Learning Markov Logic Network Structure via Hypergraph Lifting

Stanley Kok KOKS@CS.WASHINGTON.EDU
Pedro Domingos PEDROD@CS.WASHINGTON.EDU
Department of Computer Science & Engineering, University of Washington, Seattle, WA 98195, USA

Abstract

Markov logic networks (MLNs) combine logic and probability by attaching weights to first-order clauses, and viewing these as templates for features of Markov networks. Learning MLN structure from a relational database involves learning the clauses and weights. The state-of-the-art MLN structure learners all involve some element of greedily generating candidate clauses, and are susceptible to local optima. To address this problem, we present an approach that directly utilizes the data in constructing candidates. A relational database can be viewed as a hypergraph with constants as nodes and relations as hyperedges. We find paths of true ground atoms in the hypergraph that are connected via their arguments. To make this tractable (there are exponentially many paths in the hypergraph), we *lift* the hypergraph by jointly clustering the constants to form higher-level concepts, and find paths in it. We variabilize the ground atoms in each path, and use them to form clauses, which are evaluated using a pseudo-likelihood measure. In our experiments on three real-world datasets, we find that our algorithm outperforms the state-of-the-art approaches.

1. Introduction

In recent years, there has been a surge of interest in combining statistical and relational learning approaches (Getoor & Taskar, 2007), driven by the realization that many applications require both. Recently, Richardson and Domingos (2006) introduced Markov logic networks (MLNs), a statistical relational language combining first-order logic and Markov networks. An MLN consists of weighted first-order logic formulas, viewed as templates for Markov network features. Learning MLN structure is an important but challenging task, and to date only a few approaches have been proposed (Kok & Domingos, 2005; Mihalkova & Mooney, 2007; Biba et al., 2008b; etc.).

Most of these approaches systematically enumerate candidate clauses by starting from an empty clause, greedily adding literals to it, and testing the resulting clause's empirical fit to training data. Such a strategy has two shortcomings: searching the large space of clauses is computationally expensive; and it is susceptible to converging to a local optimum, missing potentially useful clauses. These shortcomings can be ameliorated by using the data to *a priori* constrain the space of candidates. This is the basic idea in *relational pathfinding* (Richards & Mooney, 1992), which finds paths of true ground atoms that are linked via their arguments and then generalizes them into first-order rules. Each path corresponds to a conjunction that is true at least once in the data. Since most conjunctions are false, this helps to concentrate the search on regions with promising rules. However, pathfinding potentially amounts to exhaustive search over an exponential number of paths. Hence, systems using relational pathfinding (e.g., BUSL (Mihalkova & Mooney, 2007)) typically restrict themselves to very short paths, creating short clauses from them and greedily joining them into longer ones.

In this paper, we present LHL, an approach that uses relational pathfinding to a fuller extent than previous ones. It mitigates the exponential search problem by first inducing a more compact representation of data, in the form of a hypergraph over clusters of constants. Pathfinding on this 'lifted' hypergraph is typically at least an order of magnitude faster than on the ground training data, and produces MLNs that are more accurate than previous state-of-the-art approaches. LHL is short for Learning via Hypergraph Lifting.

We begin by reviewing Markov logic in the next section. We then describe our structure learning algorithm (Section 3) and report our experiments with it (Section 4). Finally, we discuss related work (Section 5) and future work (Section 6).

Appearing in *Proceedings of the 26th International Conference on Machine Learning*, Montreal, Canada, 2009. Copyright 2009 by the author(s)/owner(s).

2. Markov Logic

In first-order logic (Genesereth & Nilsson, 1987), formulas are constructed using four types of symbols: constants, variables, functions, and predicates. (In this paper we use only function-free logic.) Constants represent objects in a domain of discourse (e.g., people: `Anna`, `Bob`, etc.). Variables (e.g., `x`, `y`) range over the objects. Predicates represent relations among objects (e.g., `Advises`), or attributes of objects (e.g., `Student`). (In this paper, we use *predicate* and *relation* interchangeably.) Variables and constants may be typed. An *atom* is a predicate symbol applied to a list of arguments, which may be variables or constants (e.g., $\texttt{Advises}(\texttt{x}, \texttt{Bob})$). A *ground atom* is an atom all of whose arguments are constants. A *world* is an assignment of truth values to all possible ground atoms. A *database* is a partial specification of a world; each atom in it is true, false or (implicitly) unknown. In this paper, we make a closed-world assumption: a ground atom not in the database is assumed to be false. A *clause* is a disjunction of non-negated/negated atoms.

A *Markov network* (Pearl, 1988) represents the joint distribution of a set of variables $X = (X_1, \ldots, X_n) \in \mathcal{X}$ as a product of factors: $P(\mathbf{X}{=}\mathbf{x}) = \frac{1}{Z}\prod_k f_k(\mathbf{x}_k)$, where each factor f_k is a non-negative function of a subset of the variables $\mathbf{x}_k$, and Z is a normalization constant. As long as $P(\mathbf{X}{=}\mathbf{x}) > 0$ for all x, the distribution can be equivalently represented as a *log-linear model*: $P(\mathbf{X}{=}\mathbf{x}) = \frac{1}{Z}\exp\left(\sum_i w_i g_i(\mathbf{x})\right)$, where the *features* $g_i(\mathbf{x})$ are arbitrary functions of (a subset of) the variables' state.

