Markov Logic

Conference Paper · January 2008

DOI: 10.1007/978-3-540-78652-8_4 · Source: DBLP

Pedro Domingos
University of Washington Seattle
215 PUBLICATIONS **23,356** CITATIONS

SEE PROFILE

Stanley Kok
Singapore University of Technology and Design
18 PUBLICATIONS **1,377** CITATIONS

SEE PROFILE

Daniel Lowd
University of Oregon
41 PUBLICATIONS **1,297** CITATIONS

SEE PROFILE

Markov Logic

Pedro Domingos[1], Stanley Kok[1], Daniel Lowd[1], Hoifung Poon[1], Matthew Richardson[2], and Parag Singla[1]

[1] Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195-2350, U.S.A.
{pedrod, koks, lowd, hoifung, parag}@cs.washington.edu
[2] Microsoft Research
Redmond, WA 98052
mattri@microsoft.com

Abstract. Most real-world machine learning problems have both statistical and relational aspects. Thus learners need representations that combine probability and relational logic. Markov logic accomplishes this by attaching weights to first-order formulas and viewing them as templates for features of Markov networks. Inference algorithms for Markov logic draw on ideas from satisfiability, Markov chain Monte Carlo and knowledge-based model construction. Learning algorithms are based on the conjugate gradient algorithm, pseudo-likelihood and inductive logic programming. Markov logic has been successfully applied to problems in entity resolution, link prediction, information extraction and others, and is the basis of the open-source Alchemy system.

1 Introduction

Two key challenges in most machine learning applications are uncertainty and complexity. The standard framework for handling uncertainty is probability; for complexity, it is first-order logic. Thus we would like to be able to learn and perform inference in representation languages that combine the two. This is the focus of the burgeoning field of statistical relational learning [11]. Many approaches have been proposed in recent years, including stochastic logic programs [33], probabilistic relational models [9], Bayesian logic programs [17], relational dependency networks [34], and others. These approaches typically combine probabilistic graphical models with a subset of first-order logic (e.g., Horn clauses), and can be quite complex. Recently, we introduced Markov logic, a language that is conceptually simple, yet provides the full expressiveness of graphical models and first-order logic in finite domains, and remains well-defined in many infinite domains [44, 53]. Markov logic extends first-order logic by attaching weights to formulas. Semantically, weighted formulas are viewed as templates for constructing Markov networks. In the infinite-weight limit, Markov logic reduces to standard first-order logic. Markov logic avoids the assumption of i.i.d. (independent and identically distributed) data made by most statistical learners by leveraging the power of first-order logic to compactly represent dependencies among objects and relations. In this chapter, we describe the Markov logic representation and give an overview of current inference and learning algorithms for it. We begin with some background on Markov networks and first-order logic.

2 Markov Networks

A *Markov network* (also known as *Markov random field*) is a model for the joint distribution of a set of variables $X = (X_1, X_2, \ldots, X_n) \in \mathcal{X}$ [37]. It is composed of an undirected graph G and a set of potential functions ϕ_k. The graph has a node for each variable, and the model has a potential function for each clique in the graph. A potential function is a non-negative real-valued function of the state of the corresponding clique. The joint distribution represented by a Markov network is given by

$$P(X\!=\!x) = \frac{1}{Z} \prod_k \phi_k(x_{\{k\}}) \tag{1}$$

where $x_{\{k\}}$ is the state of the kth clique (i.e., the state of the variables that appear in that clique). Z, known as the *partition function*, is given by $Z = \sum_{x \in \mathcal{X}} \prod_k \phi_k(x_{\{k\}})$. Markov networks are often conveniently represented as *log-linear models*, with each clique potential replaced by an exponentiated weighted sum of features of the state, leading to

$$P(X\!=\!x) = \frac{1}{Z} \exp\left(\sum_j w_j f_j(x) \right) \tag{2}$$

A feature may be any real-valued function of the state. This chapter will focus on binary features, $f_j(x) \in \{0, 1\}$. In the most direct translation from the potential-function form (Equation 1), there is one feature corresponding to each possible state $x_{\{k\}}$ of each clique, with its weight being $\log \phi_k(x_{\{k\}})$. This representation is exponential in the size of the cliques. However, we are free to specify a much smaller number of features (e.g., logical functions of the state of the clique), allowing for a more compact representation than the potential-function form, particularly when large cliques are present. Markov logic will take advantage of this.

Inference in Markov networks is #P-complete [47]. The most widely used method for approximate inference in Markov networks is Markov chain Monte Carlo (MCMC) [12], and in particular Gibbs sampling, which proceeds by sampling each variable in turn given its Markov blanket. (The Markov blanket of a node is the minimal set of nodes that renders it independent of the remaining network; in a Markov network, this is simply the node's neighbors in the graph.) Marginal probabilities are computed by counting over these samples; conditional probabilities are computed by running the Gibbs sampler with the conditioning variables clamped to their given values. Another popular method for inference in Markov networks is belief propagation [59].

Maximum-likelihood or MAP estimates of Markov network weights cannot be computed in closed form but, because the log-likelihood is a concave function of the weights, they can be found efficiently (modulo inference) using standard gradient-based or quasi-Newton optimization methods [35]. Another alternative is iterative scaling [7]. Features can also be learned from data, for example by greedily constructing conjunctions of atomic features [7].

3 First-Order Logic

A *first-order knowledge base (KB)* is a set of sentences or formulas in first-order logic [10]. Formulas are constructed using four types of symbols: constants, variables, functions, and predicates. Constant symbols represent objects in the domain of interest (e.g., people: `Anna`, `Bob`, `Chris`, etc.). Variable symbols range over the objects in the domain. Function symbols (e.g., `MotherOf`) represent mappings from tuples of objects to objects. Predicate symbols represent relations among objects in the domain (e.g., `Friends`) or attributes of objects (e.g., `Smokes`). An *interpretation* specifies which objects, functions and relations in the domain are represented by which symbols. Variables and constants may be *typed*, in which case variables range only over objects of the corresponding type, and constants can only represent objects of the corresponding type. For example, the variable $\mathtt{x}$ might range over people (e.g., Anna, Bob, etc.), and the constant $\mathtt{C}$ might represent a city (e.g, Seattle, Tokyo, etc.).

A *term* is any expression representing an object in the domain. It can be a constant, a variable, or a function applied to a tuple of terms. For example, `Anna`, $\mathtt{x}$, and $\mathtt{GreatestCommonDivisor}(\mathtt{x},\mathtt{y})$ are terms. An *atomic formula* or *atom* is a predicate symbol applied to a tuple of terms (e.g., $\mathtt{Friends}(\mathtt{x},\mathtt{MotherOf}(\mathtt{Anna}))$). Formulas are recursively constructed from atomic formulas using logical connectives and quantifiers. If F_1 and F_2 are formulas, the following are also formulas: $\neg F_1$ (negation), which is true iff F_1 is false; $F_1 \wedge F_2$ (conjunction), which is true iff both F_1 and F_2 are true; $F_1 \vee F_2$ (disjunction), which is true iff F_1 or F_2 is true; $F_1 \Rightarrow F_2$ (implication), which is true iff F_1 is false or F_2 is true; $F_1 \Leftrightarrow F_2$ (equivalence), which is true iff F_1 and F_2 have the same truth value; $\forall\mathtt{x}\ F_1$ (universal quantification), which is true iff F_1 is true for every object $\mathtt{x}$ in the domain; and $\exists\mathtt{x}\ F_1$ (existential quantification), which is true iff F_1 is true for at least one object $\mathtt{x}$ in the domain. Parentheses may be used to enforce precedence. A *positive literal* is an atomic formula; a *negative literal* is a negated atomic formula. The formulas in a KB are implicitly conjoined, and thus a KB can be viewed as a single large formula. A *ground term* is a term containing no variables. A *ground atom* or *ground predicate* is an atomic formula all of whose arguments are ground terms. A *possible world* (along with an interpretation) assigns a truth value to each possible ground atom.

A formula is *satisfiable* iff there exists at least one world in which it is true. The basic inference problem in first-order logic is to determine whether a knowledge base KB *entails* a formula F, i.e., if F is true in all worlds where KB is true (denoted by $KB \models F$). This is often done by *refutation*: KB entails F iff

$KB \cup \neg F$ is unsatisfiable. (Thus, if a KB contains a contradiction, all formulas trivially follow from it, which makes painstaking knowledge engineering a necessity.) For automated inference, it is often convenient to convert formulas to a more regular form, typically *clausal form* (also known as *conjunctive normal form (CNF)*). A KB in clausal form is a conjunction of *clauses*, a clause being a disjunction of literals. Every KB in first-order logic can be converted to clausal form using a mechanical sequence of steps.[3] Clausal form is used in resolution, a sound and refutation-complete inference procedure for first-order logic [46].

Inference in first-order logic is only semidecidable. Because of this, knowledge bases are often constructed using a restricted subset of first-order logic with more desirable properties. The most widely-used restriction is to *Horn clauses*, which are clauses containing at most one positive literal. The Prolog programming language is based on Horn clause logic [25]. Prolog programs can be learned from databases by searching for Horn clauses that (approximately) hold in the data; this is studied in the field of inductive logic programming (ILP) [22].

