewed as a way to soften first-order logic, making worlds that violate some of the clauses less likely but not altogether impossible. Let $\mathbf{X}$ be the set of all propositions describing a world (i.e. all gliterals formed by grounding the predicates with the constants in the domain), $\mathcal{F}$ be the set of all clauses in the MLN, w_i be the weight associated with clause $f_i \in \mathcal{F}$, $\mathcal{G}_{f_i}$ be the set of all possible groundings of clause f_i with the constants in the domain Then the probability of a particular truth assignment $\mathbf{x}$ to $\mathbf{X}$ is given by the formula (Richardson & Domingos, 2006):

$$P(\mathbf{X} = \mathbf{x}) = (1/Z) \exp \left(\sum_{f_i \in \mathcal{F}} w_i \sum_{g \in \mathcal{G}_{f_i}} g(\mathbf{x}) \right)$$

Here Z is the normalizing partition function. The value of $g(\mathbf{x})$ is either 1 or 0, depending on whether g is satisfied. Thus the quantity $\sum_{g \in \mathcal{G}_{f_i}} g(\mathbf{x})$ counts the number of groundings of f_i that are true given the current truth assignment to $\mathbf{X}$. The first-order clauses are commonly referred to as *structure*. Figure 1 shows a sample MLN. To perform inference over a given MLN, one needs to ground it into its corresponding Markov network (Pearl, 1988). As described by Richardson and Domingos (2006), this is done as follows. First, all possible gliterals in the domain are formed, and they serve as the nodes in the Markov network. The edges are determined by the groundings of the first-order clauses: gliterals that participate together in a grounding of a clause are connected by an edge. Thus, nodes that appear together in a ground clause form cliques. For example, Figure 2 shows the ground Markov network corresponding to the MLN in Figure 1. Several techniques, such as Gibbs sampling (Richardson & Domingos, 2006) or MC-SAT (Poon & Domingos, 2006), can be used to perform inference over the ground Markov network.

0.7	$Actor(A) \Rightarrow \neg Director(A)$
1.2	$Director(A) \Rightarrow \neg WorkedFor(A, B)$
1.4	$Movie(T, A) \wedge WorkedFor(A, B) \Rightarrow Movie(T, B)$

Figure 1. Simple MLN for example domain

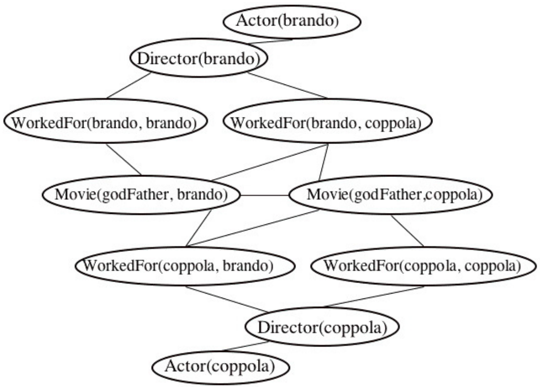


Figure 2. Result of grounding the example MLN

The current state-of-the-art MLN structure learning algorithm, due to Kok and Domingos (2005), proceeds in a top-down fashion, employing either beam search or shortest-first search. We will compare to the beam-search version, which we call TDSL (for Top-Down Structure Learning).[1] TDSL performs several iterations of beam search, and after each iteration adds to the MLN the best clause found. Clauses are evaluated using a weighted pseudo log-likelihood measure (WPLL), introduced in (Kok & Domingos, 2005), that sums over the log-likelihood of each node given its Markov blanket, weighting it appropriately to ensure that predicates with many gliterals do not dominate

[1] The shortest-first search constructs candidates in the same way but conducts a more complete search, which, however, requires longer training times.

the result. The beam search in each iteration starts from all single-vliteral clauses. It generates candidates by adding a vliteral in each possible way to the initial clauses, keeps the best $beamSize$ clauses, from which it generates new candidates by performing all possible vliteral additions, keeps the best $beamSize$ and continues in this way until candidates stop improving the WPLL. At this point, the best candidate found is added to the MLN, and a new beam search iteration begins. Weights need to be learned for a given structure before its WPLL can be computed. Weight-training can be efficiently implemented as an optimization procedure (Richardson & Domingos, 2006). While TDSL has been empirically shown to outperform an impressive number of competitive baselines (Kok & Domingos, 2005), it has two potential drawbacks. First, the iterative process of generating and testing candidates may result in long training times; and second, the greedy beam search is susceptible to overlooking potentially useful clauses. We present an algorithm that attempts to avoid these drawbacks by learning MLN structure in a bottom-up fashion. We call our algorithm BUSL (for Bottom-Up Structure Learning).