A *Markov logic network (MLN)* is a set of weighted first-order formulas. Together with a set of constants representing objects in the domain, it defines a Markov network with one variable per ground atom and one feature per ground formula. The probability distribution over possible worlds x is given by $P(X = x) = \frac{1}{Z}\exp(\sum_{i \in F}\sum_{j \in G_i} w_i g_j(x))$ where Z is the partition function, 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 enables us to compactly represent complex models in non-i.i.d. domains. General algorithms for inference and learning in Markov logic are discussed in Richardson and Domingos (2006).

3. Learning via Hypergraph Lifting

We call our algorithm LHL, for Learning via Hypergraph Lifting. In LHL, we make use 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. More formally, we define a hypergraph as a pair (V, E) where V is a set of nodes, and E is a multiset of labeled non-empty ordered subsets of V called hyperedges. In LHL, we find *paths* in a hypergraph. A path is defined as a set of hyperedges such that for any two hyperedges e_0 and e_n in the set, there exists an ordering of (a subset of) hyperedges in the set $e_0, e_1, \ldots, e_{n-1}, e_n$ such that e_i and e_{i+1} share at least one node.

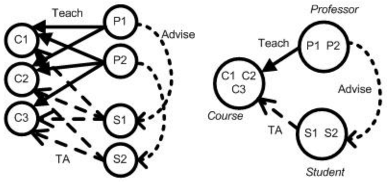

Figure 1. Lifting a hypergraph.

A database can be viewed as a hypergraph with constants as nodes, and true ground atoms as hyperedges. Each hyperedge is labeled with a predicate symbol. Nodes (constants) are linked by a hyperedge (true ground atom) if and only if they appear as arguments in the hyperedge. (Henceforth we use *node* and *constant* interchangeably, and likewise for *hyperedge* and *true ground atom.*) A path of hyperedges can be generalized into a first-order clause by variabilizing their arguments. To avoid tracing the exponential number of paths in the hypergraph, LHL first jointly clusters the nodes into higher-level concepts, and by doing so it also clusters the hyperedges (i.e., the ground atoms containing the clustered nodes). The 'lifted' hypergraph has fewer nodes and hyperedges, and therefore fewer paths, reducing the cost of finding them.

Figure 1 provides an example. We have a database describing an academic department where professors tend to have students whom they are advising as teaching assistants (TAs) in the classes the professors are teaching. The left graph is created from the database, and after lifting, results in the right graph. Observe that the lifted graph is simpler and the clustered constants correspond to the high-level concepts of *Professor*, *Student*, and *Course*.

Algorithm 1 gives the pseudocode for LHL. LHL begins by lifting a hypergraph (Algorithm 2). Then it finds paths in the lifted hypergraph (Algorithm 3). Finally it creates candidate clauses from the paths, and learn their weights to create an MLN (Algorithm 4).

3.1. Hypergraph Lifting

We call our hypergraph lifting algorithm LiftGraph. LiftGraph is similar to the MRC and SNE algo-

Algorithm 1 $LHL(D, T, \omega, \mu, \nu, \pi, \pi', \sigma)$

```
input: $D$, a relational database
       $T$, a set of types, where a type is a set of constants
       $\omega$, maximum number of hyperedges in a path
       $\mu$, minimum number of ground atoms per hyperedge in
         a path in order for it to be selected
       $\nu$, maximum number of ground atoms to sample in a path
       $\pi, \pi'$, length penalties on clauses
       $\sigma$, fraction of atoms to sample from $D$
output: $(Clauses, Weights)$, an MLN containing a set of learned
        clauses and their weights
note: Index $H$ maps from each node $\gamma_i$ to the set of
        hyperedges $r(\gamma_1, \ldots, \gamma_i, \ldots, \gamma_n)$ containing $\gamma_i$
      $E$ is a set of hyperedges in a lifted hypergraph
      $Paths$ is a set of paths, each path being a set of hyperedges
$(H, E) \leftarrow LiftGraph(D, T)$
$Paths \leftarrow \emptyset$
for each $r(\gamma_1, \ldots, \gamma_n) \in E$
  $Paths \leftarrow Paths \cup FindPaths(\{r(\gamma_1, \ldots, \gamma_n)\}, \{\gamma_1, \ldots, \gamma_n\}, \omega, H)$
$(Clauses, Weights) \leftarrow CreateMLN(Paths, D, \mu, \nu, \pi, \pi', \sigma)$
return $(Clauses, Weights)$
```