Table 1 shows a simple KB and its conversion to clausal form. Notice that, while these formulas may be *typically* true in the real world, they are not *always* true. In most domains it is very difficult to come up with non-trivial formulas that are always true, and such formulas capture only a fraction of the relevant knowledge. Thus, despite its expressiveness, pure first-order logic has limited applicability to practical AI problems. Many *ad hoc* extensions to address this have been proposed. In the more limited case of propositional logic, the problem is well solved by probabilistic graphical models. The next section describes a way to generalize these models to the first-order case.

Table 1. Example of a first-order knowledge base and MLN. $\mathtt{Fr}()$ is short for $\mathtt{Friends}()$, $\mathtt{Sm}()$ for $\mathtt{Smokes}()$, and $\mathtt{Ca}()$ for $\mathtt{Cancer}()$.

First-Order Logic	Clausal Form	Weight
"Friends of friends are friends."		
$\forall \mathtt{x} \forall \mathtt{y} \forall \mathtt{z}\ \mathtt{Fr}(\mathtt{x},\mathtt{y}) \land \mathtt{Fr}(\mathtt{y},\mathtt{z}) \Rightarrow \mathtt{Fr}(\mathtt{x},\mathtt{z})$	$\neg\mathtt{Fr}(\mathtt{x},\mathtt{y}) \lor \neg\mathtt{Fr}(\mathtt{y},\mathtt{z}) \lor \mathtt{Fr}(\mathtt{x},\mathtt{z})$	0.7
"Friendless people smoke."		
$\forall \mathtt{x}\ (\neg(\exists \mathtt{y}\ \mathtt{Fr}(\mathtt{x},\mathtt{y})) \Rightarrow \mathtt{Sm}(\mathtt{x}))$	$\mathtt{Fr}(\mathtt{x},\mathtt{g}(\mathtt{x})) \lor \mathtt{Sm}(\mathtt{x})$	2.3
"Smoking causes cancer."		
$\forall \mathtt{x}\ \mathtt{Sm}(\mathtt{x}) \Rightarrow \mathtt{Ca}(\mathtt{x})$	$\neg\mathtt{Sm}(\mathtt{x}) \lor \mathtt{Ca}(\mathtt{x})$	1.5
"If two people are friends, then either		
both smoke or neither does."	$\neg\mathtt{Fr}(\mathtt{x},\mathtt{y}) \lor \mathtt{Sm}(\mathtt{x}) \lor \neg\mathtt{Sm}(\mathtt{y})$,	1.1
$\forall \mathtt{x} \forall \mathtt{y}\ \mathtt{Fr}(\mathtt{x},\mathtt{y}) \Rightarrow (\mathtt{Sm}(\mathtt{x}) \Leftrightarrow \mathtt{Sm}(\mathtt{y}))$	$\neg\mathtt{Fr}(\mathtt{x},\mathtt{y}) \lor \neg\mathtt{Sm}(\mathtt{x}) \lor \mathtt{Sm}(\mathtt{y})$	1.1

[3] This conversion includes the removal of existential quantifiers by Skolemization, which is not sound in general. However, in finite domains an existentially quantified formula can simply be replaced by a disjunction of its groundings.

4 Markov Logic

A first-order KB can be seen as a set of hard constraints on the set of possible worlds: if a world violates even one formula, it has zero probability. The basic idea in Markov logic is to soften these constraints: when a world violates one formula in the KB it is less probable, but not impossible. The fewer formulas a world violates, the more probable it is. Each formula has an associated weight (e.g., see Table 1) that reflects how strong a constraint it is: the higher the weight, the greater the difference in log probability between a world that satisfies the formula and one that does not, other things being equal.

Definition 1. [44] *A Markov logic network (MLN) L is a set of pairs (F_i, w_i), where F_i is a formula in first-order logic and w_i is a real number. Together with a finite set of constants $C = \{c_1, c_2, \ldots, c_{|C|}\}$, it defines a Markov network $M_{L,C}$ (Equations 1 and 2) as follows:*

1. *$M_{L,C}$ contains one binary node for each possible grounding of each atom appearing in L. The value of the node is 1 if the ground atom is true, and 0 otherwise.*
2. *$M_{L,C}$ contains one feature for each possible grounding of each formula F_i in L. The value of this feature is 1 if the ground formula is true, and 0 otherwise. The weight of the feature is the w_i associated with F_i in L.*

Thus there is an edge between two nodes of $M_{L,C}$ iff the corresponding ground atoms appear together in at least one grounding of one formula in L. For example, an MLN containing the formulas $\forall \mathtt{x}\ \mathtt{Smokes(x)} \Rightarrow \mathtt{Cancer(x)}$ (smoking causes cancer) and $\forall \mathtt{x} \forall \mathtt{y}\ \mathtt{Friends(x,y)} \Rightarrow (\mathtt{Smokes(x)} \Leftrightarrow \mathtt{Smokes(y)})$ (friends have similar smoking habits) applied to the constants `Anna` and `Bob` (or `A` and `B` for short) yields the ground Markov network in Figure 1. Its features include $\mathtt{Smokes(Anna)} \Rightarrow \mathtt{Cancer(Anna)}$, etc. Notice that, although the two formulas above are false as universally quantified logical statements, as weighted features of an MLN they capture valid statistical regularities, and in fact represent a standard social network model [55].

An MLN can be viewed as a *template* for constructing Markov networks. From Definition 1 and Equations 1 and 2, the probability distribution over possible worlds x specified by the ground Markov network $M_{L,C}$ is given by

$$P(X=x) = \frac{1}{Z} \exp\left(\sum_{i=1}^{F} w_i n_i(x)\right) \tag{3}$$

where F is the number of formulas in the MLN and $n_i(x)$ is the number of true groundings of F_i in x. As formula weights increase, an MLN increasingly resembles a purely logical KB, becoming equivalent to one in the limit of all infinite weights. When the weights are positive and finite, and all formulas are simultaneously satisfiable, the satisfying solutions are the modes of the distribution represented by the ground Markov network. Most importantly, Markov

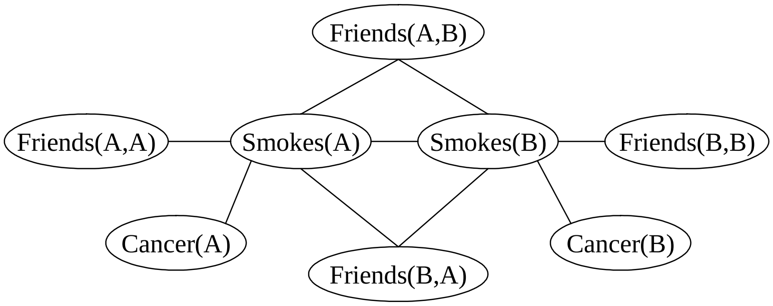

Fig. 1. Ground Markov network obtained by applying an MLN containing the formulas $\forall \mathtt{x}\, \mathtt{Smokes(x)} \Rightarrow \mathtt{Cancer(x)}$ and $\forall \mathtt{x} \forall \mathtt{y}\, \mathtt{Friends(x,y)} \Rightarrow (\mathtt{Smokes(x)} \Leftrightarrow \mathtt{Smokes(y)})$ to the constants $\mathtt{Anna(A)}$ and $\mathtt{Bob(B)}$.

logic allows contradictions between formulas, which it resolves simply by weighing the evidence on both sides. This makes it well suited for merging multiple KBs. Markov logic also provides a natural and powerful approach to the problem of merging knowledge and data in different representations that do not align perfectly, as will be illustrated in the application section.

It is interesting to see a simple example of how Markov logic generalizes first-order logic. Consider an MLN containing the single formula $\forall \mathtt{x}\, \mathtt{R(x)} \Rightarrow \mathtt{S(x)}$ with weight w, and $C = \{\mathtt{A}\}$. This leads to four possible worlds: $\{\neg\mathtt{R(A)}, \neg\mathtt{S(A)}\}$, $\{\neg\mathtt{R(A)}, \mathtt{S(A)}\}$, $\{\mathtt{R(A)}, \neg\mathtt{S(A)}\}$, and $\{\mathtt{R(A)}, \mathtt{S(A)}\}$. From Equation 3 we obtain that $P(\{\mathtt{R(A)}, \neg\mathtt{S(A)}\}) = 1/(3e^w + 1)$ and the probability of each of the other three worlds is $e^w/(3e^w + 1)$. (The denominator is the partition function Z; see Section 2.) Thus, if $w > 0$, the effect of the MLN is to make the world that is inconsistent with $\forall \mathtt{x}\; \mathtt{R(x)} \Rightarrow \mathtt{S(x)}$ less likely than the other three. From the probabilities above we obtain that $P(\mathtt{S(A)}|\mathtt{R(A)}) = 1/(1 + e^{-w})$. When $w \to \infty$, $P(\mathtt{S(A)}|\mathtt{R(A)}) \to 1$, recovering the logical entailment.

It is easily seen that all discrete probabilistic models expressible as products of potentials, including Markov networks and Bayesian networks, are expressible in Markov logic. In particular, many of the models frequently used in AI can be stated quite concisely as MLNs, and combined and extended simply by adding the corresponding formulas. Most significantly, Markov logic facilitates the construction of non-i.i.d. models (i.e., models where objects are not independent and identically distributed).