4. Bottom-Up Structure Learning

As pointed out by Richardson and Domingos (2006), MLNs serve as templates for constructing Markov networks when different sets of constants are provided. Because the edges of the ground Markov network are defined by the groundings of the same set of first-order clauses, the same pattern is repeated several times in the graph, corresponding to each grounding of a particular clause.

Example: *We observe that in Figure 2 the pattern of nodes and edges appearing above the two $Movie$ gliterals is repeated below them with different constants. In fact, this Markov network can be viewed as an instantiation of the template shown in Figure 3.*

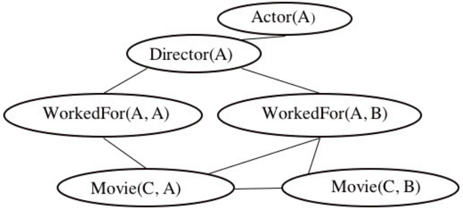


Figure 3. Example Markov Network Template

The basic idea behind BUSL is to learn MLN structure by first *automatically* creating a Markov network template similar to the one in Figure 3 from the provided data. The nodes in this template are used as components from which clauses are constructed and can contain one or more vliterals that are connected by a shared variable. We will call these nodes *TNodes* for template nodes. As in ordinary Markov networks, a TNode is independent of all other TNodes given its immediate neighbors (i.e. its Markov blanket). Every probability distribution respecting the independencies captured by the graph of a Markov network can be represented as the product of functions defined only over the cliques of the graph (Pearl, 1988). Analogously, to specify the probability distribution over a Markov network template, the algorithm only needs to consider clauses defined over the cliques of the template. Thus, BUSL uses the Markov network template to restrict the search space for clauses only to those candidates whose literals correspond to TNodes that form a clique in the template. Algorithm 1 gives the skeleton of BUSL. Letting $\mathcal{P}$ be the set of all predicates in the domain, the algorithm considers each predicate $P \in \mathcal{P}$ in turn. A Markov network template is constructed for each P. Template construction involves creating variablized TNodes and determining the edges between them. The template does not specify the actual MLN clauses or their weights. To search for actual clauses, we generate clause candidates by focusing on each maximal clique in turn and producing all possible clauses consistent with it. We can then evaluate each candidate using the WPLL score. We next describe how a Markov network template is created for the current predicate P.

Algorithm 1 Skeleton of BUSL

```
for each $P \in \mathcal{P}$ do
    Construct TNodes for predicate $P$ (Section 4.1)
    Connect the TNodes to form a Markov network template (Section 4.2)
    Create candidate clauses, using this template to constrain the search (Section 4.3)
end for
Remove duplicate candidates
Evaluate candidates and add best ones to final MLN
```

4.1. TNode construction

TNodes contain conjunctions of one or more vliterals and serve as building blocks for creating clauses. Intuitively, TNodes are constructed by looking for groups of constant-sharing gliterals that are true in the data and variablizing them. Thus, TNodes could also be viewed as portions of clauses that have true groundings in the data. TNode construction is inspired by relational pathfinding (Richards & Mooney, 1992). The result of running TNode construction for P is the set of TNodes and a matrix M_P containing a column for each of the created TNodes and a row for each gliteral of P. Each entry $M_P[r][c]$ is a Boolean value that indicates whether the data contains a true grounding of the TNode corresponding to column c with at least one of the constants of the gliteral corresponding to row r. This matrix is used later to find the edges between the TNodes. Algorithm 2 describes how the set of TNodes and the matrix M_P are constructed. It uses the following definitions:

Definition 1 *Two gliterals (vliterals) are* connected *if there exists a constant (variable) that is an argument of both.*