Algorithm 2 $LiftGraph(D, T)$

```
note: The inputs and output are as described in Algorithm 1
for each $t \in T$
  $\Gamma_t \leftarrow \emptyset$
  for each $x \in t$
    $\Gamma_t \leftarrow \Gamma_t \cup \{\gamma_x\}$ ($\gamma_x$ is a unit cluster containing $x$)
    $H[\gamma_x] \leftarrow \emptyset$   ($H$ maps from nodes to hyperedges)
$E \leftarrow \emptyset$   ($E$ contains hyperedges)
for each true ground atom $r(x_1, \ldots, x_n) \in D$
  $E \leftarrow E \cup \{r(\gamma_{x_1}, \ldots, \gamma_{x_n})\}$
  for each $x_i \in \{x_1, \ldots, x_n\}$
    $H[\gamma_{x_i}] \leftarrow H[\gamma_{x_i}] \cup \{r(\gamma_{x_1}, \ldots, \gamma_{x_n})\}$
repeat
  for each $t \in T$
    $(\gamma_{best}, \gamma'_{best}) \leftarrow ClusterPairWithBestGain(\Gamma_t)$
    if $\{(\gamma_{best}, \gamma'_{best})\} \neq \emptyset$
      $\gamma_{new} \leftarrow \gamma_{best} \cup \gamma'_{best}$
      $\Gamma_t \leftarrow (\Gamma_t \setminus \{\gamma_{best}, \gamma'_{best}\}) \cup \gamma_{new}$
      $H[\gamma_{new}] \leftarrow \emptyset$
      for each $\gamma \in \{\gamma_{best}, \gamma'_{best}\}$
        for each $r(\gamma_1, \ldots, \gamma, \ldots, \gamma_n) \in H[\gamma]$
          $H[\gamma_{new}] \leftarrow H[\gamma_{new}] \cup \{r(\gamma_1, \ldots, \gamma_{new}, \ldots, \gamma_n)\}$
          $E \leftarrow E \setminus \{r(\gamma_1, \ldots, \gamma, \ldots, \gamma_n)\}$
          $E \leftarrow E \cup \{r(\gamma_1, \ldots, \gamma_{new}, \ldots, \gamma_n)\}$
        $H[\gamma] \leftarrow \emptyset$
until no clusters are merged for all $t$
return $(H, E)$
```

Algorithm 3 $FindPaths(CurPath, V, \omega, H)$

```
input: $CurPath$, set of connected hyperedges
       $V$, set of nodes in $CurPath$
note: The other inputs & output are as described in Algorithm 1
if $|CurPath| = \omega$
  return $\emptyset$
$Paths \leftarrow \emptyset$
for each $\gamma_i \in V$
  for each $r(\gamma_1, \ldots, \gamma_n) \in H[\gamma_i]$
    if $r(\gamma_1, \ldots, \gamma_n) \notin CurPath$
      $CurPath \leftarrow CurPath \cup \{r(\gamma_1, \ldots, \gamma_n)\}$
      $Paths \leftarrow Paths \cup \{CurPath\}$
      $V' \leftarrow \emptyset$
      for each $\gamma_j \in \{\gamma_1, \ldots, \gamma_n\}$
        if $\gamma_j \notin V$
          $V \leftarrow V \cup \{\gamma_j\}$
          $V' \leftarrow V' \cup \{\gamma_j\}$
      $Paths \leftarrow Paths \cup FindPath(CurPath, V, \omega, H)$
      $CurPath \leftarrow CurPath \setminus \{r(\gamma_1, \ldots, \gamma_n)\}$
      $V \leftarrow V \setminus V'$
return $Paths$
```

rithms (Kok & Domingos, 2007; Kok & Domingos, 2008). It differs from them in the following ways. LiftGraph can handle relations of arbitrary arity, whereas SNE can only handle binary relations. Unlike MRC, LiftGraph finds a single clustering of constant symbols rather than multiple clusterings. While both SNE and MRC can cluster predicate symbols, in this paper, for simplicity, we do not cluster predicates. (However, it is straightforward to extend LiftGraph to do so.) Most domains contain many fewer predicates than objects, and structure learning alone suffices to capture the dependencies among them, which is what LHL does. (Because SNE and MRC do not have a structure learning component, it is essential for them to cluster predicates in order to learn the dependencies among them.)

LiftGraph works by jointly clustering the constants in a hypergraph in a bottom-up agglomerative manner, allowing information to propagate from one cluster to another as they are formed. The number of clusters need not be pre-specified. As a consequence of clustering the constants, the ground atoms in which the constants appear are also clustered. Each hyperedge in the lifted hypergraph contains at least one true ground atom.

LiftGraph is defined using Markov logic. We use the variable r to represent a predicate, x_i for the ith argument of a predicate, γ_i for a cluster of ith arguments of a predicate (i.e., a set of constant symbols), and Γ_t for a clustering of constant symbols of type t (i.e., a set of clusters or, equivalently, a partitioning of a set of symbols). If x_i is in γ_i, we say that $(x_1, \ldots, x_n)$ is in the *cluster combination* $(\gamma_1, \ldots, \gamma_n)$, and that $(\gamma_1, \ldots, \gamma_n)$ *contains* the atom $r(x_1, \ldots, x_n)$. $r(\gamma_1, \ldots, \gamma_n)$ denotes a hyperedge connecting nodes $\gamma_1, \ldots, \gamma_n$. A hypergraph representing the true ground atoms $r(x_1, \ldots, x_n)$ in a database is simply $(V = \{\{x_i\}\}, E = \{\, r(\{x_1\}, \ldots, \{x_n\})\,\})$ with each constant x_i in its own cluster, and a hyperedge for each true ground atom.