When working with Markov logic, we typically make three assumptions about the logical representation: different constants refer to different objects (unique names), the only objects in the domain are those representable using the constant and function symbols (domain closure), and the value of each function for each tuple of arguments is always a known constant (known functions). These assumptions ensure that the number of possible worlds is finite and that the Markov logic network will give a well-defined probability distribution. These assumptions are quite reasonable in most practical applications, and greatly simplify the use of MLNs. We will make these assumptions for the remainder of the chapter. See Richardson and Domingos [44] for further details on the Markov logic representation.

Markov logic can also be applied to a number of interesting infinite domains where some of these assumptions do not hold. See Singla and Domingos [53] for details on Markov logic in infinite domains.

5 Inference

5.1 MAP/MPE Inference

In the remainder of this chapter, we assume that the MLN is in function-free clausal form for convenience, but these methods can be applied to other MLNs as well. A basic inference task is finding the most probable state of the world given some evidence. (This is known as MAP inference in the Markov network literature, and MPE inference in the Bayesian network literature.) Because of the form of Equation 3, in Markov logic this reduces to finding the truth assignment that maximizes the sum of weights of satisfied clauses. This can be done using any weighted satisfiability solver, and (remarkably) need not be more expensive than standard logical inference by model checking. (In fact, it can be faster, if some hard constraints are softened.) We have successfully used MaxWalkSAT, a weighted variant of the WalkSAT local-search satisfiability solver, which can solve hard problems with hundreds of thousands of variables in minutes [16]. MaxWalkSAT performs this stochastic search by picking an unsatisfied clause at random and flipping the truth value of one of the atoms in it. With a certain probability, the atom is chosen randomly; otherwise, the atom is chosen to maximize the sum of satisfied clause weights when flipped. This combination of random and greedy steps allows MaxWalkSAT to avoid getting stuck in local optima while searching. Pseudocode for MaxWalkSAT is shown in Algorithm 1. DeltaCost(v) computes the change in the sum of weights of unsatisfied clauses that results from flipping variable v in the current solution. Uniform(0,1) returns a uniform deviate from the interval $[0, 1]$.

One problem with this approach is that it requires propositionalizing the domain (i.e., grounding all atoms and clauses in all possible ways), which consumes memory exponential in the arity of the clauses. We have overcome this by developing LazySAT, a lazy version of MaxWalkSAT which grounds atoms and clauses only as needed [52]. This takes advantage of the sparseness of relational domains, where most atoms are false and most clauses are trivially satisfied. For example, in the domain of scientific research, most groundings of the atom $\texttt{Author}(\texttt{person}, \texttt{paper})$ are false, and most groundings of the clause $\texttt{Author}(\texttt{person1}, \texttt{paper}) \wedge \texttt{Author}(\texttt{person2}, \texttt{paper}) \Rightarrow \texttt{Coauthor}(\texttt{person1}, \texttt{person2})$ are satisfied. In LazySAT, the memory cost does not scale with the number of possible clause groundings, but only with the number of groundings that are potentially unsatisfied at some point in the search.

Algorithm 2 gives pseudo-code for LazySAT, highlighting the places where it differs from MaxWalkSAT. LazySAT maintains a set of *active atoms* and a

Algorithm 1 **MaxWalkSAT**(*weighted_clauses*, *max_flips*, *max_tries*, *target*, *p*)

```
$vars \leftarrow$ variables in $weighted\_clauses$
for $i \leftarrow 1$ to $max\_tries$ do
  $soln \leftarrow$ a random truth assignment to $vars$
  $cost \leftarrow$ sum of weights of unsatisfied clauses in $soln$
  for $i \leftarrow 1$ to $max\_flips$ do
    if $cost \leq target$ then
      return "Success, solution is", $soln$
    end if
    $c \leftarrow$ a randomly chosen unsatisfied clause
    if Uniform(0,1) $< p$ then
      $v_f \leftarrow$ a randomly chosen variable from $c$
    else
      for each variable $v$ in $c$ do
        compute DeltaCost($v$)
      end for
      $v_f \leftarrow v$ with lowest DeltaCost($v$)
    end if
    $soln \leftarrow soln$ with $v_f$ flipped
    $cost \leftarrow cost$ + DeltaCost($v_f$)
  end for
end for
return "Failure, best assignment is", best $soln$ found
```

set of *active clauses*. A clause is active if it can be made unsatisfied by flipping zero or more of its active atoms. (Thus, by definition, an unsatisfied clause is always active.) An atom is active if it is in the initial set of active atoms, or if it was flipped at some point in the search. The initial active atoms are all those appearing in clauses that are unsatisfied if only the atoms in the database are true, and all others are false. The unsatisfied clauses are obtained by simply going through each possible grounding of all the first-order clauses and materializing the groundings that are unsatisfied; search is pruned as soon as the partial grounding of a clause is satisfied. Given the initial active atoms, the definition of active clause requires that some clauses become active, and these are found using a similar process (with the difference that, instead of checking whether a ground clause is unsatisfied, we check whether it should be active). Each run of LazySAT is initialized by assigning random truth values to the active atoms. This differs from MaxWalkSAT, which assigns random values to all atoms. However, the LazySAT initialization is a valid MaxWalkSAT initialization, and we have verified experimentally that the two give very similar results. Given the same initialization, the two algorithms will produce exactly the same results.

At each step in the search, the variable that is flipped is activated, as are any clauses that by definition should become active as a result. When evaluating the effect on cost of flipping a variable v, if v is active then all of the relevant clauses are already active, and DeltaCost(v) can be computed as in MaxWalkSAT. If v is inactive, DeltaCost(v) needs to be computed using the knowledge base. This is

Algorithm 2 **LazySAT**(*weighted_KB*, *DB*, *max_flips*, *max_tries*, *target*, *p*)

```
for $i \leftarrow 1$ to $max\_tries$ do
    $active\_atoms \leftarrow$ atoms in clauses not satisfied by $DB$
    $active\_clauses \leftarrow$ clauses activated by $active\_atoms$
    $soln \leftarrow$ a random truth assignment to $active\_atoms$
    $cost \leftarrow$ sum of weights of unsatisfied clauses in $soln$
    for $i \leftarrow 1$ to $max\_flips$ do
        if $cost \leq target$ then
            return "Success, solution is", $soln$
        end if
        $c \leftarrow$ a randomly chosen unsatisfied clause
        if Uniform(0,1) $< p$ then
            $v_f \leftarrow$ a randomly chosen variable from $c$
        else
            for each variable $v$ in $c$ do
                compute DeltaCost($v$), using $weighted\_KB$ if $v \notin active\_atoms$
            end for
            $v_f \leftarrow v$ with lowest DeltaCost($v$)
        end if
        if $v_f \notin active\_atoms$ then
            add $v_f$ to $active\_atoms$
            add clauses activated by $v_f$ to $active\_clauses$
        end if
        $soln \leftarrow soln$ with $v_f$ flipped
        $cost \leftarrow cost$ + DeltaCost($v_f$)
    end for
end for
return "Failure, best assignment is", best $soln$ found
```

done by retrieving from the KB all first-order clauses containing the atom that v is a grounding of, and grounding each such clause with the constants in v and all possible groundings of the remaining variables. As before, we prune search as soon as a partial grounding is satisfied, and add the appropriate multiple of the clause weight to DeltaCost(v). (A similar process is used to activate clauses.) While this process is costlier than using pre-grounded clauses, it is amortized over many tests of active variables. In typical satisfiability problems, a small core of "problem" clauses is repeatedly tested, and when this is the case LazySAT will be quite efficient.

At each step, LazySAT flips the same variable that MaxWalkSAT would, and hence the result of the search is the same. The memory cost of LazySAT is on the order of the maximum number of clauses active at the end of a run of flips. (The memory required to store the active atoms is dominated by the memory required to store the active clauses, since each active atom appears in at least one active clause.)

Experiments on entity resolution and planning problems show that this can yield very large memory reductions, and these reductions increase with domain size [52]. For domains whose full instantiations fit in memory, running time is comparable; as problems become larger, full instantiation for MaxWalkSAT becomes impossible.

5.2 Marginal and Conditional Probabilities

Another key inference task is computing the probability that a formula holds, given an MLN and set of constants, and possibly other formulas as evidence. By definition, the probability of a formula is the sum of the probabilities of the worlds where it holds, and computing it by brute force requires time exponential in the number of possible ground atoms. An approximate but more efficient alternative is to use Markov chain Monte Carlo (MCMC) inference [12], which samples a sequence of states according to their probabilities, and counting the fraction of sampled states where the formula holds. This can be extended to conditioning on other formulas by rejecting any state that violates one of them.

For the remainder of the chapter, we focus on the typical case where the evidence is a conjunction of ground atoms. In this scenario, further efficiency can be gained by applying a generalization of knowledge-based model construction [57]. This constructs only the minimal subset of the ground network required to answer the query, and runs MCMC (or any other probabilistic inference method) on it. The network is constructed by checking if the atoms that the query formula directly depends on are in the evidence. If they are, the construction is complete. Those that are not are added to the network, and we in turn check the atoms they depend on. This process is repeated until all relevant atoms have been retrieved. While in the worst case it yields no savings, in practice it can vastly reduce the time and memory required for inference. See Richardson and Domingos [44] for details.