Algorithm 2 Construct TNode Set

```
Input:
 1: P: Predicate currently under consideration
 2: m: Maximum number of vliterals in a TNode
Output:
 3: TNodeVector: Vector of constructed TNodes
 4: $M_P$: Matrix of Boolean values
Procedure:
 5: Make head TNode, headTN, and place it in position 0 of
    TNodeVector
 6: for each (true or false) gliteral, $G_P$, of P do
 7:   Add a row of 0-s to $M_P$
 8:   if $G_P$ is true then
 9:     Set $M_P[\text{lastRow}(M_P)][0] = 1$
10:   end if
11:   Let $\mathcal{C}_{G_P}$ be the set of true gliterals connected to $G_P$
12:   for each $c \in \mathcal{C}_{G_P}$ do
13:     for each possible TNode based on c do
14:       newTNode = CreateTNode(c, $G_P$, headTN, m)
15:       position = TNodeVector.find(newTNode)
16:       if position is not valid then
17:         append newTNode to end of TNodeVector
18:         append a column of 0-s to $M_P$
19:         position = numColumns($M_P$) − 1
20:       end if
21:       Set $M_P[\text{lastRow}(M_P)][\text{position}] = 1$
22:     end for
23:   end for
24: end for
```

Algorithm 3 CreateTNode

```
Input:
 1: $G_P$: Current gliteral of P under consideration
 2: c: Gliteral connected to $G_P$ on which TNode is based
 3: headTN: Head TNode
 4: m: Max number of of vliterals allowed in a TNode
Output:
 5: newTNode: The constructed TNode
Procedure:
 6: size = pick a size for this TNode (between 1 and m)
 7: v = variablize(c)
 8: Create newTNode containing v
 9: prevGliteral, lastVliteralInChain = c, v
10: while length(newTNode) < size do
11:   $c_1$ = pick true gliteral connected to prevGliteral
12:   $v_1$ = variablize($c_1$) and add $v_1$ to newTNode
13:   prevGliteral, lastVliteralInChain = $c_1$, $v_1$
14: end while
```

Definition 2 *A* chain *of literals of length l is a list of l literals such that for $1 < k \leqslant l$ the kth literal is connected to the $(k-1)$th via a previously unshared variable.*

First, in line 5, the algorithm creates a *head* TNode that consists of a vliteral of P in which each argument is assigned a unique variable. This TNode is analogous to the head in a definite clause; however, note that our algorithm is not limited to constructing only definite clauses. Next, in lines 6 to 24, the algorithm considers each (true or false) gliteral G_P of P in turn (the true gliterals are those stated to hold in the data, and the rest are false). A row of zeros is added to M_P for G_P, and the value corresponding to the head TNode is set to 1 if G_P is true and to 0 otherwise (lines 8-10). The algorithm then considers the set $\mathcal{C}_{G_P}$ of all true gliterals in the data that are connected to G_P. For each $c \in \mathcal{C}_{G_P}$, it constructs each possible TNode based on c containing at most m vliterals (Algorithm 3). If a particular TNode was previously created, its value in the row corresponding to G_P is set to 1 (line 21). Otherwise, a new column of zeros is added to M_P and the entry in the G_P row is set to 1 (lines 16-20).

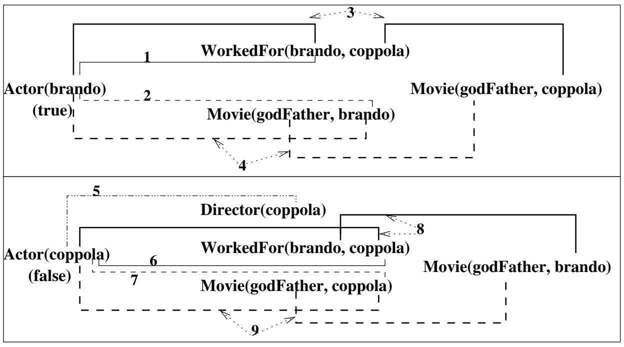


Figure 4. Illustration of TNode construction (See example below). The thin lines show the connections defining single-vliteral TNodes, and the thick lines the connections defining two-vliteral TNodes. The lines link the constants shared between the gliterals.