The learning problem in LiftGraph consists of finding the cluster assignment $\{\Gamma\}$ that maximizes the posterior probability $P(\{\Gamma\}|D) \propto P(\{\Gamma\})P(D|\{\Gamma\})$, where D is a database of truth assignments to the observable $r(x_1, \ldots, x_n)$ ground atoms. The prior $P(\{\Gamma\})$ is simply an MLN containing two rules. The first rule states that each symbol belongs to exactly one cluster. This rule is hard, i.e., it has infinite weight and cannot be violated.

$$\forall x\ \exists^1 \gamma \ \ x \in \gamma$$

The second rule is

$$\forall \gamma_1, \ldots, \gamma_n\, \exists x_1, \ldots, x_n \quad x_1 \in \gamma_1 \wedge \ldots \wedge x_n \in \gamma_n$$

with negative weight $-\infty < -\lambda < 0$, which imposes an exponential prior on the number of cluster combina-

Algorithm 4 $CreateMLN(Paths, D, \mu, \nu, \pi, \pi', \sigma)$

```
calls: $VariabilizePaths(Paths)$, replaces the nodes in each path in
                $Paths$ with variables
       $MakeClauses(Path)$, creates clauses from hyperedges in $Path$
       $Sample(Path, \nu)$, uniformly samples $\nu$ ground atoms from $Path$
       $SampleDB(D, \sigma)$, uniformly samples a fraction $\sigma$ of atoms
                from database $D$
       $NumTrueGroundAtoms(Path)$, counts the number of true
                ground atoms in $Path$
note: The inputs and output are as described in Algorithm 1
(only select paths with enough true ground atoms (heuristic 1))
$Paths \leftarrow VariabilizePaths(Paths)$
$SelectedPaths \leftarrow \emptyset$
for each $p \in Paths$
  if $(NumTrueGroundAtoms(p) >= PathLength(p) * \mu)$
    $SelectedPaths \leftarrow SelectedPaths \cup \{p\}$
(evaluate candidates with ground atoms in $Path$ (heuristic 2))
$CandidateClauses \leftarrow \emptyset$
for each $p \in SelectedPaths$
  $D' \leftarrow Sample(p, \nu)$
  for each $c \in MakeClauses(p)$
    if $Score(c, D') > Score(\emptyset, D')$
      $CandidateClauses \leftarrow CandidateClauses \cup \{c\}$
$CandidateClauses \leftarrow SortByLength(CandidateClauses)$
(Evaluate candidates with ground atoms in database $D$)
$D' \leftarrow SampleDB(D, \sigma)$
$SelectedClauses \leftarrow \emptyset$
for each $c \in CandidateClauses$
  $BetterThanSubClauses \leftarrow True$
  for each $c' \in (SubClauses(c) \cap SelectedClauses)$
    if $Score(c, D') < Score(c', D')$
      $BetterThanSubClauses \leftarrow False$
      break
  if $(BetterThanSubClauses)$
    $SelectedClauses \leftarrow SelectedClauses \cup \{c\}$
$AddClausesToMLN(SelectedClauses)$
$Weights \leftarrow LearnWeights(SelectedClauses)$
return $(SelectedClauses, Weights)$
```

tions to prevent overfitting. The parameter λ is fixed during learning, and is the penalty in log-posterior incurred by adding a cluster combination.

The main MLN for the likelihood $P(D|\{\Gamma\})$ contains the following rules. For each predicate r and each cluster combination $(\gamma_1, \ldots, \gamma_n)$ that contains a true ground atom of r, the MLN contains the rule:

$$\forall x_1, \ldots, x_n \quad x_1 \in \gamma_1 \wedge \ldots \wedge x_n \in \gamma_n \Rightarrow r(x_1, \ldots, x_n)$$

We call these *atom prediction* rules because they state that the truth value of an atom is determined by the cluster combination it belongs to. These rules are soft. At most there can be one such rule for each true ground atom (i.e., when each constant is in its own cluster).

For each predicate r, we create a rule

$$\forall x_1, \ldots, x_n \\ \left(\bigwedge_{i=1}^{m} \neg(x_1 \in \gamma_1^i \wedge \ldots \wedge x_n \in \gamma_n^i)\right) \Rightarrow r(x_1, \ldots, x_n)$$

where $(\gamma_1^1, \ldots, \gamma_n^1), \ldots, (\gamma_1^m, \ldots, \gamma_n^m)$ are cluster combinations containing true ground atoms of r. This rule accounts for all atoms (all false) that are not in any cluster combination with true ground atoms of r. We call such a rule a *default* atom prediction rule because its antecedents are analogous to a default cluster combination that contains all atoms that are not in the cluster combinations of any atom prediction rule.