One problem with applying MCMC to MLNs is that it breaks down in the presence of deterministic or near-deterministic dependencies (as do other probabilistic inference methods, e.g., belief propagation [59]). Deterministic dependencies break up the space of possible worlds into regions that are not reachable from each other, violating a basic requirement of MCMC. Near-deterministic dependencies greatly slow down inference, by creating regions of low probability that are very difficult to traverse. Running multiple chains with random starting points does not solve this problem, because it does not guarantee that different regions will be sampled with frequency proportional to their probability, and there may be a very large number of regions.

We have successfully addressed this problem by combining MCMC with satisfiability testing in the MC-SAT algorithm [40]. MC-SAT is a *slice sampling* MCMC algorithm. It uses a combination of satisfiability testing and simulated annealing to sample from the slice. The advantage of using a satisfiability solver (WalkSAT) is that it efficiently finds isolated modes in the distribution, and as a result the Markov chain mixes very rapidly. The slice sampling scheme ensures that detailed balance is (approximately) preserved.

Algorithm 3 MC-SAT(*clauses, weights, num_samples*)

```
$x^{(0)} \leftarrow$ Satisfy(hard clauses)
for $i \leftarrow 1$ to num_samples do
  $M \leftarrow \emptyset$
  for all $c_k \in$ clauses satisfied by $x^{(i-1)}$ do
    With probability $1 - e^{-w_k}$ add $c_k$ to $M$
  end for
  Sample $x^{(i)} \sim \mathcal{U}_{SAT(M)}$
end for
```

MC-SAT is orders of magnitude faster than standard MCMC methods such as Gibbs sampling and simulated tempering, and is applicable to any model that can be expressed in Markov logic, including many standard models in statistical physics, vision, natural language processing, social network analysis, spatial statistics, etc.

Slice sampling [5] is an instance of a widely used approach in MCMC inference that introduces *auxiliary variables* to capture the dependencies between observed variables. For example, to sample from $P(X = x) = (1/Z) \prod_k \phi_k(x_{\{k\}})$, we can define $P(X = x, U = u) = (1/Z) \prod_k I_{[0,\phi_k(x_{\{k\}})]}(u_k)$, where ϕ_k is the kth potential function, u_k is the kth auxiliary variable, $I_{[a,b]}(u_k) = 1$ if $a \leq u_k \leq b$, and $I_{[a,b]}(u_k) = 0$ otherwise. The marginal distribution of X under this joint is $P(X = x)$, so to sample from the original distribution it suffices to sample from $P(x, u)$ and ignore the u values. $P(u_k|x)$ is uniform in $[0, \phi_k(x_{\{k\}})]$, and thus easy to sample from. The main challenge is to sample x given u, which is uniform among all $\mathcal{X}$ that satisfies $\phi_k(x_{\{k\}}) \geq u_k$ for all k. MC-SAT uses SampleSAT [56] to do this. In each sampling step, MC-SAT takes the set of all ground clauses satisfied by the current state of the world and constructs a subset, M, that must be satisfied by the next sampled state of the world. (For the moment we will assume that all clauses have positive weight.) Specifically, a satisfied ground clause is included in M with probability $1 - e^{-w}$, where w is the clause's weight. We then take as the next state a uniform sample from the set of states $SAT(M)$ that satisfy M. (Notice that $SAT(M)$ is never empty, because it always contains at least the current state.) Algorithm 3 gives pseudo-code for MC-SAT. $\mathcal{U}_S$ is the uniform distribution over set S. At each step, all hard clauses are selected with probability 1, and thus all sampled states satisfy them. Negative weights are handled by noting that a clause with weight $w < 0$ is equivalent to its negation with weight $-w$, and a clause's negation is the conjunction of the negations of all of its literals. Thus, instead of checking whether the clause is satisfied, we check whether its negation is satisfied; if it is, with probability $1 - e^{w}$ we select all of its negated literals, and with probability e^{w} we select none.

It can be shown that MC-SAT satisfies the MCMC criteria of detailed balance and ergodicity [40], assuming a perfect uniform sampler. In general, uniform sampling is #P-hard and SampleSAT [56] only yields approximately uniform samples. However, experiments show that MC-SAT is still able to produce very accurate probability estimates, and its performance is not very sensitive to the parameter setting of SampleSAT.

We have applied the ideas of LazySAT to implement a lazy version of MC-SAT that avoids grounding unnecessary atoms and clauses. A working version of this algorithm is present in the open-source Alchemy system [20].

It is also possible to carry out lifted first-order probabilistic inference (akin to resolution) in Markov logic [3]. These methods speed up inference by reasoning at the first-order level about groups of indistinguishable objects rather than propositionalizing the entire domain. This is particularly applicable when the population size is given but little is known about most individual members.

6 Learning

6.1 Generative Weight Learning

MLN weights can be learned generatively by maximizing the likelihood of a relational database (Equation 3). This relational database consists of one or more "possible worlds" that form our training examples. Note that we can learn to generalize from even a single example because the clause weights are shared across their many respective groundings. We assume that the set of constants of each type is known. We also make a closed-world assumption: all ground atoms not in the database are false. This assumption can be removed by using an EM algorithm to learn from the resulting incomplete data. The gradient of the log-likelihood with respect to the weights is

$$\frac{\partial}{\partial w_i} \log P_w(X=x) = n_i(x) - \sum_{x'} P_w(X=x')\, n_i(x') \tag{4}$$

where the sum is over all possible databases x', and $P_w(X = x')$ is $P(X = x')$ computed using the current weight vector $w = (w_1, \ldots, w_i, \ldots)$. In other words, the ith component of the gradient is simply the difference between the number of true groundings of the ith formula in the data and its expectation according to the current model. Unfortunately, computing these expectations requires inference over the model, which can be very expensive. Most fast numeric optimization methods (e.g., conjugate gradient with line search, L-BFGS) also require computing the likelihood itself and hence the partition function Z, which is also intractable. Although inference can be done approximately using MCMC, we have found this to be too slow. Instead, we maximize the pseudo-likelihood of the data, a widely-used alternative [2]. If x is a possible world (relational database) and x_l is the lth ground atom's truth value, the pseudo-log-likelihood of x given weights w is

$$\log P^*_w(X=x) = \sum_{l=1}^{n} \log P_w(X_l = x_l | MB_x(X_l)) \tag{5}$$

where $MB_x(X_l)$ is the state of X_l's Markov blanket in the data (i.e., the truth values of the ground atoms it appears in some ground formula with). Computing the pseudo-likelihood and its gradient does not require inference, and is therefore much faster. Combined with the L-BFGS optimizer [24], pseudo-likelihood yields efficient learning of MLN weights even in domains with millions of ground atoms [44]. However, the pseudo-likelihood parameters may lead to poor results when long chains of inference are required.

In order to reduce overfitting, we penalize each weight with a Gaussian prior. We apply this strategy not only to generative learning, but to all of our weight learning methods, even those embedded within structure learning.

6.2 Discriminative Weight Learning

Discriminative learning is an attractive alternative to pseudo-likelihood. In many applications, we know *a priori* which atoms will be evidence and which ones will be queried, and the goal is to correctly predict the latter given the former. If we partition the ground atoms in the domain into a set of evidence atoms X and a set of query atoms Y, the *conditional likelihood (CLL)* of Y given X is $P(y|x) = (1/Z_x) \exp\left(\sum_{i \in F_Y} w_i n_i(x, y)\right) = (1/Z_x) \exp\left(\sum_{j \in G_Y} w_j g_j(x, y)\right)$, where F_Y is the set of all MLN clauses with at least one grounding involving a query atom, $n_i(x, y)$ is the number of true groundings of the ith clause involving query atoms, G_Y is the set of ground clauses in $M_{L,C}$ involving query atoms, and $g_j(x, y) = 1$ if the jth ground clause is true in the data and 0 otherwise. The gradient of the CLL is

$$\begin{aligned}\frac{\partial}{\partial w_i} \log P_w(y|x) &= n_i(x, y) - \sum_{y'} P_w(y'|x) n_i(x, y') \\ &= n_i(x, y) - E_w[n_i(x, y)] \end{aligned} \qquad (6)$$

As before, computing the expected counts $E_w[n_i(x, y)]$ is intractable. However, they can be approximated by the counts $n_i(x, y_w^*)$ in the MAP state $y_w^*(x)$ (i.e., the most probable state of y given x). This will be a good approximation if most of the probability mass of $P_w(y|x)$ is concentrated around $y_w^*(x)$. Computing the gradient of the CLL now requires only MAP inference to find $y_w^*(x)$, which is much faster than the full conditional inference for $E_w[n_i(x, y)]$. This is the essence of the voted perceptron algorithm, initially proposed by Collins [4] for discriminatively learning hidden Markov models. Because HMMs have a very simple linear structure, their MAP states can be found in polynomial time using the Viterbi algorithm, a form of dynamic programming [43]. The voted perceptron initializes all weights to zero, performs T iterations of gradient ascent using the approximation above, and returns the parameters averaged over all iterations, $w_i = \sum_{t=1}^{T} w_{i,t}/T$. The parameter averaging helps to combat overfitting. T is chosen using a validation subset of the training data. We have extended the voted perceptron to Markov logic simply by replacing Viterbi with MaxWalkSAT to find the MAP state [50].

