Using Structural Motifs for Learning Markov Logic Networks

Stanley Kok **Pedro Domingos**
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195-2350, U.S.A.
{*koks, pedrod*}@*cs.washington.edu*

Abstract

Markov logic networks (MLNs) use first-order formulas to define features of Markov networks. Current MLN structure learners can only learn short clauses (4-5 literals) due to extreme computational costs, and thus are unable to represent complex regularities in data. To address this problem, we present LSM, the first MLN structure learner capable of efficiently and accurately learning long clauses. LSM is based on the observation that relational data typically contains patterns that are variations of the same *structural motifs*. By constraining the search for clauses to occur within motifs, LSM can greatly speed up the search and thereby reduce the cost of finding long clauses. LSM uses *random walks* to identify densely connected objects in data, and groups them and their associated relations into a motif. Our experiments on three real-world datasets show that our approach is 2-5 orders of magnitude faster than the state-of-the-art ones, while achieving the same or better predictive performance.

Introduction

Markov logic networks (MLNs; Domingos & Lowd, 2009) have gained traction in the AI community in recent years because of their ability to combine the expressiveness of first-order logic with the robustness of probabilistic representations. An MLN is a set of weighted first-order formulas, and learning its structure consists of learning both formulas and their weights. Learning MLN structure from data is an important task because it allows us to discover novel knowledge, but it is also a challenging one because of its super-exponential search space. Hence only a few practical approaches have been proposed to date (Kok & Domingos, 2005; Mihalkova & Mooney, 2007; Biba et al., 2008b; Kok & Domingos, 2009; etc.).

These approaches can be categorized according to their search strategies: top-down versus bottom-up. Top-down approaches (e.g., Kok & Domingos, 2005) systematically enumerate formulas and greedily select those with good empirical fit to data. Such approaches are susceptible to local optima, and their search over the large space of formulas is computationally expensive. To overcome these drawbacks, bottom-up approaches (e.g., Mihalkova and Mooney, 2007) use the data to constrain the space of formulas. They find paths of true atoms that are linked via their arguments, and generalize them into first-order formulas. Each path thus corresponds to a conjunction that is true at least once in the data, and since most conjunctions are false, this focuses the search on regions with promising formulas. However, such approaches amount to intractable search over an exponential number of paths. In short, none of the approaches can tractably learn long formulas.

Learning long formulas is important for two reasons. First, long formulas can capture more complex dependencies than short ones. Second, when we lack domain knowledge, we typically want to set the maximum formula length to a large value so as not to *a priori* preclude any good rule.

In this paper, we present *Learning using Structural Motifs* (*LSM*) (Kok and Domingos 2010), an approach that can find long formulas (i.e., formulas with more than 4 or 5 literals). Its key insight is that relational data usually contains recurring patterns, which we term *structural motifs*. These motifs confer three benefits. First, by confining its search to *within* motifs, LSM need not waste time following spurious paths *between* motifs. Second, LSM only searches in each unique motif once, rather than in all its occurrences in the data. Third, by creating various motifs over a set of objects, LSM can capture different interactions among them. A structural motif is frequently characterized by objects that are densely connected via many paths, allowing us to identify motifs using the concept of *truncated hitting time* in *random walks*. This concept has been used in many applications, and we are the first to successfully apply it to learning MLN formulas.

The remainder of the paper is organized as follows. We begin by reviewing some background in Section . Then we describe LSM in detail (Section), and report our experiments (Section). Next we discuss related work (Section). Finally, we conclude with future work (Section).

Background

We review the building blocks of our algorithm: Markov logic, random walks, truncated hitting times, and the LHL system (Kok and Domingos 2009).

Markov Logic

Markov logic is a probabilistic extension of first-order logic (Genesereth and Nilsson 1987). A *Markov logic net-*

work (MLN) is a set of weighted first-order formulas. Together with a set of constants representing objects in a domain, it defines a Markov network (Pearl 1988) with one node per ground atom and one feature per ground formula. The weight of a feature is the weight of the first-order formula that originated it. The probability distribution over possible worlds x specified by the ground Markov network is given by

$$P(X = x) = \frac{1}{Z} \exp\left(\sum_{i \in F} \sum_{j \in G_i} w_i g_j(x)\right) \quad (1)$$

where Z is a normalization constant, F is the set of all first-order formulas in the MLN, G_i is the set of groundings of the ith first-order formula, and $g_j(x) = 1$ if the jth ground formula is true and $g_j(x) = 0$ otherwise. Markov logic can compactly represent complex models in non-i.i.d. domains.