Algorithm 3 lists the `CreateTNode` procedure. The algorithm determines the number of vliterals in the new TNode in line 6. It then variablizes the current gliteral c connected to G_P by replacing the constants c shares with G_P with their corresponding variables from the head TNode. If the chosen TNode size is greater than 1, the algorithm enters the while loop in lines 10-14. In each iteration of this loop we extend the TNode with an additional vliteral that is constructed by variablizing a gliteral connected to the gliteral considered in the previous iteration so that any constants shared with the head TNode or with the previous gliteral are replaced with their corresponding variables.

Example: *Suppose we are given the following database for our example domain where the listed gliterals are $true$ and the omitted ones are $false$:*

$$Actor(brando)\ Director(coppola)$$
$$WorkedFor(brando, coppola)$$
$$Movie(godFather, coppola)\ Movie(godFather, brando)$$

Let P = Actor and $m = 2$ (i.e. at most 2 vliterals per TNode). The head TNode is Actor(A). Figure 4 shows the gliteral chains considered in the main loop (lines 6-24) of Algorithm 2 for each gliteral of P. Let us first focus on the case when G_P is Actor(brando) (top part). Connections 1 and 2 lead to the TNodes WorkedFor(A, B) and Movie(C, A) respectively. Connection 3 gives rise to the 2-vliteral TNode [WorkedFor(A, D), Movie(E, D)], and connection

4, to the TNode [Movie(F, A), Movie(F, G)]. The following table lists the values in M_P at this point.

Actor(A)	*Worked For(A, B)*	*Movie(C, A)*	*WorkedFor(A, D) Movie(E, D)*	*Movie(F, A) Movie(F, G)*
1	*1*	*1*	*1*	*1*

Note that when constructing the TNodes, we replaced shared constants with the same variables, and constants shared with G_P with the corresponding variable from the head TNode.

We next consider the bottom part of Figure 4 that deals with the second iteration in which G_P is Actor(coppola). From connection 5, we construct the TNode Director(A) and from connection 6 the TNode WorkedFor(H, A), which differs from the WorkedFor TNode found earlier by the position of the variable A shared with the head TNode. An appropriate TNode for connection 7 (Movie(C,A)) already exists. Connection 8 gives rise to the two-vliteral TNode [WorkedFor(I, A), Movie(J, I)]. A TNode for connection 9, [Movie(F, A), Movie(F, G)] was constructed in the previous iteration. Table 1 lists the final set of TNodes.

Larger values of m mean longer TNodes that could help build more informative clauses. However, a larger m also leads to the construction of more TNodes, thus increasing the search space for clauses. We used a conservative setting of $m = 2$. Note that this does not limit the final clause length to 2. To further reduce the search space, we require that TNodes with more than one vliteral contain at most one free variable (i.e. a variable that does not appear in more than one of the vliterals in the TNode or in the head TNode). We did not experiment with more liberal settings of these parameters but, as our experiments demonstrate, these values worked well in our domains.

TNode construction is very much in the spirit of bottom-up learning. Rather than producing all possible vliterals that share variables with one another in all possible ways, the algorithm focuses only on vliterals for which there is a true gliteral in the data. Thus, the data already guides and constrains the algorithm. This is related to bottom-up ILP techniques such as least-general generalizations (LGG) and inverse resolution (Lavrač & Džeroski, 1994). However, as opposed to LGG, our TNode construction algorithm always uses the generalization that leads to completely variablized TNodes and unlike inverse resolution, the process does not lead to the creation of complete clauses and does not use any logical inference algorithms like resolution.

4.2. Adding the Edges

To complete the template construction, we need to find which TNodes are connected by edges. Recall that the templates represent variablized analogs of Markov networks. Thus, finding the edges can be cast as a Markov network structure learning problem where the TNodes are the nodes in the Markov network and the matrix M_P provides training data. Any Markov network learning algorithm can be employed. We chose the recent Grow-Shrink Markov Network (GSMN) algorithm by Bromberg et al. (2006). GSMN uses χ^2 statistical tests to determine whether two nodes are conditionally independent of each other.