In practice, the voted perceptron algorithm can exhibit extremely slow convergence when applied to MLNs. One cause of this is that the gradient can easily vary by several orders of magnitude among the different clauses. For example, consider a transitivity rule such as $\texttt{Friends}(\texttt{x},\texttt{y}) \wedge \texttt{Friends}(\texttt{y},\texttt{z}) \Rightarrow \texttt{Friends}(\texttt{x},\texttt{z})$ compared to a simple attribute relationship such as $\texttt{Smokes}(\texttt{x}) \Rightarrow \texttt{Cancer}(\texttt{x})$. In a social network domain of 1000 people, the former clause has one billion groundings while the latter has only 1000. Since each dimension of the gradient is a difference of clause counts and these can vary by orders of magnitude from one clause to another, a learning rate that is small enough to avoid divergence in some weights is too small for fast convergence in others.

This is an instance of the well-known problem of ill-conditioning in numerical optimization, and many candidate solutions for it exist [35]. However, the most common ones are not easily applicable to MLNs because of the nature of the function being optimized. As in Markov networks, computing the likelihood in MLNs requires computing the partition function, which is generally intractable. This makes it difficult to apply methods that require performing line searches, which involve computing the function as well as its gradient. These include most conjugate gradient and quasi-Newton methods (e.g., L-BFGS). Two exceptions to this are scaled conjugate gradient [32] and Newton's method with a diagonalized Hessian [1]. In the remainder of this subsection, we focus on scaled conjugate gradient, since we found it to be the best-performing method for discriminative weight learning.

In many optimization problems, gradient descent can be sped up by performing a line search to find the optimum along the chosen descent direction instead of taking a small step of constant size at each iteration. However, on ill-conditioned problems this is still inefficient, because line searches along successive directions tend to partly undo the effect of each other: each line search makes the gradient along its direction zero, but the next line search will generally make it non-zero again. In long narrow valleys, instead of moving quickly to the optimum, gradient descent zigzags.

A solution to this is to impose at each step the condition that the gradient along previous directions remain zero. The directions chosen in this way are called *conjugate*, and the method *conjugate gradient* [49]. Conjugate gradient methods are some of the most efficient available, on a par with quasi-Newton ones. While the standard conjugate gradient algorithm uses line searches to choose step sizes, we can use the Hessian (matrix of second derivatives of the function) instead. This method is known as *scaled conjugate gradient* (SCG), and was originally proposed by Møller [32] for training neural networks.

In a Markov logic network, the Hessian is simply the negative covariance matrix of the clause counts:

$$\frac{\partial}{\partial w_i \partial w_j} \log P(Y\!=\!y|X\!=\!x) = E_w[n_i]E_w[n_j] - E_w[n_i n_j]$$

Both the gradient and the Hessian matrix can be estimated using samples collected with the MC-SAT algorithm, described earlier. While full convergence could require many samples, we find that as few as five samples are often sufficient for estimating the gradient and Hessian. This is due in part to the efficiency of MC-SAT as a sampler, and in part to the tied weights: the many groundings of each clause can act to reduce the variance.

Given a conjugate gradient search direction $\mathbf{d}$ and Hessian matrix $\mathbf{H}$, we compute the step size α as follows:

$$\alpha = \frac{\mathbf{d}^T\mathbf{g}}{\mathbf{d}^T\mathbf{H}\mathbf{d} + \lambda\mathbf{d}^T\mathbf{d}}$$

For a quadratic function and $\lambda = 0$, this step size would move to the minimum function value along $\mathbf{d}$. Since our function is not quadratic, a non-zero λ term serves to limit the size of the step to a region in which our quadratic approximation is good. After each step, we adjust λ to increase or decrease the size of the so-called *model trust region* based on how well the approximation matched the function. We cannot evaluate the function directly, but the dot product of the step we just took and the gradient after taking it is a lower bound on the improvement in the actual log-likelihood. This works because the log-likelihood of an MLN is convex.

In models with thousands of weights or more, storing the entire Hessian matrix becomes impractical. However, when the Hessian appears only inside a quadratic form, as above, the value of this form can be computed simply as:

$$\mathbf{d}^T\mathbf{H}\mathbf{d} = (E_w[\textstyle\sum_i d_i n_i])^2 - E_w[(\textstyle\sum_i d_i n_i)^2]$$

The product of the Hessian by a vector can also be computed compactly [38].

Conjugate gradient is usually more effective with a preconditioner, a linear transformation that attempts to reduce the condition number of the problem (e.g., [48]). Good preconditioners approximate the inverse Hessian. We use the inverse diagonal Hessian as our preconditioner. Performance with the preconditioner is much better than without.

See Lowd and Domingos [26] for more details and results.

6.3 Structure Learning

The structure of a Markov logic network is the set of formulas or clauses to which we attach weights. In principle, this structure can be learned or revised using any inductive logic programming (ILP) technique. However, since an MLN represents a probability distribution, much better results are obtained by using an evaluation function based on pseudo-likelihood, rather than typical ILP ones like accuracy and coverage [18]. Log-likelihood or conditional log-likelihood are potentially better evaluation functions, but are vastly more expensive to compute. In experiments on two real-world datasets, our MLN structure learning algorithm found better MLN rules than CLAUDIEN [6], FOIL [42], Aleph [54], and even a hand-written knowledge base.

MLN structure learning can start from an empty network or from an existing KB. Either way, we have found it useful to start by adding all unit clauses (single atoms) to the MLN. The weights of these capture (roughly speaking) the marginal distributions of the atoms, allowing the longer clauses to focus on modeling atom dependencies. To extend this initial model, we either repeatedly find the best clause using beam search and add it to the MLN, or add all "good" clauses of length l before trying clauses of length $l+1$. Candidate clauses are formed by adding each predicate (negated or otherwise) to each current clause, with all possible combinations of variables, subject to the constraint that at least one variable in the new predicate must appear in the current clause. Hand-coded clauses are also modified by removing predicates.

We now discuss the evaluation measure, clause construction operators, search strategy, and speedup methods in greater detail.

As an evaluation measure, pseudo-likelihood (Equation 5) tends to give undue weight to the largest-arity predicates, resulting in poor modeling of the rest. We thus define the weighted pseudo-log-likelihood (WPLL) as

$$\log P_w^{\bullet}(X\!=\!x) = \sum_{r\in R} c_r \sum_{k=1}^{g_r} \log P_w(X_{r,k}\!=\!x_{r,k}|MB_x(X_{r,k})) \tag{7}$$

where R is the set of first-order atoms, g_r is the number of groundings of first-order atom r, and $x_{r,k}$ is the truth value (0 or 1) of the kth grounding of r. The choice of atom weights c_r depends on the user's goals. In our experiments, we simply set $c_r = 1/g_r$, which has the effect of weighting all first-order predicates equally. If modeling a predicate is not important (e.g., because it will always be part of the evidence), we set its weight to zero. To combat overfitting, we penalize the WPLL with a structure prior of $e^{-\alpha \sum_{i=1}^{F} d_i}$, where d_i is the number of literals that differ between the current version of the clause and the original one. (If the clause is new, this is simply its length.) This is similar to the approach used in learning Bayesian networks [14].

A potentially serious problem that arises when evaluating candidate clauses using WPLL is that the optimal (maximum WPLL) weights need to be computed for each candidate. Given that this involves numerical optimization, and may need to be done thousands or millions of times, it could easily make the algorithm too slow to be practical. We avoid this bottleneck by simply initializing L-BFGS with the current weights (and zero weight for a new clause). Second-order, quadratic-convergence methods like L-BFGS are known to be very fast if started near the optimum. This is what happens in our case; L-BFGS typically converges in just a few iterations, sometimes one. The time required to evaluate a clause is in fact dominated by the time required to compute the number of its true groundings in the data. This time can be greatly reduced using sampling and other techniques [18].

When learning an MLN from scratch (i.e., from a set of unit clauses), the natural operator to use is the addition of a literal to a clause. When refining a hand-coded KB, the goal is to correct the errors made by the human experts. These errors include omitting conditions from rules and including spurious ones, and can be corrected by operators that add and remove literals from a clause.

These are the basic operators that we use. In addition, we have found that many common errors (wrong direction of implication, wrong use of connectives with quantifiers, etc.) can be corrected at the clause level by flipping the signs of atoms, and we also allow this. When adding a literal to a clause, we consider all possible ways in which the literal's variables can be shared with existing ones, subject to the constraint that the new literal must contain at least one variable that appears in an existing one. To control the size of the search space, we set a limit on the number of distinct variables in a clause. We only try removing literals from the original hand-coded clauses or their descendants, and we only consider removing a literal if it leaves at least one path of shared variables between each pair of remaining literals.