Random Walks and Hitting Times

Random walks and truncated hitting times are defined in terms of *hypergraphs*. A hypergraph is a straightforward generalization of a graph in which an edge can link any number of nodes, rather than just two. Formally, a hypergraph G is a pair (V, E) where V is a set of nodes, and E is a set of labeled non-empty subsets of V called hyperedges. A *path* of length t between nodes u and u' is a sequence of nodes and hyperedges $(v_0, e_0, v_1, e_1, \ldots, e_{t-1}, v_t)$ such that $u = v_0$, $u' = v_t$, $e_i \in E$, $v_i \in e_i$ and $v_{i+1} \in e_i$ for $i \in \{0, \ldots, t-1\}$. u is said to be *reachable* from u' iff there is a path from u to u'. G is *connected* iff all its nodes are reachable from each other. p_s^v denotes a path from s to v.

In a *random walk* (Lovasz 1996), we travel from node to node via hyperedges. Suppose that at some time step we are at node i. In the next step, we move to one of its neighbors j by first randomly choosing a hyperedge e from the set E_i of hyperedges that are incident to i, and then randomly choosing j from among the nodes that are connected by e (excluding i). The probability of moving from i to j is called the *transition probability* p_{ij}, and is given by $p_{ij} = \sum_{e \in E_i \cap E_j} \frac{1}{|E_i|} \frac{1}{|e|-1}$. The *truncated hitting time* h_{ij}^T (Sarkar, Moore, and Prakash 2008) from node i to j is defined as the average number of steps required to reach j for the first time starting from i in a random walk limited to at most T steps. The larger the number of paths between i and j, and the shorter the paths, the smaller h_{ij}^T. Thus, truncated hitting time is useful for capturing the notion of 'closeness' between nodes. It is recursively defined as $h_{ij}^T = 1 + \sum_k p_{ik} h_{kj}^{T-1}$. $h_{ij}^T = 0$ if $i = j$ or $T = 0$, and $h_{ij}^T = T$ if j is not reached in T steps. Sarkar et al. showed that h_{ij}^T can be approximated accurately with high probability by sampling. They run W independent length-T random walks from node i. In w of these runs, node j is visited for the first time at time steps $t_j^1, \ldots, t_j^w$. The *estimated* truncated hitting time is given by

$$\hat{h}_{ij}^T = (1/W) \sum_{k=1}^{w} t_j^k + (1 - w/W)T. \quad (2)$$

Learning via Hypergraph Lifting (LHL)

LHL is a state-of-the-art algorithm for learning MLNs. It consists of three components: LiftGraph, FindPaths, and CreateMLN. LSM uses the last two.

In LiftGraph, LHL represents a database as a hypergraph with constants as nodes and true ground atoms as hyperedges. LHL defines a model in Markov logic, and finds a *single* global clustering of nodes and hyperedges that optimizes the joint likelihood of the database under the model. The resulting hypergraph has fewer nodes and hyperedges, and therefore fewer paths, ameliorating the cost of finding paths in the next component. In LHL, two nodes v and v' are clustered together if they are related to many common nodes. Thus, intuitively, LHL is making use of length-2 paths to determine the similarity of nodes. In contrast, LSM uses longer paths, and thus more information, to find various clusterings of nodes (motifs) rather than just a global one. Also note that spurious edges present in LHL's initial hypergraph are retained in the clustered one.

In FindPaths, LHL uses a variant of relational pathfinding (Richards and Mooney 1992). LHL iterates over the hyperedges in the clustered hypergraph. For each hyperedge, it begins by adding it to an empty path, and then recursively adds hyperedges linked to nodes already present in the path. Its search terminates when the path reaches a length limit or when no new hyperedge can be added. Each time a hyperedge is added to the path, FindPaths stores the resulting path as a new one. Note that each path corresponds to a conjunction of ground atoms.

In CreateMLN, LHL creates a clause from each path by replacing each unique node with a variable, and converting each hyperedge into a negative literal[1]. In addition, LHL adds clauses with the signs of up to n literals flipped. Each clause is then evaluated using weighted pseudo-log-likelihood (WPLL; Kok and Domingos, 2005). WPLL estimates the log-likelihood as a sum over the conditional log-likelihood of every ground atom given its Markov blanket (weighting all first-order predicates equally). Rather than summing over all atoms, LHL estimates the WPLL by sampling θ_{atoms} of them. The WPLL score of a clause is penalized with a length penalty $-\pi d$ where d is the number of atoms in a clause. LHL iterates over the clauses from shortest to longest. For each clause, LHL compares its WPLL against those of its sub-clauses (considered separately) that have already been retained. If the clause scores higher than all of these, it is retained. Finally, LHL greedily adds the retained clauses to an MLN.