4.3. Search for clauses

As discussed earlier we only construct clauses from TNodes that form cliques in the Markov network template; i.e., any two TNodes participating together in a clause must be connected by an edge in the template. The head TNode must participate in every candidate. A clause can contain at most one multiple-literal TNode and at most one TNode containing a single non-unary literal. These restrictions are designed to decrease the number of free variables in a clause, thus decreasing the size of the ground MLN during inference, and further reducing the search space. Complying with the above restrictions, we consider each clique in which the head TNode participates and construct all possible clauses of length 1 to the size of the clique by forming disjunctions from the literals of the participating TNodes with all possible negation/non-negation combinations.

After template creation and clause candidate generation are carried out for each predicate in the domain, duplicates are removed and the candidates are evaluated using the WPLL. To compute this score, one needs to assign a weight to each clause. For weight-learning, we use L-BFGS like Richardson and Domingos (2006). After all candidates are scored, they are considered for addition to the MLN in order of decreasing score. To reduce overfitting and speed up inference, only candidates with weight greater than *minWeight* are considered. Candidates that do not increase the overall WPLL of the learned structure are discarded.

5. Experimental Setup

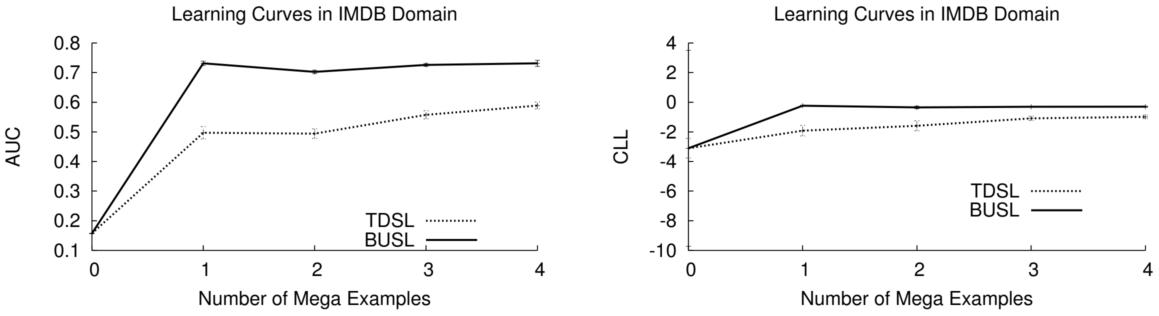
We compared BUSL and TDSL in three relational domains—IMDB, UW-CSE, and WebKB. Each dataset is broken down into *mega-examples*, where each mega-example contains a connected group of facts. Individual mega-examples are independent of each other. Learning curves were generated using a leave-1-mega-example-out approach, averaging over k different runs, where k is the number of mega-examples in the domain. In each run, we reserved a different mega-example for testing and trained on the remaining $k-1$, provided one by one. Both learners observed the same sequence of mega-examples.

The IMDB database contains five mega-examples, each of which describes four movies, their directors, and the first-billed actors who appear in them. Each director is ascribed genres based on the genres of the movies he or she directed. The *Gender* predicate is only used to state the genders of actors. The UW-CSE database was first

Actor(A)	WorkedFor(A, B)	Movie(C, A)	WorkedFor(A, D)
			Movie(E, D)
				Movie(F, A)
				Movie(F, G)
					Director(A)	WorkedFor(H, A)	WorkedFor(I, A)
							Movie(J, I)

1	1	1	1	1	0	0	0
0	0	1	0	1	1	1	1

Table 1. Final set of TNodes and their corresponding M_P matrix. The head TNode is in bold.



a)

b)

Figure 5. Accuracy in IMDB domain. a) AUC b) CLL

used by Richardson and Domingos (2006).[2] It lists facts about people in an academic department (i.e. $Student$, $Professor$) and their relationships (i.e. $AdvisedBy$, $Publication$) and is divided into mega-examples based on five areas of computer science.[3] The WebKB database contains information about entities from the "University Computer Science Department" data set, compiled by Craven et al. (1998). The original dataset contains web pages from four universities labeled according to the entity they describe (e.g. student, course), as well as the words that occur in these pages. Our version of WebKB contains the predicates $Student(A)$, $Faculty(A)$, $CourseTA(C, A)$, $CourseProf(C, A)$, $Project(P, A)$ and $SamePerson(A, B)$. The textual information is ignored. This data contains four mega-examples, each of which describes one university. The following table provides additional statistics about the domains:

Data Set	Num Consts	Num Types	Num Preds	Num True Gliterals	Total Num Gliterals
IMDB	316	4	10	1,540	32,615
UW-CSE	1,323	9	15	2,673	678,899
WebKB	1,700	3	6	2,065	688,193

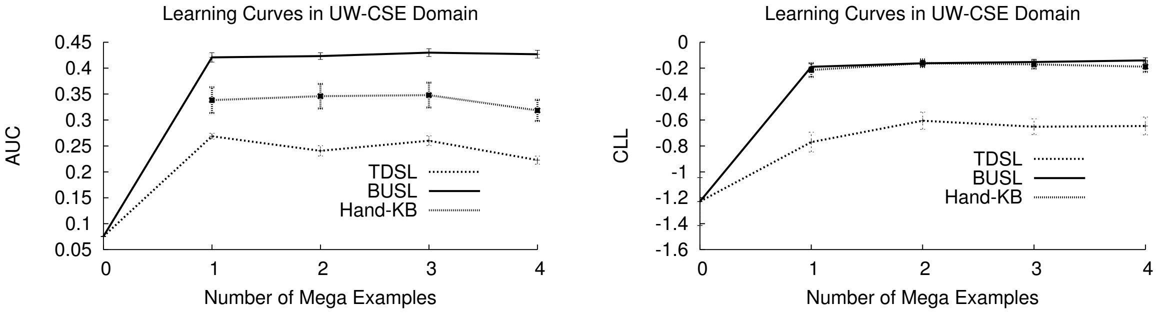
We measured the performance of BUSL and TDSL using the two metrics employed by Kok and Domingos (2005), the area under the precision-recall curve (AUC) and the conditional log-likelihood (CLL). The AUC is useful because it demonstrates how well the algorithm predicts the few positives in the data. The CLL determines the quality of the probability predictions output by the algorithm. To calculate the AUC and the CLL of a given MLN, one needs to perform inference over it, providing some of the gliterals in the test mega-example as evidence and testing the predictions for the remaining ones. We used the MC-SAT inference algorithm (Poon & Domingos, 2006) and tested for the gliterals of each of the predicates of the domain in turn, providing the rest as evidence. Thus each point on the curves is the average CLL or AUC calculated after doing inference for each of the predicates in the domain over the MLN learned in each of the k runs. The error bars are formed by averaging the standard error over the predictions for the groundings of each predicate and over the learning runs. We performed all timing runs within the same domain on the same dedicated machine. We used the implementation of TDSL provided in the Alchemy package (Kok et al., 2005) and implemented BUSL as part of the same package. We set TDSL's parameters as in (Kok & Domingos, 2005) for the UW-CSE experiment. The same settings were kept in the two new domains, except that we found that $minWeight = 1$ was too restrictive and instead used $minWeight = 0.1$. We set BUSL's $minWeight = 0.5$ for all experiments. Those parameter settings were independently informally optimized for both systems. Even though the two algorithms both have a parameter called $minWeight$, they use it in different ways and the same value is therefore not necessarily optimal for both systems.

6. Experimental Results

Figures 5-7 show learning curves in the three domains. Error bars are drawn on all curves but in some cases they are

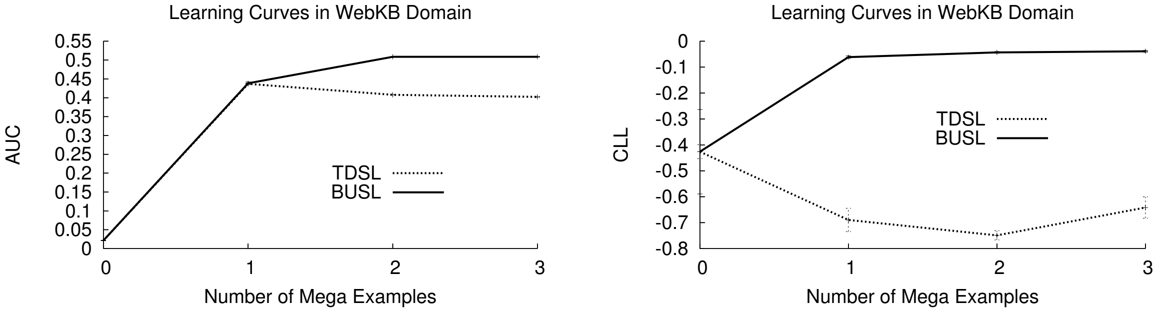
[2] http://www.cs.washington.edu/ai/mln/database.html.