We have implemented two search strategies, one faster and one more complete. The first approach adds clauses to the MLN one at a time, using beam search to find the best clause to add: starting with the unit clauses and the expert-supplied ones, we apply each legal literal addition and deletion to each clause, keep the b best ones, apply the operators to those, and repeat until no new clause improves the WPLL. The chosen clause is the one with highest WPLL found in any iteration of the search. If the new clause is a refinement of a hand-coded one, it replaces it. (Notice that, even though we both add and delete literals, no loops can occur because each change must improve WPLL to be accepted.)

The second approach adds k clauses at a time to the MLN, and is similar to that of McCallum [30]. In contrast to beam search, which adds the best clause of any length found, this approach adds all "good" clauses of length l before attempting any of length $l + 1$. We call it *shortest-first search.*

The algorithms described in the previous section may be very slow, particularly in large domains. However, they can be greatly sped up using a combination of techniques described in Kok and Domingos [18]. These include looser convergence thresholds, subsampling atoms and clauses, caching results, and ordering clauses to avoid evaluating the same candidate clause twice.

Recently, Mihalkova and Mooney [31] introduced BUSL, an alternative, bottom-up structure learning algorithm for Markov logic. Instead of blindly constructing candidate clauses one literal at a time, they let the training data guide and constrain clause construction. First, they use a propositional Markov network structure learner to generate a graph of relationships among atoms. Then they generate clauses from paths in this graph. In this way, BUSL focuses on clauses that have support in the training data. In experiments on three datasets, BUSL evaluated many fewer candidate clauses than our top-down algorithm, ran more quickly, and learned more accurate models.

We are currently investigating further approaches to learning MLNs, including automatically inventing new predicates (or, in statistical terms, discovering hidden variables) [19].

7 Applications

Markov logic has been successfully applied in a variety of areas. A system based on it recently won a competition on information extraction for biology [45]. Cycorp has used it to make parts of the Cyc knowledge base probabilistic [29]. The CALO project is using it to integrate probabilistic predictions from many components [8]. We have applied it to link prediction, collective classification, entity resolution, information extraction, social network analysis and other problems [44, 50, 18, 51, 40, 41]. Applications to Web mining, activity recognition, natural language processing, computational biology, robot mapping and navigation, game playing and others are under way.

7.1 Entity Resolution

The application to entity resolution illustrates well the power of Markov logic [51]. Entity resolution is the problem of determining which observations (e.g., database records, noun phrases, video regions, etc.) correspond to the same real-world objects, and is of crucial importance in many areas. Typically, it is solved by forming a vector of properties for each pair of observations, using a learned classifier (such as logistic regression) to predict whether they match, and applying transitive closure. Markov logic yields an improved solution simply by applying the standard logical approach of removing the unique names assumption and introducing the equality predicate and its axioms: equality is reflexive, symmetric and transitive; groundings of a predicate with equal constants have the same truth values; and constants appearing in a ground predicate with equal constants are equal. This last axiom is not valid in logic, but captures a useful statistical tendency. For example, if two papers are the same, their authors are the same; and if two authors are the same, papers by them are more likely to be the same. Weights for different instances of these axioms can be learned from data. Inference over the resulting MLN, with entity properties and relations as the evidence and equality atoms as the query, naturally combines logistic regression and transitive closure. Most importantly, it performs *collective* entity resolution, where resolving one pair of entities helps to resolve pairs of related entities.

As a concrete example, consider the task of deduplicating a citation database in which each citation has author, title, and venue fields. We can represent the domain structure with eight relations: $\texttt{Author}(\texttt{bib}, \texttt{author})$, $\texttt{Title}(\texttt{bib}, \texttt{title})$, and $\texttt{Venue}(\texttt{bib}, \texttt{venue})$ relate citations to their fields; $\texttt{HasWord}(\texttt{author/title/venue}, \texttt{word})$ indicates which words are present in each field; $\texttt{SameAuthor}(\texttt{author}, \texttt{author})$, $\texttt{SameTitle}(\texttt{title}, \texttt{title})$, and $\texttt{SameVenue}(\texttt{venue}, \texttt{venue})$ represent field equivalence; and $\texttt{SameBib}(\texttt{bib}, \texttt{bib})$ represents citation equivalence. The truth values of all relations except for the equivalence relations are provided as background theory. The objective is to predict the $\texttt{SameBib}$ relation.

We begin with a logistic regression model to predict citation equivalence based on the words in the fields. This is easily expressed in Markov logic by rules such as the following:

$$\begin{aligned}\texttt{Title(b1,t1)} \wedge \texttt{Title(b2,t2)} \wedge \texttt{HasWord(t1,+word)} \\ \wedge \texttt{HasWord(t2,+word)} \Rightarrow \texttt{SameBib(b1,b2)}\end{aligned}$$

The '+' operator here generates a separate rule (and with it, a separate learnable weight) for each constant of the appropriate type. When given a positive weight, each of these rules increases the probability that two citations with a particular title word in common are equivalent. We can construct similar rules for other fields. Note that we may learn negative weights for some of these rules, just as logistic regression may learn negative feature weights. Transitive closure consists of a single rule:

$$\texttt{SameBib(b1,b2)} \wedge \texttt{SameBib(b2,b3)} \Rightarrow \texttt{SameBib(b1,b3)}$$

This model is similar to the standard solution, but has the advantage that the classifier is learned in the context of the transitive closure operation.

We can construct similar rules to predict the equivalence of two fields as well. The usefulness of Markov logic is shown further when we link field equivalence to citation equivalence:

$$\begin{aligned}\texttt{Author(b1,a1)} \wedge \texttt{Author(b2,a2)} \wedge \texttt{SameBib(b1,b2)} \Rightarrow \texttt{SameAuthor(a1,a2)} \\ \texttt{Author(b1,a1)} \wedge \texttt{Author(b2,a2)} \wedge \texttt{SameAuthor(a1,a2)} \Rightarrow \texttt{SameBib(b1,b2)}\end{aligned}$$

The above rules state that if two citations are the same, their authors should be the same, and that citations with the same author are more likely to be the same. The last rule is not valid in logic, but captures a useful statistical tendency.

Most importantly, the resulting model can now perform *collective* entity resolution, where resolving one pair of entities helps to resolve pairs of related entities. For example, inferring that a pair of citations are equivalent can provide evidence that the names *AAAI-06* and *21st Natl. Conf. on AI* refer to the same venue, even though they are superficially very different. This equivalence can then aid in resolving other entities.

Experiments on citation databases like Cora and BibServ.org show that these methods can greatly improve accuracy, particularly for entity types that are difficult to resolve in isolation as in the above example [51]. Due to the large number of words and the high arity of the transitive closure formula, these models have thousands of weights and ground millions of clauses during learning, even after using canopies to limit the number of comparisons considered. Learning at this scale is still reasonably efficient: preconditioned scaled conjugate gradient with MC-SAT for inference converges within a few hours [26].

7.2 Information Extraction

In this citation example, it was assumed that the fields were manually segmented in advance. The goal of information extraction is to extract database records starting from raw text or semi-structured data sources. Traditionally, information extraction proceeds by first segmenting each candidate record separately, and then merging records that refer to the same entities. Such a pipeline architecture is adopted by many AI systems in natural language processing, speech recognition, vision, robotics, etc. Markov logic allows us to perform the two tasks jointly [41]. This enables us to use the segmentation of one candidate record to help segment similar ones. For example, resolving a well-segmented field with a less-clear one can disambiguate the latter's boundaries. We will continue with the example of citations, but similar ideas could be applied to other data sources, such as Web pages or emails.

The main evidence predicate in the information extraction MLN is $\texttt{Token}(\texttt{t}, \texttt{i}, \texttt{c})$, which is true iff token $\texttt{t}$ appears in the $\texttt{i}$th position of the $\texttt{c}$th citation. A token can be a word, date, number, etc. Punctuation marks are not treated as separate tokens; rather, the predicate $\texttt{HasPunc}(\texttt{c}, \texttt{i})$ is true iff a punctuation mark appears immediately after the $\texttt{i}$th position in the $\texttt{c}$th citation. The query predicates are $\texttt{InField}(\texttt{i}, \texttt{f}, \texttt{c})$ and $\texttt{SameCitation}(\texttt{c}, \texttt{c}')$. $\texttt{InField}(\texttt{i}, \texttt{f}, \texttt{c})$ is true iff the $\texttt{i}$th position of the $\texttt{c}$th citation is part of field $\texttt{f}$, where $\texttt{f} \in \{\texttt{Title}, \texttt{Author}, \texttt{Venue}\}$, and inferring it performs segmentation. $\texttt{SameCitation}(\texttt{c}, \texttt{c}')$ is true iff citations $\texttt{c}$ and $\texttt{c}'$ represent the same publication, and inferring it performs entity resolution.