Learning Using Structural Motifs

We call our algorithm Learning using Structural Motifs (LSM; Algorithm 1). The crux of LSM is that relational data frequently contains recurring patterns of densely connected objects, and by limiting our search to within these patterns, we can find good rules quickly. We call such patterns *structural motifs*.

[1] In Markov logic, a conjunction of positive literals is equivalent to a disjunction of negative literals with its weight negated.

Table 1: LSM

Input: $G = (V, E)$, a ground hypergraph representing a database
Output: MLN, a set of weighted clauses

$Motifs \leftarrow \emptyset$
For each $s \in V$
  Run N_{walks} random walks of length T from s to estimate h^T_{sv} for all $v \in V$
  Create V_s to contain nodes whose $h^T_{sv} < \theta_{hit}$
  Create E_s to contain hyperedges that only connect to V_s
  Partition V_s into $\{A_1, \dots A_l\}$ where $\forall v \in A_j, \exists v' \in A_j$:
    $|h^T_{sv} - h^T_{sv'}| < \theta_{sym}$
  $\mathcal{V}_s \leftarrow \emptyset$
  For each $A_i \in \{A_1, \dots A_l\}$
    Partition A_i into $H = \{H_1, \dots, H_m\}$ so that symmetrical nodes in A_i
      belong to the same $H_j \in H$
    Add $H_1, \dots, H_m$ to $\mathcal{V}_s$
  Create $\mathcal{E}_s = \{E_1, \dots, E_k\}$ where hyperedges in E with the same label,
    and that connect to the same sets in $\mathcal{V}_s$ belong to the same $E_j \in \mathcal{E}_s$
  Let lifted hypergraph $L = (\mathcal{V}_s, \mathcal{E}_s)$
  Create $Motif(L)$ using DFS, add it to $Motifs$
For each $m \in Motifs$
  Let n_m be the number of unique true groundings returned by DFS for m
  If $n_m < \theta_{motif}$, remove m from $Motifs$
$Paths \leftarrow$ FindPaths($Motifs$)
$MLN \leftarrow$ CreateMLN($Paths$)
Return MLN

A structural motif is a set of literals, which defines a set of clauses that can be created by forming disjunctions over the negations/non-negations of one or more of the literals. Thus, it defines a subspace within the space of all clauses. LSM discovers subspaces where literals are densely connected, and groups them into a motif. To do so, LSM views a database as a hypergraph with constants as nodes, and true ground atoms as hyperedges. Each hyperedge is labeled with a predicate symbol. LSM groups nodes that are densely connected by many paths, and the hyperedges connecting them into a motif. Then it compresses the motif by clustering nodes into high-level concepts, reducing the search space of clauses in the motif. Next it quickly estimates whether the motif appears often enough in the data to be retained. Finally, LSM runs relational pathfinding on each motif to find candidate rules, and retains the good ones in an MLN.

Figure 1 provides an example of a graph created from a university database describing two departments. The bottom motifs are extracted from the top graph. Note that the motifs have gotten rid of the spurious link between departments, preventing us from tracing paths straddling departments that do not translate to good rules. Also note that by searching only once in each unique motif, we avoid duplicating the search in all its occurrences in the graph. Observe that both motifs are created from each department's subgraph. In the left motif, individual students and books are clustered into high-level concepts *Student* and *Book* because they are indistinguishable with respect to professor $P1$ (they have symmetrical paths from $P1$). In the right motif, the clustering is done with respect to book $B1$. LSM's ability to create different motifs over a set of objects allows it to capture various interactions among the objects, and thus to potentially discover more good rules.

