Markov Logic:
A Language and Algorithms for Link Mining

Pedro Domingos[1], Daniel Lowd[2], Stanley Kok[1], Aniruddh Nath[1], Hoifung Poon[1], Matthew Richardson[3], and Parag Singla[4]

[1] Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195-2350
{pedrod, koks, nath, hoifung}@cs.washington.edu
[2] Department of Computer and Information Science
University of Oregon
Eugene, OR 97403-1202
lowd@uoregon.edu
[3] Microsoft Research
Redmond, WA 98052
mattri@microsoft.com
[4] Department of Computer Science
The University of Texas at Austin
1 University Station C0500
Austin, TX 78712-0233
paragsingla@gmail.com

Abstract. Link mining problems are characterized by high complexity (since linked objects are not statistically independent) and uncertainty (since data is noisy and incomplete). Thus they necessitate a modeling language that is both probabilistic and relational. Markov logic provides this by attaching weights to formulas in first-order logic and viewing them as templates for features of Markov networks. Many link mining problems can be elegantly formulated and efficiently solved using Markov logic. Inference algorithms for Markov logic draw on ideas from satisfiability testing, Markov chain Monte Carlo, belief propagation and resolution. Learning algorithms are based on convex optimization, pseudo-likelihood and inductive logic programming. Markov logic has been used successfully in a wide variety of link mining applications, and is the basis of the open-source Alchemy system.

1   Introduction

Most objects and entities in the world are not independent, but are instead linked to many other objects through a diverse set of relationships: people have friends, family, and coworkers; scientific papers have authors, venues, and references to other papers; Web pages link to other Web pages and have hierarchical structure; proteins have locations and functions, and interact with other proteins. In these

examples, as in many others, the context provided by these relationships is essential for understanding the entities themselves. Furthermore, the relationships are often worthy of analysis in their own right. In link mining, the connections among objects are explicitly modeled to improve performance in tasks such as classification, clustering, and ranking, as well as enabling new applications, such as link prediction.

As link mining grows in popularity, the number of link mining problems and approaches continues to multiply. Rather than solving each problem and developing each technique in isolation, we need a common representation language for link mining. Such a language would serve as an interface layer between link mining applications and the algorithms used to solve them, much as the Internet serves as an interface layer for networking, relational models serve as an interface layer for databases, etc. This would both unify many approaches and lower the barrier of entry to new researchers and practitioners.

At a minimum, a formal language for link mining must be (a) relational and (b) probabilistic. Link mining problems are clearly relational, since each link among objects can be viewed as a relation. First-order logic is a powerful and flexible way to represent relational knowledge. Important concepts such as transitivity (e.g., "My friend's friend is also my friend"), homophily (e.g., "Friends have similar smoking habits"), and symmetry (e.g., "Friendship is mutual") can be expressed as short formulas in first-order logic. It is also possible to represent much more complex, domain-specific rules, such as "Each graduate student coauthors at least one publication with his or her advisor."

Most link mining problems have a great deal of uncertainty as well. Link data is typically very noisy and incomplete. Even with a perfect model, few questions can be answered with certainty due to limited evidence and inherently stochastic domains. The standard language for modeling uncertainty is probability. In particular, probabilistic graphical models have proven an effective tool in solving a wide variety of problems in data mining and machine learning.

Since link mining problems are both relational and uncertain, they require methods that combine logic and probability. Neither one alone suffices: first-order logic is too brittle, and does not handle uncertainty; standard graphical models assume data points are i.i.d. (independent and identically distributed), and do not handle the relational dependencies and variable-size networks present in link mining problems.

Markov logic [7] is a simple yet powerful generalization of probabilistic graphical models and first-order logic, making it ideally suited for link mining. A *Markov logic network* is a set of weighted first-order formulas, viewed as templates for constructing Markov networks. This yields a well-defined probability distribution in which worlds are more likely when they satisfy a higher-weight set of ground formulas. Intuitively, the magnitude of the weight corresponds to the relative strength of its formula; in the infinite-weight limit, Markov logic reduces to first-order logic. Weights can be set by hand or learned automatically from data. Algorithms for learning or revising formulas from data have also been developed. Inference algorithms for Markov logic combine ideas from probabilistic

and logical inference, including Markov chain Monte Carlo, belief propagation, satisfiability, and resolution.