LiftGraph simplifies the learning problem by performing hard assignments of constant symbols to clusters (i.e., instead of computing probabilities of cluster membership, a symbol is simply assigned to its most likely cluster). The weights and the log-posterior can now be computed in closed form.[1] LiftGraph thus simply searches over cluster assignments, evaluating each one by its posterior probability. It begins by assigning each constant symbol x_i to its own cluster $\{x_i\}$, and creating a hyperedge $r(\{x_1\}, \ldots, \{x_n\})$ for each true ground atom $r(x_1, \ldots, x_n)$. Next it creates candidate pairs of clusters of each type, and for each pair, it evaluates the gain in posterior probability if its clusters are merged. It then chooses the pair that gives the largest gain to be merged. When clusters γ_i and γ_i' are merged to form γ_i^{new}, each hyperedge $r(\gamma_1, \ldots, \gamma_i, \ldots, \gamma_n)$ is replaced with $r(\gamma_1, \ldots, \gamma_i^{new}, \ldots, \gamma_n)$ (and similarly for hyperedges containing γ_i'). Since $r(\gamma_1, \ldots, \gamma_i, \ldots, \gamma_n)$ contains at least one true ground atom, $r(\gamma_1, \ldots, \gamma_i^{new}, \ldots, \gamma_n)$ must do too. To avoid trying all possible candidate pairs of clusters, LiftGraph only tries to merge γ_i and γ_i' if they appear in hyperedges $r(\gamma_1, \ldots, \gamma_i, \ldots, \gamma_n)$ and $r(\gamma_1, \ldots, \gamma_i', \ldots, \gamma_n)$. In this manner, it incrementally merges clusters until no merges can be performed to improve posterior probability. It then returns a lifted hypergraph whose hyperedges all contain at least one true ground atom.

3.2. Path Finding

FindPaths constructs paths by starting from each hyperedge in a hypergraph. It begins by adding a hyperedge to an empty path, and then recursively adds hyperedges linked to nodes already present in the path (hyperedges already in the path are not re-added). Its search terminates when the path reaches a maximum length 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. All the paths are passed on to the next step to create clauses.

3.3. Clause Creation and Pruning

A path in the hypergraph corresponds to a conjunction of $r(\gamma_1, \ldots, \gamma_n)$ hyperedges, and it guarantees that the conjunction has at least one support in the hypergraph (i.e., there is at least one true ground atom in each hyperedge). We replace each γ_i in a path with a variable, thereby creating a variabilized atom for each hyperedge. We convert the conjunction of positive literals to a clause because that is the form that is typically used

[1] See http://alchemy.cs.washington.edu/papers/kok09a for the derivation of the log-posterior.

by ILP (inductive logic programming) and MLN structure learning and inference algorithms. (In Markov logic, a conjunction of positive literals with weight w is equivalent to a clause of negative literals with weight $-w$). In addition, we add clauses with the signs of up to n literals flipped (where n is a user-defined parameter), since the resulting clauses may also be useful. (Notice that if all but one of the literals are negative, this is a definite clause whose antecedent is supported by a path in the hypergraph.)

We evaluate each clause using weighted pseudo-log-likelihood (WPLL) (Kok & Domingos, 2005). WPLL is defined as: $\log P^{\bullet}_{w,F,D}(X = x) = \sum_{r \in R} c_r \sum_{g \in G^D_r} \log P_{w,F}(X_g = x_g | MB_x(X_g))$ where F is a set of clauses, w is a set of clause weights, R is the set of first-order predicates, G^D_r is a set of ground atoms of predicate r in database D, and x_g is the truth value (0 or 1) of ground atom g, and $P_{w,F}(X_g = x_g | MB_x(X_g)) = \frac{\exp\left(\sum_{i \in F} w_i n_i(x)\right)}{\exp\left(\sum_{i \in F} w_i n_i(x_{[X_g=0]})\right) + \exp\left(\sum_{i \in F} w_i n_i(x_{[X_g=1]})\right)}$. $MB_x(X_g)$ is the state of X_g's *Markov blanket* in the data, $n_i(x)$ is the number of true groundings of the ith clause in x, $n_i(x_{[X_g=0]})$ is the number of true groundings of the ith clause when we force $X_g = 0$ and leave the remaining data unchanged, and similarly for $n_i(x_{[X_g=1]})$. Following Kok & Domingos, we set $c_r = 1/|G^D_r|$ to weight all first-order predicates equally, and penalize the WPLL with a length penalty $-\pi d$, where d is the number of atoms in a clause. Summing over all ground atoms in WPLL is computationally expensive, so we only sum over a randomly-sampled fraction σ of them. We define the score of a clause c as $Score(c, D) = \log P^{\bullet}_{w',F',D}(X = x) - \pi d$, where F' is a set containing c and one unit clause for each predicate in R, and w' is a set of optimal weights for the clauses in F'.

We iterate over the clauses from shortest to longest. For each clause, we compare its scores against those of its sub-clauses (considered separately) that have already been retained. If the clause scores higher than all of these sub-clauses, it is retained; otherwise, it is discarded. In this manner, we discard clauses which are unlikely to be useful. Note that this process is efficient because the score of a clause only needs to be computed once, and can be cached for future comparisons. (Alternatively, we could evaluate a clause against all its sub-clauses taken together, but this would require re-optimizing the weights for each combination of sub-clauses for every comparison, which is computationally expensive.)