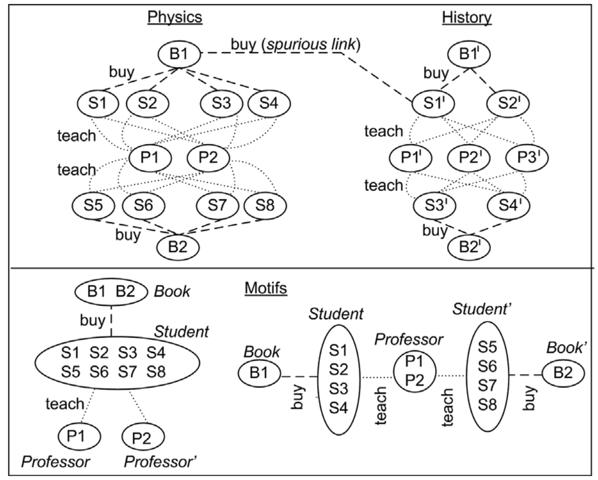

Figure 1: Motifs extracted from a ground hypergraph.

Preliminaries

We define some terms and state a propositionthat are used by our algorithm. A *ground* hypergraph $G = (V, E)$ has constants as nodes, and true ground atoms as hyperedges. An r-hyperedge is a hyperedge labeled with predicate symbol r. There cannot be two or more r-hyperedges connected to a set of nodes because they correspond to the same ground atom. $\sigma(p)$ refers to the string that is created by replacing nodes in path p with integers indicating the order in which the nodes are first visited, and replacing hyperedges with their predicate symbols. Nodes which are visited simultaneously via a hyperedge have their order determined by their argument positions in the hyperedge. Two paths p and p' are *symmetrical* iff $\sigma(p) = \sigma(p')$. Nodes v and v' are symmetrical relative to s, denoted as $Sym_s(v, v')$, iff there is a bijective mapping between the set of all paths from s to v and the set of all paths from s to v' such that each pair of mapped paths are symmetrical. Node sets $V = \{v_1, \dots, v_n\}$ and $V' = \{v'_1, \dots, v'_n\}$ are symmetrical iff $Sym_s(v_i, v'_i)$ for $i = 1, \dots, n$. Note that Sym_s is reflexive, symmetric and transitive. Note that symmetrical nodes v and v' have identical truncated hitting times from s. Also note that symmetrical paths p^v_s and $p^{v'}_s$ have the same probability of being sampled respectively from the set of all paths from s to v and the set of all paths from s to v'. $L_{G,s}$ is the 'lifted' hypergraph that is created as follows from a ground hypergraph $G = (V, E)$ whose nodes are all reachable from a node s. Partition V into disjoint subsets $\mathcal{V} = \{V_1, \dots, V_k\}$ such that all nodes with symmetrical paths from s are in the same V_i. Partition E into disjoint subsets $\mathcal{E} = \{E_1, \dots, E_l\}$ such that all r-hyperedges that connect nodes from the same V_i's are grouped into the same E_j, which is also labeled r. $L_{G,s} = (\mathcal{V}, \mathcal{E})$ intuitively represents a high-level concept with each V_i, and an interaction between the concepts with each E_j. Note that $L_{G,s}$ is connected since no hyperedge in E is removed during its construction. Also note that s is in its own $V_s \in \mathcal{V}$ since no other node has the empty path to it.

Proposition 1 *Let v, v' and s be nodes in a ground hypergraph whose nodes are all reachable from s, and $Sym_s(v, v')$. If an r-hyperedge connects v to a node set W, then an r-hyperedge connects v' to a node set W' that is symmetrical to W.*[2]

We create a structural motif $Motif(L_{G,s})$ from $L_{G,s} = (\mathcal{V}, \mathcal{E})$ as follows. We run depth-first search (DFS) on $L_{G,s}$ but treat hyperedges as nodes and vice versa (a straightforward modification), allowing DFS to visit each hyperedge in $\mathcal{E}$ exactly once. Whenever it visits a hyperedge $E_j \in \mathcal{E}$, DFS selects an $e_j \in E_j$ that is connected to a ground node $v_i \in V$ that is linked to the e_i selected in the previous step (e_j exists by Proposition 1). When several e_j's are connected to v_i, it selects the one connected to the smallest number of unique nodes. The selected e_j's are then variabilized (the same variable is used for the same node across e_j's), and added as literals to the set $Motif(L_{G,s})$. Let $Conj(m)$ denote the conjunction formed by conjoining the (positive) literals in motif m. Note that the selected e_j's are connected and form a true grounding of $Conj(Motif(L_{G,s}))$. The true grounding will be used later to estimate the total number of true groundings of $Conj(Motif(L_{G,s}))$ in the data.