Markov logic has already been used to efficiently develop state-of-the-art models for many link mining problems, including collective classification, link-based clustering, record linkage, and link prediction, in application areas such as the Web, social networks, molecular biology, information extraction, and others. Markov logic makes link mining easier by offering a simple framework for representing well-defined probability distributions over uncertain, relational data. Many existing approaches can be described by a few weighted formulas, and multiple approaches can be combined by including all of the relevant formulas. Many algorithms, as well as sample datasets and applications, are available in the open-source Alchemy system [17] (alchemy.cs.washington.edu).

In this chapter, we describe Markov logic and its algorithms, and show how they can be used as a general framework for link mining. We begin with background on first-order logic and Markov networks. We then define Markov logic and a few of its basic extensions. Next, we discuss a number of inference and learning algorithms. Finally, we show two link mining applications, each of which can be written in just a few formulas and solved using the previous algorithms.

2 First-Order Logic

A *first-order knowledge base (KB)* is a set of sentences or formulas in first-order logic [9]. 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 `x` might range over people (e.g., Anna, Bob, etc.), and the constant `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`, `x`, and `GreatestCommonDivisor(x, y)` are terms. An *atomic formula* or *atom* is a predicate symbol applied to a tuple of terms (e.g., `Friends(x, MotherOf(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; $\forall x \, F_1$ (universal quantification), which is true iff F_1 is true for every object x

in the domain; and $\exists x\ F_1$ (existential quantification), which is true iff F_1 is true for at least one object 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.[5] Clausal form is used in resolution, a sound and refutation-complete inference procedure for first-order logic [34].

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 [20]. 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) [18].

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 link mining 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 such as Markov networks, described in the next section. We will later show how to generalize these models to the first-order case.

3   Markov Networks

A *Markov network* (also known as *Markov random field*) is a model for the joint distribution of a set of variables $X = (X_1, X_2, \ldots, X_n) \in \mathcal{X}$ [27]. It is composed

---

[5] 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.

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

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

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.

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. [32] *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 $\forall x\, \texttt{Smokes}(x) \Rightarrow \texttt{Cancer}(x)$ (smoking causes cancer) and $\forall x \forall y\, \texttt{Friends}(x, y) \Rightarrow (\texttt{Smokes}(x) \Leftrightarrow \texttt{Smokes}(y))$ (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 $\texttt{Smokes(Anna)} \Rightarrow \texttt{Cancer(Anna)}$, 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 [43]. Notice also that nodes and links in the social networks are both represented as nodes in the Markov network; arcs in the Markov network represent probabilistic dependencies between nodes and links in the social network (e.g., Anna's smoking habits depend on her friends' smoking habits).

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

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

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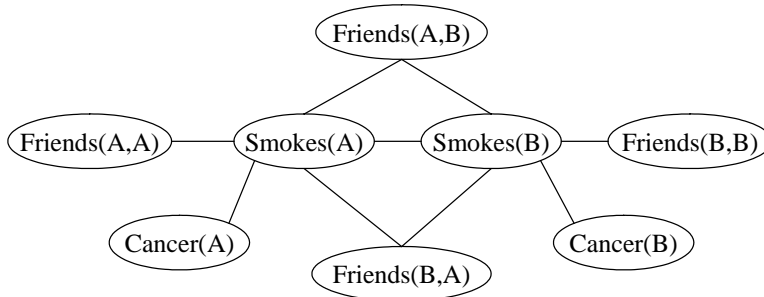
Fig. 1. Ground Markov network obtained by applying an MLN containing the formulas $\forall x\, \texttt{Smokes(x)} \Rightarrow \texttt{Cancer(x)}$ and $\forall x \forall y\, \texttt{Friends(x, y)} \Rightarrow (\texttt{Smokes(x)} \Leftrightarrow \texttt{Smokes(y)})$ to the constants $\texttt{Anna(A)}$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

represented by the ground Markov network. Most importantly, Markov logic allows contradictions between formulas, which it resolves simply by weighing the evidence on both sides.