Finally we add the retained clauses to an MLN. We have the option of doing this in several ways. We could greedily add the clauses one at a time in order of decreasing score. After adding each clause, we relearn the weights, and keep the clause in the MLN if it improves the overall WPLL. Alternatively, we could add all the clauses to the MLN, and learn weights using L1 regularization to prune away 'bad' clauses by giving them zero weights (Huynh & Mooney, 2008). Lastly, we could use L2-regularization instead if the number of clauses is not too large, and rely on the regularization to give 'bad' clauses low weight. Optionally, we discard clauses containing 'dangling' variables (i.e., variables which only appear once in a clause), since these are unlikely to be useful.

Table 1. Information on datasets.

Dataset	Types	Const- ants	Predi- cates	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

We use two heuristics to speed up clause evaluation. First we discard a path at the outset if it contains fewer than μ true ground atoms per hyperedge. This cuts the time we spend evaluating clauses that are not well supported by data. Second, before evaluating a clause's WPLL with respect to a database, we evaluate it with respect to the smaller number of ground atoms contained in the paths that gave rise to it. (Note that a clause can be created from different paths.) We limit the number of such ground atoms to a maximum of ν. We use a smaller structure prior π' to avoid prematurely removing good clauses.

4. Experiments

4.1. Datasets

We carried out experiments to investigate whether LHL performs better than previous approaches, and to evaluate the contributions of its components. We used three datasets publicly available at http://alchemy.cs.washington.edu. Their details are shown in Table 1.

IMDB. This dataset, created by Mihalkova and Mooney (2007) from the IMDB.com database, describes a movie domain. It contains predicates describing movies, actors, directors, and their relationships (e.g, `Actor(person)`, `WorkedIn(person, movie)`, etc.) It is divided into 5 independent folds. We omitted 4 equality predicates (e.g., `SameMovie(movie, movie)`) that are true if and only if their arguments are the same. They are superseded by the equality operator in the systems we are comparing, and can be easily predicted with a unit clause (e.g., $\texttt{SameMovie}(\texttt{x},\texttt{x})$), trivially boosting the systems' performances.

UW-CSE. This dataset, prepared by Richardson and Domingos (2006), describes an academic department. Its predicates describe students, faculty, and their relationships (e.g, $\texttt{Professor}(\texttt{person})$, $\texttt{TaughtBy}(\texttt{course},\texttt{person},\texttt{quarter})$, etc.). The dataset is divided into 5 independent areas/folds (AI, graphics, etc.). We omitted 9 equality predicates for the same reasons as above.

Cora. This dataset is a collection of citations to computer science papers, created by Andrew McCallum, and later processed by Singla and Domingos (2006) into 5 folds for the task of deduplicating the citations, and their title, author, and venue fields. Predicates include: $\texttt{SameCitation}(\texttt{cit1},\texttt{cit2})$, $\texttt{TitleHasWord}(\texttt{title},\texttt{word})$, etc.

4.2. Systems

We compared LHL to two state-of-the-art systems: BUSL (Mihalkova & Mooney, 2007) and MSL (Kok & Domingos, 2005). Both systems are implemented in the Alchemy software package (Kok et al., 2009).

BUSL. BUSL uses a form of relational pathfinding to find a path of ground atoms in the training data, but restricts itself to very short paths (length 2) to avoid fully searching the large space of paths. It variabilizes each ground atom in the path, and constructs a Markov network whose nodes are the paths viewed as Boolean variables (conjunctions of atoms). It uses the Grow-Shrink Markov network learning algorithm (Bromberg et al., 2006) to find the edges between the nodes. For each node, the algorithm greedily adds and removes nodes from its Markov blanket using the χ^2 measure of dependence. From the 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 then computes the WPLL of the clauses, and adds them one at a time, in order of decreasing WPLL, to an MLN containing only unit clauses. After adding a clause, the weights of all clauses in the MLN are relearned to compute the new WPLL. If a clause increases the overall WPLL, it is retained in the MLN.

MSL. We used the beam search version of MSL that is implemented in Alchemy. MSL begins by adding all possible unit clauses to an MLN. MSL maintains a set of n clauses that give the best score improvement over the current MLN. Initially, the set is empty. MSL creates all possible clauses of length two, and adds the n clauses with the highest improvement in WPLL 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.

To investigate the importance of hypergraph lifting, we removed the LiftGraph component from LHL, and let FindPaths run on the unlifted hypergraph. The rules it learned were pruned by CreateMLN as normal. We call this system LHL-FindPaths. We also investigated the contribution of hypergraph lifting alone by applying LiftGraph's MLN on the test sets. We call this system LHL-LiftGraph. We also investigated the effectiveness of the two heuristics in CreateMLN, by disabling them and observing the performance of the MLN thus learned by LHL. We call this system LHL-NoHeu. Altogether we compared six systems: LHL, LHL-NoHeu, LHL-FindPaths, LHL-LiftGraph, BUSL and MSL. All systems are implemented in C++.