Motif Identification

LSM begins by creating a ground hypergraph from a database. Then it iterates over the nodes. For each node i, LSM finds nodes that are symmetrical relative to i. To do so, it has to compare all paths from i to all other nodes, which is intractable. Thus LSM uses an approximation. It runs N_{walks} random walks of length T from i. In each random walk, when a node is visited, the node stores the path p to it as $\sigma(p)$ (up to a maximum of Max_{paths} paths), and records the number of times $\sigma(p)$ is seen. After running all random walks, LSM estimates the truncated hitting time h^T_{iv} from i to each node v that is visited at least once using Equation 2. (Nodes not visited have $h^T_{iv} = T$.) Nodes whose h^T_{iv}'s exceed a threshold $\theta_{hit} < T$ are discarded (these are 'too loosely' connected to i). The remaining nodes and the hyperedges that only connect to them constitute a ground hypergraph G. LSM groups together nodes in G whose h^T_{iv}'s are less than θ_{sym} apart as potential symmetrical nodes.

Within each group, LSM uses greedy agglomerative clustering to cluster symmetrical nodes together. Two nodes are approximated as symmetrical if their distributions of stored paths are similar. Since the most frequently appearing paths are more representative of a distribution, we only use the top N_{top} paths in each node. Path similarity is measured using Jensen-Shannon divergence (Fugledge & Topsoe, 2004; a symmetric version of the Kullback-Leibler divergence). Each node starts in its own cluster. At each step, LSM merges the pair of clusters whose path distributions are most similar. When there is more than one node in a cluster, its path distribution is the average over those of its nodes. The clustering stops when no pair of clusters have divergence less than θ_{js}. Once the clusters of symmetrical nodes are identified, LSM creates lifted hypergraph $L_{G,s}$ and motif $Motif(L_{G,s})$ as described earlier. Then LSM repeats the process for the next node $i + 1$.

After iterating over all nodes, LSM will have created a set of motifs. It then estimates how often a motif m appears in the data by computing a lower bound n_m on the number of true groundings of $Conj(m)$. It sets n_m to the number of unique true groundings of m that are returned by DFS. If n_m is less than a threshold θ_{motif}, the motif is discarded.

PathFinding and MLN Creation

LSM finds paths in each identified motif in the same manner as LHL's FindPath. The paths are limited to a user-specified maximum length. After that, LSM creates candidate clauses from each path in a similar way as LHL's CreateMLN, with a modification. At the end of CreateMLN, rather than adding clauses greedily to an empty MLN (which is susceptible to local optima), LSM adds all clauses to the MLN, finds their optimal weights, and removes those whose weights are less than θ_{wt}. (We use a zero-mean Gaussian prior on each weight. In our experiments, we use this modification for LHL too.)

Table 2: Details of datasets.

Dataset	Types	Constants	Predicates	True Atoms	Total Atoms
IMDB	4	316	6	1224	17,793
UW-CSE	9	929	12	2112	260,254
Cora	5	3079	10	42,558	687,422

Experiments

Datasets

Our experiments used three publicly available datasets[3] (Table 2) as in Kok & Domingos (2009). The IMDB dataset (Mihalkova and Mooney 2007) is created from the IMDB.com database, and describes relationships among movies, actors and directors (e.g, `WorkedIn(person, movie)`, etc.). The UW-CSE dataset (Richardson and Domingos 2006) describes an academic department with predicates such as `TaughtBy(course, person, quarter)`, etc. The Cora dataset is a collection of citations to computer science papers, created by Andrew McCallum, and later processed by Singla and Domingos (2006) for the task of deduplicating the citations, and their title, author, and venue fields.

Systems

We compared LSM to three state-of-the-art systems: LHL, BUSL and MSL. We implemented LHL, and used the BUSL and MSL implementations in the Alchemy software package (Kok et al. 2010).

Bottom-up Structure Learner (BUSL). BUSL (Mihalkova and Mooney 2007) finds paths of ground atoms in training data, but restricts itself to very short paths (length 2) for tractability reasons. It variabilizes each ground atom in the path, and constructs a Markov network whose nodes are the paths viewed as Boolean variables (conjunctions of

[2]The proof and DFS pseudocode are given in an online appendix at http://alchemy.cs.washington.edu/papers/kok10/.

[3]Available at http://alchemy.cs.washington.edu.

atoms). For each node, BUSL finds nodes connected to it by greedily adding and removing nodes from its Markov blanket using the χ^2 measure of dependence. From the maximal cliques thus created in the Markov network, BUSL creates clauses. For each clique, it forms disjunctions of the atoms in the clique's nodes, and creates clauses with all possible negation/non-negation combinations of the atoms. BUSL computes the WPLL of the clauses, and greedily adds them one at a time to an MLN. This makes BUSL susceptible to local optima. Thus we modified BUSL to use LSM's CreateMLN algorithm to add clauses to the MLN. (Empirically, the modification allowed more good clauses to be included in the MLN.)