It is interesting to see a simple example of how Markov logic generalizes first-order logic. Consider an MLN containing the single formula $\forall x\, \texttt{R(x)} \Rightarrow \texttt{S(x)}$ with weight w, and $C = \{\texttt{A}\}$. This leads to four possible worlds: $\{\neg\texttt{R(A)}, \neg\texttt{S(A)}\}$, $\{\neg\texttt{R(A)}, \texttt{S(A)}\}$, $\{\texttt{R(A)}, \neg\texttt{S(A)}\}$, and $\{\texttt{R(A)}, \texttt{S(A)}\}$. From Equation 3 we obtain that $P(\{\texttt{R(A)}, \neg\texttt{S(A)}\}) = 1/(3e^w + 1)$ 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 3.) Thus, if $w > 0$, the effect of the MLN is to make the world that is inconsistent with $\forall x\, \texttt{R(x)} \Rightarrow \texttt{S(x)}$ less likely than the other three. From the probabilities above we obtain that $P(\texttt{S(A)}|\texttt{R(A)}) = 1/(1 + e^{-w})$. When $w \to \infty$, $P(\texttt{S(A)}|\texttt{R(A)}) \to 1$, 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 machine learning and data mining 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). The application section shows how to describe logistic regression in Markov logic and easily extend it to perform collective classification over a set of linked objects.

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 [32] 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 [39] for details on Markov logic in infinite domains.

For decision theoretic problems, such as the viral marketing application we will discuss later, we can easily extend MLNs to Markov logic decision networks (MLDNs) by attaching a utility to each formula as well as a weight [25]. The utility of a world is the sum of the utilities of its satisfied formulas. Just as an MLN plus a set of constants defines a Markov network, an MLDN plus a set of constants defines a Markov decision network. The optimal decision is the setting of the action predicates that jointly maximizes expected utility.

5   Inference

Given an MLN model of a link mining problem, the questions of interest are answered by performing inference on it. (For example, "What are the topics of these Web pages, given the words on them and the links between them?") Recall that an MLN acts as a template for a Markov network. Therefore, we can always answer queries using standard Markov network inference methods on the instantiated network. Several of these methods have been extended to take particular advantage of the logical structure in an MLN, yielding tremendous savings in memory and time. We first provide an overview of inference in Markov networks, and then describe how these methods can be adapted to take advantage of MLN structure.

5.1   Markov Network Inference

The main inference problem in Markov networks is computing the probabilities of query variables given some evidence, and is #P-complete [35]. The most widely used method for approximate inference in Markov networks is Markov chain Monte Carlo (MCMC) [10], 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 [46]. Belief propagation is an algorithm for computing the exact marginal probability of each query variable in a tree-structured graphical model. The method consists of passing messages between variables and the potential functions they participate in. The message from a variable x to a potential function

f is

$$\mu_{x \to f}(x) = \prod_{h \in nb(x) \setminus \{f\}} \mu_{h \to x}(x) \qquad (4)$$

where $nb(x)$ is the set of potentials x appears in. The message from a potential function to a variable is

$$\mu_{f \to x}(x) = \sum_{\sim \{x\}} \left(f(\mathbf{x}) \prod_{y \in nb(f) \setminus \{x\}} \mu_{y \to f}(y) \right) \qquad (5)$$

where $nb(f)$ are the variables in f, and the sum is over all of these except x. In a tree, the messages from leaf variables are initialized to 1, and a pass from the leaves to the root and back to the leaves suffices. The (unnormalized) marginal of each variable x is then given by $\prod_{h \in nb(x)} \mu_{h \to x}(x)$. Evidence is incorporated by setting $f(\mathbf{x}) = 0$ for states $\mathbf{x}$ that are incompatible with it. This algorithm can still be applied when the graph has loops, repeating the message-passing until convergence. Although this *loopy* belief propagation has no guarantees of convergence or of giving accurate results, in practice it often does, and can be much more efficient than other methods.

Another 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. (MAP means "maximum *a posteriori*," and MPE means "most probable explanation.") It is NP-hard. Notice that MAP inference cannot be solved simply by computing the probability of each random variable and then assigning the most probable value, because the combination of two assignments that are individually probable may itself be improbable or even impossible. Belief propagation can also be used to solve the MAP problem, by replacing summation with maximization in Equation 5. Other popular methods include greedy search, simulated annealing and graph cuts.

We first look at how to perform MAP inference, and then at computing probabilities. 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.