The following parameter values were used for the LHL systems on all datasets: $\lambda = 1$, $\mu = 50$, $\nu = 500$, $\sigma = 0.5$. The other parameters were set as follows: $\omega = 5$ (IMDB, UW-CSE) and 4 (Cora); $\pi = 0.01$ (UW-CSE, Cora) and 0.1 (IMDB); and $\pi' = 0.001$ (UW-CSE, Cora) and 0.01 (IMDB). (See Algorithm 1 for the parameter descriptions.) For BUSL and MSL, we set their parameters corresponding to π to values we used for LHL. We also set their $minWeight$ parameter to zero (empirically we found that this value performed better than their defaults). All other BUSL and MSL parameters were set to their default values. For all LHL systems, we created clauses with all combinations of negated/non-negated atoms in a variabilized path; greedily added clauses one at a time in order of decreasing score to an MLN (initially empty); and excluded clauses with dangling variables from the final MLN. (To ensure fairness, we also tried excluding dangling clauses in BUSL and MSL, and report the best results for each.) The parameters were set in an *ad hoc* manner, and per-fold optimization using a validation set could conceivably yield better results. All systems were run on identically configured machines (2.8GHz, 4GB RAM).

4.3. 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 be-

Table 2. Experimental results.

System	IMDB AUC	IMDB CLL	IMDB Time (min)	UW-CSE AUC	UW-CSE CLL	UW-CSE Time (hr)	Cora AUC	Cora CLL	Cora Time (hr)
LHL	0.69±0.01	−0.13±0.00	15.63±1.88	0.22±0.01	−0.04±0.00	7.55±1.53	0.87±0.00	−0.26±0.00	14.82±1.78
LHL-NoHeu	0.69±0.01	−0.13±0.00	39.00±13.56	0.22±0.01	−0.04±0.00	158.24±46.70	0.87±0.00	−0.26±0.00	33.99±3.86
LHL-FindPaths	0.69±0.01	−0.13±0.00	242.41±30.31	0.19±0.01	−0.04±0.00	56.69±19.98	0.91±0.00	−0.17±0.00	5935.50±39.21
LHL-LiftGraph	0.45±0.01	−0.27±0.01	0.18±0.01	0.14±0.01	−0.06±0.00	0.001±0.000	-	-	0.01±0.01
BUSL	0.47±0.01	−0.14±0.00	4.69±1.02	0.21±0.01	−0.05±0.00	12.97±9.80	0.17±0.00	−0.37±0.00	18.65±9.52
MSL	0.41±0.01	−0.17±0.00	0.17±0.10	0.18±0.01	−0.57±0.00	2.13±0.38	0.17±0.00	−0.37±0.00	65.60±1.82

ing true, using the groundings of all other predicates as evidence. Exceptions are the predicates Actor and Director (IMDB), and Student and Professor (UW-CSE). We evaluated groundings for those predicates together, using all other predicates as evidence. This is because groundings of those predicates for the same constant are mutually exclusive and exhaustive (e.g., Actor(Bob) and Director(Bob)). Knowing one determines the value of the other. 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 used Alchemy's Gibbs sampling for all systems except LHL-LiftGraph. For LHL-LiftGraph, we used Alchemy's MC-SAT algorithm (Poon & Domingos, 2006) because it has been shown to give better results for MLNs containing deterministic rules, which LHL-LiftGraph does. 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). The advantage of the CLL is that it directly measures the quality of the probability estimates produced. The advantage of the AUC is that it is insensitive to the large number of true negatives (i.e., atoms that are false and predicted to be false). The precision-recall curve for a predicate is computed by varying the threshold CLL above which an atom is predicted to be true.

4.4. Results

Table 2 reports the 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 LHL to BUSL and MSL. In both AUC and CLL, LHL outperforms BUSL and MSL on all datasets. The differences between LHL and BUSL/MSL on all datasets are statistically significant according to one-tailed paired t-tests (p-values ≤ 0.02 for both AUC and CLL). LHL is slower than BUSL and MSL on the smallest dataset (IMDB), mixed on the medium one (UW-CSE), and faster on the largest one (Cora). This suggests that LHL scales better than BUSL and MSL.

Next we compare LHL to its components LHL-LiftGraph and LHL-FindPaths. Comparing the runtimes of LHL and LHL-FindPaths, we see that LHL is much faster than LHL-FindPaths. LHL's AUC and CLL are similar to or better than LHL-FindPaths's on IMDB and UW-CSE, but are worse on Cora. These results suggest that: LHL is a lot faster than LHL-FindPaths without any loss in accuracy on some datasets; and when LHL-FindPaths does better, it does so at a huge computational cost (e.g., it took about 247 days to run on Cora[2]). LHL also outperforms LHL-LiftGraph on both AUC and CLL on the IMDB and UW-CSE datasets.[3] This suggests that LHL's ability to learn clauses that capture complex dependencies among predicates is an advantage over the simple rules in LHL-LiftGraphs.