Markov Logic Structure Learner (MSL). We used the beam search version of MSL (Kok and Domingos 2005) in Alchemy. MSL maintains a set of n clauses that give the best score improvement over the current MLN. MSL creates all possible clauses of length two, and adds the n highest-scoring clauses to the set. It then repeatedly adds literals to the clauses in the set, and evaluates the WPLL of the newly formed clauses, always maintaining the n highest-scoring ones in the set. When none can be added to the set, it adds the best performing clause in the set to the MLN. It then restarts the search from an empty set. MSL terminates when it cannot find a clause that improves upon the current MLN's WPLL.

We ran each system with two limits on clause length. The short limit is set to 5 (IMDB, UW-CSE) and 4 (Cora). The long limit is set to 10. Systems with the short and long limits are respectively appended with '-S' and '-L'. For the short limit, we allowed LSM, LHL, and BUSL to create more candidate clauses from a candidate containing only negative literals by non-negating the literals in all possible ways. For the long limit, we permitted a maximum of two non-negations to avoid generating too many candidates. Following Kok & Domingos (2009), we disallowed clauses with variables that only appeared once, since these were unlikely to be useful. To investigate the contribution of our motif identification algorithm, we removed it to give the system LSM-NoMot, which found paths directly on the ground hypergraph created from a database. Altogether, we compared ten systems.

The LSM parameter values were: $N_{walks} = 15,000$, $T = 5$, $\theta_{hit} = 4.9$, $\theta_{sym} = 0.1$, $\theta_{js} = 1$, $N_{top} = 3$, $Max_{paths} = 100$, $\theta_{motif} = 10$, $\pi = 0.1$ (IMDB) and 0.01 (UW-CSE, Cora), $\theta_{atoms} = 0.5$, $\theta_{wt} = 0.01$. The other systems had their corresponding parameters set to the same values, and their other parameters set to default values. The parameters were set in an *ad-hoc* manner, and per-fold optimization using a validation set could conceivably yield better results. All systems ran on identically configured machines (2.3GHz, 16GB RAM, 4096KB CPU cache) for a maximum of 28 days.

Methodology

For each dataset, we performed cross-validation using the five previously defined folds. For IMDB and UW-CSE, we performed inference over the groundings of each predicate to compute their probabilities of being true, using the groundings of all other predicates as evidence. For Cora, we ran inference over each of the four predicates `SameCitation`, `SameTitle`, `SameAuthor`, and `SameVenue` in turn, using the groundings of all other predicates as evidence. We also ran inference over all four predicates together, which is a more challenging task than inferring each individually. We denote this task as "Cora (Four Predicates)". To obtain the best possible results for an MLN, we relearned its clause weights for each query predicate (or set of query predicates in the case of Cora) before performing inference. This accounts for the differences in our results from those reported by Kok & Domingos (2009). We used Alchemy's Gibbs sampling for all systems. Each run of the inference algorithms drew 1 million samples, or ran for a maximum of 24 hours, whichever came earlier. To evaluate the performance of the systems, we measured the average conditional log-likelihood of the test atoms (CLL), and the area under the precision-recall curve (AUC).

Results

Tables 3 and 4 report AUCs, CLLs and runtimes. The AUC and CLL results are averages over all atoms in the test sets and their standard deviations. Runtimes are averages over the five folds.

We first compare LSM to LHL. The results indicate that LSM scales better than LHL, and that LSM equals LHL's predictive performance on small simple domains, but surpasses LHL on large complex ones. LSM-S is marginally slower than LHL-S on the smallest dataset, but is faster on the two larger ones. The scalability of LSM becomes clear when the systems learn long clauses: LSM-L is consistently 100-100,000 times faster than LHL-L on all datasets.[4] Note that LSM-L performs better than LSM-S on AUC and CLL, substantiating the importance of learning long rules.