Our segmentation model is essentially a hidden Markov model (HMM) with enhanced ability to detect field boundaries. The observation matrix of the HMM correlates tokens with fields, and is represented by the simple rule

$$\texttt{Token}(+\texttt{t}, \texttt{i}, \texttt{c}) \Rightarrow \texttt{InField}(\texttt{i}, +\texttt{f}, \texttt{c})$$

If this rule was learned in isolation, the weight of the (t, f)th instance would be $\log(p_{tf}/(1-p_{tf}))$, where p_{tf} is the corresponding entry in the HMM observation matrix. In general, the transition matrix of the HMM is represented by a rule of the form

$$\texttt{InField}(\texttt{i}, +\texttt{f}, \texttt{c}) \Rightarrow \texttt{InField}(\texttt{i}+1, +\texttt{f}', \texttt{c})$$

However, we (and others, e.g., [13]) have found that for segmentation it suffices to capture the basic regularity that consecutive positions tend to be part of the same field. Thus we replace $\texttt{f}'$ by $\texttt{f}$ in the formula above. We also impose the condition that a position in a citation string can be part of at most one field; it may be part of none.

The main shortcoming of this model is that it has difficulty pinpointing field boundaries. Detecting these is key for information extraction, and a number of approaches use rules designed specifically for this purpose (e.g., [21]). In citation matching, boundaries are usually marked by punctuation symbols. This can be incorporated into the MLN by modifying the rule above to

$$\texttt{InField}(\texttt{i}, +\texttt{f}, \texttt{c}) \wedge \neg\texttt{HasPunc}(\texttt{c}, \texttt{i}) \Rightarrow \texttt{InField}(\texttt{i}+1, +\texttt{f}, \texttt{c})$$

The $\neg\texttt{HasPunc}(\texttt{c}, \texttt{i})$ precondition prevents propagation of fields across punctuation marks. Because propagation can occur differentially to the left and right, the MLN also contains the reverse form of the rule. In addition, to account

Table 2. CiteSeer entity resolution: cluster recall on each section.

Approach	Constr.	Face	Reason.	Reinfor.
Fellegi-Sunter	84.3	81.4	71.3	50.6
Lawrence et al. (1999)	89	94	86	79
Pasula et al. (2002)	93	97	96	94
Wellner et al. (2004)	95.1	96.9	93.7	94.7
Joint MLN	96.0	97.1	95.1	96.7

for commas being weaker separators than other punctuation, the MLN includes versions of these rules with `HasComma`() instead of `HasPunc`().

Finally, the MLN contains rules capturing a variety of knowledge about citations: the first two positions of a citation are usually in the author field, and the middle one in the title; initials (e.g., "J.") tend to appear in either the author or the venue field; positions preceding the last non-venue initial are usually not part of the title or venue; and positions after the first venue keyword (e.g., "Proceedings", "Journal") are usually not part of the author or title.

By combining this segmentation model with our entity resolution model from before, we can exploit relational information as part of the segmentation process. In practice, something a little more sophisticated is necessary to get good results on real data. In Poon and Domingos [41], we define predicates and rules specifically for passing information between the stages, as opposed to just using the existing `InField`() outputs. This leads to a "higher bandwidth" of communication between segmentation and entity resolution, without letting excessive segmentation noise through. We also define an additional predicate and modify rules to better exploit information from similar citations during the segmentation process. See [41] for further details.

We evaluated this model on the CiteSeer and Cora datasets. For entity resolution in CiteSeer, we measured *cluster recall* for comparison with previously published results. Cluster recall is the fraction of clusters that are correctly output by the system after taking transitive closure from pairwise decisions. For entity resolution in Cora, we measured both cluster recall and pairwise recall/precision. In both datasets we also compared with a "standard" Fellegi-Sunter model (see [51]), learned using logistic regression, and with oracle segmentation as the input.

In both datasets, joint inference improved accuracy and our approach outperformed previous ones. Table 2 shows that our approach outperforms previous ones on CiteSeer entity resolution. (Results for Lawrence et al. (1999) [23], Pasula et al. (2002) [36] and Wellner et al. (2004) [58] are taken from the corresponding papers.) This is particularly notable given that the models of [36] and [58] involved considerably more knowledge engineering than ours, contained more learnable parameters, and used additional training data.

Table 3 shows that our entity resolution approach easily outperforms Fellegi-Sunter on Cora, and has very high pairwise recall/precision.

Table 3. Cora entity resolution: pairwise recall/precision and cluster recall.

Approach	Pairwise Rec./Prec.	Cluster Recall
Fellegi-Sunter	78.0 / 97.7	62.7
Joint MLN	94.3 / 97.0	78.1

8 The Alchemy System

The inference and learning algorithms described in the previous sections are publicly available in the open-source Alchemy system [20]. Alchemy makes it possible to define sophisticated probabilistic models with a few formulas, and to add probability to a first-order knowledge base by learning weights from a relevant database. It can also be used for purely logical or purely statistical applications, and for teaching AI. From the user's point of view, Alchemy provides a full spectrum of AI tools in an easy-to-use, coherent form. From the researcher's point of view, Alchemy makes it possible to easily integrate a new inference or learning algorithm, logical or statistical, with a full complement of other algorithms that support it or make use of it.

Alchemy can be viewed as a declarative programming language akin to Prolog, but with a number of key differences: the underlying inference mechanism is model checking instead of theorem proving; the full syntax of first-order logic is allowed, rather than just Horn clauses; and, most importantly, the ability to handle uncertainty and learn from data is already built in. Table 4 compares Alchemy with Prolog and BUGS [28], one of the most popular toolkits for Bayesian modeling and inference.

Table 4. A comparison of Alchemy, Prolog and BUGS.

Aspect	Alchemy	Prolog	BUGS
Representation	First-order logic + Markov nets	Horn clauses	Bayes nets
Inference	Model checking, MCMC	Theorem proving	MCMC
Learning	Parameters and structure	No	Parameters
Uncertainty	Yes	No	Yes
Relational	Yes	Yes	No

9 Current and Future Research Directions

We are actively researching better learning and inference methods for Markov logic, as well as extensions of the representation that increase its generality and power.

Exact methods for learning and inference are usually intractable in Markov logic, but we would like to see better, more efficient approximations along with the automatic application of exact methods when feasible.

One method of particular interest is lifted inference. In short, we would like to reason with clusters of nodes for which we have exactly the same amount of information. The inspiration is from lifted resolution in first order logic, but must be extended to handle uncertainty. Prior work on lifted inference such as [39] and [3] mainly focused on exact inference which can be quite slow. There has been some recent work on lifted belief propagation in a Markov logic like setting [15], but only for the case in which there is no evidence. We would like to extend this body of work for approximate inference in the case where arbitrary evidence is given, potentially speeding up inference in Markov logic by orders of magnitude.

Numerical attributes must be discretized to be used in Markov logic, but we are working on extending the representation to handle continuous random variables and features. This is particularly important in domains like robot navigation, where the coordinates of the robot and nearby obstacles are real-valued. Even domains that are handled well by Markov logic, such as entity resolution, could still benefit from this extension by incorporating numeric features into similarities.

Another extension of Markov logic is to support uncertainty at multiple levels in the logical structure. A formula in first-order logic can be viewed as a tree, with a logical connective at each node, and a knowledge base can be viewed as a tree whose root is a conjunction. Markov logic makes this conjunction probabilistic, as well as the universal quantifiers directly under it, but the rest of the tree remains purely logical. Recursive random fields [27] overcome this by allowing the features to be nested MLNs instead of clauses. Unfortunately, learning them suffers from the limitations of backpropagation.

Statistical predicate invention is the problem of discovering new concepts, properties, and relations in structured data, and generalizes hidden variable discovery in statistical models and predicate invention in ILP. Rather than extending the model directly, statistical predicate invention enables richer models by extending the domain with discovered predicates. Our initial work in this area uses second-order Markov logic to generate multiple cross-cutting clusterings of constants and predicates [19]. Formulas in second-order Markov logic could also be used to add declarative bias to our structure learning algorithms.

Current work also includes semi-supervised learning, and learning with incomplete data in general. The large amount of unlabeled data on the Web is an excellent resource that, properly exploited, could lead to many exciting applications.

Finally, we would like to develop a general framework for decision-making in relational domains. This can be accomplished in Markov logic by adding utility weights to formulas and finding the settings of all action predicates that jointly maximize expected utility.

10 Conclusion

Markov logic is a simple yet powerful approach to combining logic and probability in a single representation. We have developed a series of learning and inference algorithms for it, and successfully applied them in a number of domains. These algorithms are available in the open-source Alchemy system. We hope that Markov logic and its implementation in Alchemy will be of use to researchers and practitioners who wish to have the full spectrum of logical and statistical inference and learning techniques at their disposal, without having to develop every piece themselves.

11 Acknowledgements

This research was partly supported by DARPA grant FA8750-05-2-0283 (managed by AFRL), DARPA contract NBCH-D030010, NSF grant IIS-0534881, ONR grants N00014-02-1-0408 and N00014-05-1-0313, a Sloan Fellowship and NSF CAREER Award to the first author, and a Microsoft Research fellowship awarded to the third author. 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, NSF, ONR, or the United States Government.