5.2 MAP/MPE Inference

Because of the form of Equation 3, in Markov logic the MAP inference problem 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.) The Alchemy system uses MaxWalkSAT, a weighted variant of the WalkSAT local-search satisfiability solver, which can solve hard problems with hundreds of thousands of variables in minutes [12]. MaxWalkSAT performs this stochastic search by picking an unsatisfied clause at random and flipping the truth value of

Table 2. MaxWalkSAT algorithm for MPE inference.

---

function MaxWalkSAT($L, t_{\max}, f_{\max}, target, p$)
   inputs: L, a set of weighted clauses
            $t_{\max}$, the maximum number of tries
            $f_{\max}$, the maximum number of flips
            $target$, target solution cost
            p, probability of taking a random step
   output: $soln$, best variable assignment found
$vars \leftarrow$ variables in L
for $i \leftarrow 1$ **to** $t_{\max}$
   $soln \leftarrow$ a random truth assignment to $vars$
   $cost \leftarrow$ sum of weights of unsatisfied clauses in $soln$
   for $i \leftarrow 1$ **to** $f_{\max}$
      if $cost \leq target$
         return "Success, solution is", $soln$
      $c \leftarrow$ a randomly chosen unsatisfied clause
      if Uniform(0,1) $< p$
         $v_f \leftarrow$ a randomly chosen variable from c
      else
         for each variable v in c
            compute DeltaCost(v)
         $v_f \leftarrow v$ with lowest DeltaCost(v)
      $soln \leftarrow soln$ with v_f flipped
      $cost \leftarrow cost +$ DeltaCost(v_f)
return "Failure, best assignment is", best $soln$ found

---

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 MaxWalk-SAT to avoid getting stuck in local optima while searching. Pseudocode for MaxWalkSAT is shown in Table 2. 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 $[0, 1]$.

MAP inference in Markov logic can also be performed using cutting plane methods [33] and others.

5.3 Marginal and Conditional Probabilities

We now consider the task of computing the probability that a formula holds, given an MLN and set of constants, and possibly other formulas as evidence. 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 [45]. 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 appear in the query formula 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 directly depend on (i.e., the atoms that appear in some formula with them). 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 [32] for details.

Given the relevant ground network, inference can be performed using standard methods like MCMC and belief propagation. One problem with this is that these methods break down in the presence of deterministic or near-deterministic dependencies. 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 [28]. 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. 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.

Slice sampling [4] 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 [44] 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

Table 3. Efficient MCMC inference algorithm for MLNs.

---

function MC-SAT(L, n)
    inputs: L, a set of weighted clauses $\{(w_j, c_j)\}$
              n, number of samples
    output: $\{x^{(1)}, \ldots, x^{(n)}\}$, set of n samples
  $x^{(0)} \leftarrow$ Satisfy(hard clauses in L)
  for $i \leftarrow 1$ **to** n
    $M \leftarrow \emptyset$
    for all $(w_k, c_k) \in L$ satisfied by $x^{(i-1)}$
      With probability $1 - e^{-w_k}$ add c_k to M
    Sample $x^{(i)} \sim \mathcal{U}_{SAT(M)}$

---

states $SAT(M)$ that satisfy M. (Notice that $SAT(M)$ is never empty, because it always contains at least the current state.) Table 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 [28], assuming a perfect uniform sampler. In general, uniform sampling is #P-hard and SampleSAT [44] 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.

5.4 Scaling Up Inference

Lazy Inference One problem with the aforementioned approaches is that they require propositionalizing the domain (i.e., grounding all atoms and clauses in all possible ways), which consumes memory exponential in the arity of the clauses. Lazy inference methods [38, 29] overcome this by only grounding atoms and clauses as needed. 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 papers, most groundings of the atom `Author(person, paper)` are false, and most groundings of the clause `Author(p1, paper)` $\wedge$ `Author(p2, paper)` $\Rightarrow$ `Coauthor(p1, p2)` are trivially satisfied. With lazy inference, the memory cost does not scale with the number of possible clause groundings, but only with the number of groundings that have non-default values at some point in the inference.

We first describe a general approach for making inference algorithms lazy and then show how it can be applied to create a lazy version of MaxWalkSAT. We have also developed a lazy version of MC-SAT. Working implementations of both algorithms are available in the Alchemy system. See Poon *et al.* [29] for more details.