We next compare LSM to MSL and BUSL. LSM consistently outperforms MSL on AUC and CLL for both short and long rules; and draws with BUSL on UW-CSE, but does better on IMDB and Cora. In terms of runtime, the results are mixed. Observe that BUSL and MSL have similar runtimes when learning both short and long rules (with the exception of MSL on UW-CSE). Tracing the steps taken by BUSL and MSL, we found that the systems took the same greedy search steps when learning both short and long rules, thus resulting in the same locally optimal MLNs containing only short rules. In contrast, LSM-L found longer rules than LSM-S for all datasets, even though these were only retained by CreateMLN for Cora.

Comparing LSM to LSM-NoMot, we see the importance of motifs in making LSM tractable.

Our runtimes are faster than those reported by Kok & Domingos (2009) because of our modifications to CreateMLN, and our machines are better configured (4 times more RAM, 8 times more CPU cache).

[4] LHL-L on UW-CSE and Cora, and LSM-NoMot-L exceeded the time bound of 28 days. We estimated their runtimes by extrapolating from the number of atoms they had initiated their search from.

Table 3: Area under precision-recall curve (AUC) and conditional log-likelihood (CLL) of test atoms.

	IMDB		UW-CSE		Cora		Cora (Four Predicates)	
System	AUC	CLL	AUC	CLL	AUC	CLL	AUC	CLL
LSM-S	0.71±0.01	−0.06±0.00	0.22±0.01	−0.03±0.00	0.98±0.00	−0.02±0.00	0.92±0.00	−0.42±0.00
LSM-L	0.71±0.01	−0.06±0.00	0.22±0.01	−0.03±0.00	0.98±0.00	−0.02±0.00	0.97±0.00	−0.23±0.00
LSM-NoMot-S	0.71±0.01	−0.06±0.00	0.23±0.01	−0.03±0.00	0.98±0.00	−0.02±0.00	0.93±0.00	−0.38±0.00
LSM-NoMot-L	0.34±0.01	−0.18±0.00	0.13±0.01	−0.04±0.00	0.57±0.00	−0.29±0.00	0.47±0.00	−0.94±0.00
LHL-S	0.71±0.01	−0.06±0.00	0.21±0.01	−0.03±0.00	0.95±0.00	−0.04±0.00	0.76±0.00	−0.88±0.00
LHL-L	0.71±0.01	−0.06±0.00	0.13±0.01	−0.04±0.00	0.57±0.00	−0.29±0.00	0.47±0.00	−0.94±0.00
BUSL-S	0.48±0.01	−0.11±0.00	0.22±0.01	−0.03±0.00	0.57±0.00	−0.29±0.00	0.47±0.00	−0.94±0.00
BUSL-L	0.48±0.01	−0.11±0.00	0.22±0.01	−0.03±0.00	0.57±0.00	−0.29±0.00	0.47±0.00	−0.94±0.00
MSL-S	0.38±0.01	−0.17±0.00	0.19±0.01	−0.04±0.00	0.57±0.00	−0.29±0.00	0.47±0.00	−0.94±0.00
MSL-L	0.38±0.01	−0.17±0.00	0.18±0.01	−0.04±0.00	0.57±0.00	−0.29±0.00	0.47±0.00	−0.94±0.00

Table 4: System runtimes. The times for Cora (Four Predicates) are the same as for Cora.

System	IMDB (hr)	UW-CSE (hr)	Cora (hr)
LSM-S	0.21±0.02	1.38±0.3	1.33±0.03
LSM-L	0.31±0.04	4.52±2.35	20.57±7.29
LSM-NoMot-S	1.09±0.22	50.83±18.33	332.82±60.54
LSM-NoMot-L	160,000±12,000	280,000±35,000	5,700,000±10^5
LHL-S	0.18±0.02	5.29±0.81	1.92±0.02
LHL-L	73.45±11.71	120,000±13,000	230,000±7000
BUSL-S	0.03±0.01	2.77±1.06	1.83±0.04
BUSL-L	0.03±0.01	2.77±1.06	1.83±0.04
MSL-S	0.02±0.01	1.07±0.21	9.96±1.59
MSL-L	0.02±0.01	26.22±26.14	9.81±1.50

Related Work

Huynh and Mooney (2008), and Biba et al. (2008a) proposed discriminative structure learning algorithms for MLNs. These algorithms learn clauses that predict a single target predicate, unlike LSM, which models the full joint distribution of the predicates. Relational association rule mining systems (e.g., De Raedt & Dehaspe, 1997) differ from LSM by learning clauses without first learning motifs, and are not as robust to noise (they do not use statistical models).