Comparing LHL and LHL-NoHeu, we see that the two speedup heuristics in CreateMLN are effective in reducing LHL's runtime. On all datasets, we see that the heuristics do not compromise the quality of the MLNs that LHL learns because LHL and LHL-NoHeu have the same AUC and CLL. Examining the runtime of LiftGraph, we found that it accounts for only a tiny fraction of LHL runtime (less than 0.1%).

The results for MSL on UW-CSE and Cora are much worse than those reported by Kok and Domingos (2005). They evaluated MSL by computing the probability that a ground atom is true given all other ground atoms as evidence, a much easier task than ours. We also did not use their domain-specific declarative bias to guide clause construction. (Notice how LHL is able to overcome the myopia of greedy search without the help of this bias.) The results for BUSL on IMDB and UW-CSE are also worse than that reported by Mihalkova and Mooney (2007). Unlike them, we omitted the equality predicates (as mentioned earlier) because they are superfluous and can be easily predicted with

[2] For each test fold, we ran FindPaths in parallel on all training folds, and added the runtimes.

[3] LHL-LiftGraph on Cora crashed by running out of memory. Alchemy automatically converts the default atom prediction rules into clausal form and represents each clause separately, causing a blow-up in the number of clauses.

a single unit clause. We also infer the groundings of Actor/Director and Professor/Student simultaneously, which is a harder task than theirs. The last two reasons also contribute to the poor performance of MSL.

5. 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 LHL, which models the full joint distribution of the predicates.

Besides relational pathfinding (Richards & Mooney, 1992), ILP approaches with bottom-up aspects include Muggleton & Buntine (1988), Muggleton & Feng (1992), etc. These approaches are vulnerable to noise in the data, and also only create clauses to predict a single target predicate.

Popescul and Ungar (2004) have also used clustering to improve probabilistic rule induction. Their approach is limited to logistic regression and SQL rules, uses a very simple clustering method (k-means), and requires pre-specifying the number of clusters. Craven and Slattery (2001) learn first-order rules for hypertext classification using naive Bayes models as invented predicates.

The idea of lifting comes from theorem-proving in first-order logic. In recent years, it has been extended to inference in MLNs and other probabilistic languages. In lifted belief propagation (Singla & Domingos, 2008), the algorithm forms clusters of ground atoms and clusters of ground clauses. It performs inference over the more compact network of clusters, thereby improving efficiency. This is analogous to LHL's approach of forming clusters of ground atoms to create a lifted hypergraph in which the search for clauses is more efficient.

6. Conclusion and Future Work

We proposed LHL, a novel algorithm for learning MLN structure. LHL lifts the training data into a compact hypergraph over clusters of constants, and uses relational pathfinding over the hypergraph to find clauses. Empirical comparisons with two state-of-the-art systems on three datasets show the promise of LHL.

Future work includes: more tightly integrating the components of LHL; scaling it up further; applying LHL to larger, richer domains (e.g., the Web); etc.

Acknowledgments: This research was partly funded by ARO grant W911NF-08-1-0242, 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., & Esposito, F. (2008a). Discriminative structure learning of Markov logic networks. *Proc. ILP'08.*

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

Bromberg, F., Margaritis, D., & Honavar, Y. (2006). Efficient Markov network structure discovery using independence tests. *Proc. ICDM'06.*

Craven, M., & Slattery, S. (2001). Relational learning with statistical predicate invention: Better models for hypertext. *Machine Learning, 43.*

Genesereth, M. R., & Nilsson, N. J. (1987). *Logical foundations of artificial intelligence.*

Getoor, L., & Taskar, B. (Eds.). (2007). *Introduction to statistical relational learning.*

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

Kok, S., & Domingos, P. (2005). Learning the structure of Markov logic networks. *Proc. ICML'05.*

Kok, S., & Domingos, P. (2007). Statistical predicate invention. *Proc. ICML'07.*

Kok, S., & Domingos, P. (2008). Extracting semantic networks from text via relational clustering. *Proc. ECML'08.*

Kok, S., Sumner, M., Richardson, M., Singla, P., Poon, H., Lowd, D., Wang, J., & Domingos, P. (2009). *The Alchemy system for statistical relational AI* (Technical Report). Dept. of Comp. Sci. and Eng., Univ. of Washington. http://alchemy.cs.washington.edu.

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

Muggleton, S., & Buntine, W. (1988). Machine invention of first-order predicates by inverting resolution. *Proc. ICML'88.*

Muggleton, S., & Feng, C. (1992). Efficient induction in logic programs. In S. Muggleton (Ed.), *Inductive logic programming.*

Pearl, J. (1988). *Probabilistic reasoning in intelligent systems: Networks of plausible inference.*

Poon, H., & Domingos, P. (2006). Sound and Efficient Inference with Probabilistic and Deterministic Dependencies. *Proc. AAAI'06.*

Popescul, A., & Ungar, L. H. (2004). Cluster-based concept invention for statistical relational learning. *Proc. KDD'04.*

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

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

Singla, P., & Domingos, P. (2006). Entity resolution with Markov logic. *Proc. ICDM'06.*

Singla, P., & Domingos, P. (2008). Lifted first-order belief propagation. *Proc. AAAI'08.*