[3] Our results on this dataset are not comparable to those presented by Kok and Domingos (2005) because due to privacy issues we only had access to the published version of this data, which differs from the original (Personal communic. by S. Kok).



a) b)

Figure 6. Accuracy in UW-CSE domain. a) AUC b) CLL



a) b)

Figure 7. Accuracy in WebKB domain. a) AUC b) CLL

tiny. BUSL improves over the performance of TDSL in all cases except for one in terms of AUC and in all cases in terms of CLL. Figure 6 also plots the AUC and CLL for a system that performs weight learning over the manually developed knowledge base provided as part of the UW-CSE dataset (Hand-KB). BUSL outperforms Hand-KB in terms of AUC, and does equally well in terms of CLL. In Figure 7, even though TDSL is improving its performance in terms of AUC, its CLL score decreases. This is most probably due to the extremely small relative number of true gliterals in WebKB in which the CLL can be increased by predicting $false$ for each query. Note that the learners improve very little beyond the first mega-example. This occurs because in our experience, additional data improves the WPLL estimate but has little effect on the clauses that are proposed. In particular, in BUSL candidates are based on the dependencies among the TNodes, and in our domains new data introduces few such dependencies.

Table 2 shows the average training time over all learning runs for each system, and the average number of candidate clauses each learner constructed and evaluated over all runs. BUSL constructed fewer candidates and trained faster than TDSL. BUSL spends most of its training time on calculating the WPLL for the generated candidates. This takes longer in domains like WebKB that contain many constants. On the other hand, we expect BUSL's savings in terms of number of generated candidates to be greater in domains, such as UW-CSE, that contain many predicates because the large number of predicates would increase the number of candidate clauses generated by TDSL. These considerations explain why the smallest improvement in speed is achieved in WebKB that contains the least number of predicates and the greatest number of constants. Based on the much smaller number of candidate clauses considered by BUSL, one might expect a larger speed-up. This is not observed because of optimizations within Alchemy that allow fast scoring of clauses for a fixed structure of the MLN. Because TDSL evaluates a large number of candidates with a fixed structure, it can take advantage of these optimizations. On the other hand, after initially scoring all candidates, BUSL attempts to add them one by one to the MLN, thus changing the MLN at almost each step, which slows down the WPLL computation.

Finally, we checked the importance of adding the edges in Section 4.2. This step can in principle be avoided by producing a fully connected Markov network template. We found that this step helped decrease the search space because in the experiments we presented, out of 31.44, 70.70, and 18.83 TNodes that were constructed on average over the predicates in each domain (IMDB, UW-CSE, and WebKB), only 12%, 14% and 22% respectively ended up in the Markov blanket of the head TNode on average.

7. Related Work

BUSL is related to the growing amount of research on learning statistical relational models (Getoor & Taskar, to appear 2007). However, in contrast to the dominant top-down strategy followed by most existing learners (e.g. (Heckerman, 1995)), it approaches the problem of learning MLN structure in a more bottom-up way. In this respect, it is more closely related to algorithms such as GSMN (Bromberg et al., 2006) that base the structure construction on independence tests among the variables. Because BUSL starts by limiting the search space in a bottom-up fashion and then resorts to searching within that constrained space, it is also related to hybrid top-down/bottom-up ILP algorithms (Zelle et al., 1994; Muggleton, 1995).