Random walks and hitting times have been successfully applied to a variety of applications, e.g., social network analysis (Liben-Nowell and Kleinberg 2003), word dependency estimation (Toutanova, Manning, and Ng 2004), collaborative filtering (Brand 2005), and search engine query expansion (Mei, Zhou, and Church 2008).

Conclusion and Future Work

We presented LSM, the first MLN structure learner that is able to learn long clauses. LSM tractably learns long clauses by finding motifs of densely connected objects in data, and restricting its search for clauses to within the motifs. Our empirical comparisons with three state-of-the-art systems on three datasets demonstrate the effectiveness of LSM.

As future work, we want to apply LSM to larger domains; discover motifs at multiple granularities; incorporate bottom-up (Muggleton and Feng 1990) and hybrid top-down/bottom-up techniques (Muggleton 1995); etc.

Acknowledgments

This research was partly funded by ARO grant W911NF-08-1-0242, AFRL contract FA8750-09-C-0181, DARPA contracts FA8750-05-2-0283, FA8750-07-D-0185, HR0011-06-C-0025, HR0011-07-C-0060 and NBCH-D030010, NSF grants IIS-0534881 and IIS-0803481, and ONR grant N00014-08-1-0670. 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 ARO, DARPA, NSF, ONR, or the United States Government.

References

Biba, M.; Ferilli, S.; and Esposito, F. 2008a. Discriminative structure learning of Markov logic networks. In *Proc. ILP'08*.

Biba, M.; Ferilli, S.; and Esposito, F. 2008b. Structure learning of Markov logic networks through iterated local search. In *Proc. ECAI'08*.

Brand, M. 2005. A random walks perspective on maximizing satisfaction and profit. In *Proc. of the 8th SIAM Conf. on Opt.*

De Raedt, L., and Dehaspe, L. 1997. Clausal discovery. *Machine Learning* 26.

Domingos, P., and Lowd, D. 2009. *Markov Logic: An Interface Layer for AI*. Morgan & Claypool.

Fugledge, B., and Topsoe, F. 2004. Jensen-Shannon divergence and Hilbert space embedding. In *IEEE Int. Sym Info. Theory*.

Genesereth, M. R., and Nilsson, N. J. 1987. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann.

Huynh, T. N., and Mooney, R. J. 2008. Discriminative structure and parameter learning for Markov logic networks. In *Proc. ICML'08*.

Kok, S., and Domingos, P. 2005. Learning the structure of Markov logic networks. In *Proc. ICML'05*.

Kok, S., and Domingos, P. 2009. Learning Markov logic network structure via hypergraph lifting. In *Proc. ICML'09*.

Kok, S., and Domingos, P. 2010. Learning Markov logic networks using structural motifs. In *Proc. ICML'10*.

Kok, S.; Sumner, M.; Richardson, M.; Singla, P.; Poon, H.; Lowd, D.; Wang, J.; and Domingos, P. 2010. The Alchemy system for statistical relational AI. Technical report, Dept. of Comp. Sci. & Eng., Univ. of Washington. http://alchemy.cs.washington.edu.

Liben-Nowell, D., and Kleinberg, J. 2003. The link prediction problem for social networks. In *Proc. of the 12th Int. Conf. on Info. and Know.*

Lovasz, L. 1996. Random walks on graphs: A survey. *Combinatorics* 353–397.

Mei, Q.; Zhou, D.; and Church, K. 2008. Query suggestion using hitting time. In *Proc CIKM'08*.

Mihalkova, L., and Mooney, R. J. 2007. Bottom-up learning of Markov logic network structure. In *Proc. ICML'07*.

Muggleton, S., and Feng, C. 1990. Efficient Induction of Logic Programs. In *Proc. of 1st Conference on Algorithmic Learning Theory*.

Muggleton, S. 1995. Inverse entailment and Progol. *New Generation Computing Journal* 13.

Pearl, J. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann.

Richards, B. L., and Mooney, R. J. 1992. Learning relations by pathfinding. In *Proc. AAAI'92*.

Richardson, M., and Domingos, P. 2006. Markov logic networks. *Machine Learning* 62:107–136.

Sarkar, P.; Moore, A. W.; and Prakash, A. 2008. Fast incremental proximity search in large graphs. In *Proc. ICML'08*.

Singla, P., and Domingos, P. 2006. Entity resolution with Markov logic. In *Proc. ICDM'06*.

Toutanova, K.; Manning, C. D.; and Ng, A. Y. 2004. Learning random walk models for inducing word dependency distributions. In *Proc. ICML'04*.