References

1. S. Becker and Y. Le Cun. Improving the convergence of back-propagation learning with second order methods. In *Proceedings of the 1988 Connectionist Models Summer School*, pages 29–37, San Mateo, CA, 1989. Morgan Kaufmann.
2. J. Besag. Statistical analysis of non-lattice data. *The Statistician*, 24:179–195, 1975.
3. R. Braz, E. Amir, and D. Roth. Lifted first-order probabilistic inference. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, pages 1319–1325, Edinburgh, UK, 2005. Morgan Kaufmann.
4. M. Collins. Discriminative training methods for hidden Markov models: Theory and experiments with perceptron algorithms. In *Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing*, pages 1–8, Philadelphia, PA, 2002. ACL.
5. P. Damien, J. Wakefield, and S. Walker. Gibbs sampling for Bayesian nonconjugate and hierarchical models by auxiliary variables. *Journal of the Royal Statistical Society, Series B*, 61, 1999.
6. L. De Raedt and L. Dehaspe. Clausal discovery. *Machine Learning*, 26:99–146, 1997.
7. S. Della Pietra, V. Della Pietra, and J. Lafferty. Inducing features of random fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19:380–392, 1997.
8. T. Dietterich. Experience with Markov logic networks in a large AI system. In *Probabilistic, Logical and Relational Learning - Towards a Synthesis*, number 05051 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Dagstuhl, Germany, 2007.

9. N. Friedman, L. Getoor, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 1300–1307, Stockholm, Sweden, 1999. Morgan Kaufmann.
10. M. R. Genesereth and N. J. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann, San Mateo, CA, 1987.
11. L. Getoor and B. Taskar, editors. *Introduction to Statistical Relational Learning*. MIT Press, Cambridge, MA, 2007.
12. W. R. Gilks, S. Richardson, and D. J. Spiegelhalter, editors. *Markov Chain Monte Carlo in Practice*. Chapman and Hall, London, UK, 1996.
13. T. Grenager, D. Klein, and C. D. Manning. Unsupervised learning of field segmentation models for information extraction. In *Proceedings of the Forty-Third Annual Meeting on Association for Computational Linguistics*, pages 371–378, Ann Arbor, Michigan, 2005. Association for Computational Linguistics.
14. D. Heckerman, D. Geiger, and D. M. Chickering. Learning Bayesian networks: The combination of knowledge and statistical data. *Machine Learning*, 20:197–243, 1995.
15. A. Jaimovich, O. Meshi, and N. Friedman. Template based inference in symmetric relational markov random fields. In *Proceedings of the Twenty-Third Conference on Uncertainty in Artificial Intelligence*, Vancouver, Canada, 2007. AUAI Press.
16. H. Kautz, B. Selman, and Y. Jiang. A general stochastic approach to solving problems with hard and soft constraints. In D. Gu, J. Du, and P. Pardalos, editors, *The Satisfiability Problem: Theory and Applications*, pages 573–586. American Mathematical Society, New York, NY, 1997.
17. K. Kersting and L. De Raedt. Towards combining inductive logic programming with Bayesian networks. In *Proceedings of the Eleventh International Conference on Inductive Logic Programming*, pages 118–131, Strasbourg, France, 2001. Springer.
18. S. Kok and P. Domingos. Learning the structure of Markov logic networks. In *Proceedings of the Twenty-Second International Conference on Machine Learning*, pages 441–448, Bonn, Germany, 2005. ACM Press.
19. S. Kok and P. Domingos. Statistical predicate invention. In *Proceedings of the Twenty-Fourth International Conference on Machine Learning*, pages 433–440, Corvallis, OR, 2007. ACM Press.
20. S. Kok, M. Sumner, M. Richardson, P. Singla, H. Poon, D. Lowd, and P. Domingos. The Alchemy system for statistical relational AI. Technical report, Department of Computer Science and Engineering, University of Washington, Seattle, WA, 2007. http://alchemy.cs.washington.edu.
21. N. Kushmerick. Wrapper induction: Efficiency and expressiveness. *Artificial Intelligence*, 118(1-2):15–68, 2000.
22. N. Lavrač and S. Džeroski. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, Chichester, UK, 1994.
23. S. Lawrence, K. Bollacker, and C. L. Giles. Autonomous citation matching. In *Proceedings of the Third International Conference on Autonomous Agents*, New York, 1999. ACM Press.
24. D. C. Liu and J. Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45(3):503–528, 1989.
25. J. W. Lloyd. *Foundations of Logic Programming*. Springer, Berlin, Germany, 1987.
26. D. Lowd and P. Domingos. Efficient weight learning for Markov logic networks. In *Proceedings of the Eleventh European Conference on Principles and Practice of Knowledge Discovery in Databases*, pages 200–211, Warsaw, Poland, 2007. Springer.

27. D. Lowd and P. Domingos. Recursive random fields. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*, Hyderabad, India, 2007. AAAI Press.
28. D. J. Lunn, A. Thomas, N. Best, and D. Spiegelhalter. WinBUGS – a Bayesian modeling framework: concepts, structure, and extensibility. *Statistics and Computing*, 10:325–337, 2000.
29. C. Matuszek and M. Witbrock. Personal communication. 2006.
30. A. McCallum. Efficiently inducing features of conditional random fields. In *Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence*, Acapulco, Mexico, 2003. Morgan Kaufmann.
31. L. Mihalkova and R. Mooney. Bottom-up learning of Markov logic network structure. In *Proceedings of the Twenty-Fourth International Conference on Machine Learning*, pages 625–632, Corvallis, OR, 2007. ACM Press.
32. M. Møller. A scaled conjugate gradient algorithm for fast supervised learning. *Neural Networks*, 6:525–533, 1993.
33. S. Muggleton. Stochastic logic programs. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, pages 254–264. IOS Press, Amsterdam, Netherlands, 1996.
34. J. Neville and D. Jensen. Dependency networks for relational data. In *Proceedings of the Fourth IEEE International Conference on Data Mining*, pages 170–177, Brighton, UK, 2004. IEEE Computer Society Press.
35. J. Nocedal and S. Wright. *Numerical Optimization*. Springer, New York, NY, 2006.
36. H. Pasula, B. Marthi, B. Milch, S. Russell, and I. Shpitser. Identity uncertainty and citation matching. In *Advances in Neural Information Processing Systems 14*, Cambridge, MA, 2002. MIT Press.
37. J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Francisco, CA, 1988.
38. B. Pearlmutter. Fast exact multiplication by the Hessian. *Neural Computation*, 6(1):147–160, 1994.
39. D. Poole. First-order probabilistic inference. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 985–991, Acapulco, Mexico, 2003. Morgan Kaufmann.
40. H. Poon and P. Domingos. Sound and efficient inference with probabilistic and deterministic dependencies. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, pages 458–463, Boston, MA, 2006. AAAI Press.
41. H. Poon and P. Domingos. Joint inference in information extraction. In *Proceedings of the Twenty-Second National Conference on Artificial Intelligence*, pages 913–918, Vancouver, Canada, 2007. AAAI Press.
42. J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.
43. L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77:257–286, 1989.
44. M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62:107–136, 2006.
45. S. Riedel and E. Klein. Genic interaction extraction with semantic and syntactic chains. In *Proceedings of the Fourth Workshop on Learning Language in Logic*, pages 69–74, Bonn, Germany, 2005. IMLS.
46. J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.
47. D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82:273–302, 1996.

48. F. Sha and F. Pereira. Shallow parsing with conditional random fields. In *Proceedings of the 2003 Human Language Technology Conference and North American Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics, 2003.
49. J. Shewchuck. An introduction to the conjugate gradient method without the agonizing pain. Technical Report CMU-CS-94-125, School of Computer Science, Carnegie Mellon University, 1994.
50. P. Singla and P. Domingos. Discriminative training of Markov logic networks. In *Proceedings of the Twentieth National Conference on Artificial Intelligence*, pages 868–873, Pittsburgh, PA, 2005. AAAI Press.
51. P. Singla and P. Domingos. Entity resolution with Markov logic. In *Proceedings of the Sixth IEEE International Conference on Data Mining*, pages 572–582, Hong Kong, 2006. IEEE Computer Society Press.
52. P. Singla and P. Domingos. Memory-efficient inference in relational domains. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, Boston, MA, 2006. AAAI Press.
53. P. Singla and P. Domingos. Markov logic in infinite domains. In *Proceedings of the Twenty-Third Conference on Uncertainty in Artificial Intelligence*, pages 368–375, Vancouver, Canada, 2007. AUAI Press.
54. A. Srinivasan. The Aleph manual. Technical report, Computing Laboratory, Oxford University, 2000.
55. S. Wasserman and K. Faust. *Social Network Analysis: Methods and Applications*. Cambridge University Press, Cambridge, UK, 1994.
56. W. Wei, J. Erenrich, and B. Selman. Towards efficient sampling: Exploiting random walk strategies. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence*, San Jose, CA, 2004. AAAI Press.
57. M. Wellman, J. S. Breese, and R. P. Goldman. From knowledge bases to decision models. *Knowledge Engineering Review*, 7, 1992.
58. B. Wellner, A. McCallum, F. Peng, and M. Hay. An integrated, conditional model of information extraction and coreference with application to citation matching. In *Proceedings of the Twentieth Conference on Uncertainty in Artificial Intelligence*, pages 593–601, Banff, Canada, 2004. AUAI Press.
59. J. S. Yedidia, W. T. Freeman, and Y. Weiss. Generalized belief propagation. In T. Leen, T. Dietterich, and V. Tresp, editors, *Advances in Neural Information Processing Systems 13*, pages 689–695. MIT Press, Cambridge, MA, 2001.