8. Conclusions and Future work

We have presented a novel algorithm for learning the structure of Markov logic networks by approaching the problem in a more bottom-up fashion and using the data to guide both the construction and evaluation of candidates. We demonstrate the effectiveness of our algorithm in three real-world domains. One limitation of BUSL is that, like TDSL, it cannot learn clauses that include functions. Thus, one future work direction is to extend BUSL to provide support for this capability by constructing the TNodes using least general generalization, which includes a principled way of handling functions (Lavrač & Džeroski, 1994). A second avenue for future work is to adapt BUSL so that it can be used for revision of a provided MLN. Finally, our encouraging results with BUSL suggest applying a similar bottom-up framework to learning of other models, such as Bayesian logic programs (Kersting & De Raedt, 2001).

Acknowledgement

Our thanks go to the members of the ML group at UT Austin, Pedro Domingos, and the three anonymous reviewers for their helpful comments. We also thank the Alchemy team at the University of Washington for their great help with Alchemy. This research is sponsored by the Defense Advanced Research Projects Agency (DARPA) and managed by the Air Force Research Laboratory (AFRL) under contract FA8750-05-2-0283.The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied of DARPA, AFRL, or the United States Government. Most of the experiments were run on the Mastodon Cluster, provided by NSF Grant EIA-0303609.

	Training time			# candidates	
Dataset	BUSL	TDSL	Speed-up	BUSL	TDSL
IMDB	4.59	62.23	13.56	162	7558
UW-CSE	280.31	1127.48	4.02	340	32096
WebKB	272.16	772.09	2.84	341	4643

Table 2. Average training time in minutes, average speed-up factor, and average number of candidates considered by each learner.

References

Bromberg, F., Margaritis, D., & Honavar, V. (2006). Efficient Markov network structure discovery using independence tests. *SDM-06*.

Craven, M., DiPasquo, D., Freitag, D., McCallum, A., Mitchell, T., Nigam, K., & Slattery, S. (1998). Learning to extract symbolic knowledge from the World Wide Web. *AAAI-98*.

Getoor, L., & Taskar, B. (Eds.). (to appear 2007). *Statistical relational learning*. Cambridge, MA: MIT Press.

Heckerman, D. (1995). *A tutorial on learning Bayesian networks* (Technical Report MSR-TR-95-06). Microsoft Research, Redmond, WA.

Kersting, K., & De Raedt, L. (2001). Towards combining inductive logic programming with Bayesian networks. *ILP-01*.

Kok, S., & Domingos, P. (2005). Learning the structure of Markov logic networks. *ICML-2005*.

Kok, S., Singla, P., Richardson, M., & Domingos, P. (2005). *The Alchemy system for statistical relational AI* (Technical Report). Department of Computer Science and Engineering, University of Washington. http://www.cs.washington.edu/ai/alchemy.

Lavrač, N., & Džeroski, S. (1994). *Inductive logic programming: Techniques and applications*. Ellis Horwood.

Muggleton, S. (1995). Inverse entailment and Progol. *New Generation Computing Journal*, *13*, 245–286.

Muggleton, S., & Feng, C. (1992). Efficient induction of logic programs. In S. Muggleton (Ed.), *Inductive logic programming*, 281–297. New York: Academic Press.

Pearl, J. (1988). *Probabilistic reasoning in intelligent systems: Networks of plausible inference*. San Mateo,CA: Morgan Kaufmann.

Poon, H., & Domingos, P. (2006). Sound and efficient inference with probabilistic and deterministic dependencies. *AAAI-2006*.

Quinlan, J. R. (1990). Learning logical definitions from relations. *Machine Learning*, *5*, 239–266.

Richards, B. L., & Mooney, R. J. (1992). Learning relations by pathfinding. *AAAI-92*.

Richardson, M., & Domingos, P. (2006). Markov logic networks. *Machine Learning*, *62*, 107–136.

Russell, S., & Norvig, P. (2003). *Artificial intelligence: A modern approach*. Upper Saddle River, NJ: Prentice Hall. 2 edition.

Van Laer, W., & De Raedt, L. (2001). How to upgrade propositional learners to first order logic: Case study. In S. Džeroski and N. Lavrač (Eds.), *Relational data mining*, 235–256. Berlin: Springer Verlag.

Zelle, J. M., Mooney, R. J., & Konvisser, J. B. (1994). Combining top-down and bottom-up methods in inductive logic programming. *ICML-94*.