Our approach depends on the concept of "default" values that occur much more frequently than others. In relational domains, the default is false for atoms and true for clauses. In a domain where most variables assume the default value, it is wasteful to allocate memory for all variables and functions in advance. The basic idea is to allocate memory only for a small subset of "active" variables and functions, and activate more if necessary as inference proceeds. In addition to saving memory, this can reduce inference time as well, since we do not allocate memory or compute values for functions that are never used.

Definition 2. *Let X be the set of variables and D be their domain.[6] The* default value $d^* \in D$ *is the most frequent value of the variables. An* evidence variable *is a variable whose value is given and fixed. A* function $f = f(z_1, z_2, \cdots, z_k)$ *inputs z_i's, which are either variables or functions, and outputs some value in the range of f.*

Although these methods can be applied to other inference algorithms, we focus on relational domains. Variables are ground atoms, which take binary values (i.e., $D = \{true, false\}$). The default value for variables is false (i.e., $d^* = false$). Examples of functions are clauses and DeltaCost in MaxWalkSAT (Table 2). Like variables, functions may also have default values (e.g., true for clauses). The inputs to a relational inference algorithm are a weighted KB and a set of evidence atoms (DB). Eager algorithms work by first carrying out propositionalization and then calling a propositional algorithm. In lazy inference, we directly work on the KB and DB. The following concepts are crucial to lazy inference.

Definition 3. *A variable v is* active *iff v is set to a non-default value at some point, and x is* inactive *if the value of x has always been d^*. A function f is* activated *by a variable v if either v is an input of f, or v activates a function g that is an input of f.*

Let $\mathcal{A}$ be the eager algorithm that we want to make lazy. We make three assumptions about $\mathcal{A}$:

1. $\mathcal{A}$ updates one variable at a time. (If not, the extension is straightforward.)
2. The values of variables in $\mathcal{A}$ are properly encapsulated so that they can be accessed by the rest of the algorithm only via two methods: ReadVar(x) (which returns the value of x) and WriteVar(x, v) (which sets the value of x to v). This is reasonable given the conventions in software development, and if not, it is easy to implement.

---

[6] For simplicity we assume that all variables have the same domain. The extension to different domains is straightforward.

3. $\mathcal{A}$ always sets values of variables before calling a function that depends on those variables, as it should be.

To develop the lazy version of $\mathcal{A}$, we first identify the variables (usually all) and functions to make lazy. We then modify the value-accessing methods and replace the propositionalization step with lazy initialization as follows. The rest of the algorithm remains the same.

ReadVar(x): If x is in memory, Lazy-$\mathcal{A}$ returns its value as $\mathcal{A}$; otherwise, it returns d^*.

WriteVar(x, v): If x is in memory, Lazy-$\mathcal{A}$ updates its value as $\mathcal{A}$. If not, and if $v = d^*$, no action is taken; otherwise, Lazy-$\mathcal{A}$ activates (allocates memory for) x and the functions activated by x, and then sets the value.

Initialization: Lazy-$\mathcal{A}$ starts by allocating memory for the lazy functions that output non-default values when all variables assume the default values. It then calls WriteVar to set values for evidence variables, which activates those evidence variables with non-default values and the functions they activate. Such variables become the initial active variables and their values are fixed throughout the inference.

Lazy-$\mathcal{A}$ carries out the same inference steps as $\mathcal{A}$ and produces the same result. It never allocates memory for more variables/functions than $\mathcal{A}$, but each access incurs slightly more overhead (in checking whether a variable or function is in memory). In the worst case, most variables are updated, and Lazy-$\mathcal{A}$ produces little savings. However, if the updates are sparse, as is the case for most algorithms in relational domains, Lazy-$\mathcal{A}$ can greatly reduce memory and time because it activates and computes the values for many fewer variables and functions.

Applying this method to MaxWalkSAT is fairly straightforward: each ground atom is a variable and each ground clause is a function to be made lazy. Following Singla and Domingos [38], we refer to the resulting algorithm as LazySAT. LazySAT initializes by activating true evidence atoms and initial unsatisfied clauses (i.e., clauses which are unsatisfied when the true evidence atoms are set to true and all other atoms are set to false).[7] At each step in the search, the atom that is flipped is activated, as are any clauses that by definition should become active as a result. While computing DeltaCost(v), if v is active, the relevant clauses are already in memory; otherwise, they will be activated when v is set to true (a necessary step before computing the cost change when v is set to true). Table 4 gives pseudocode for LazySAT.

Experiments in a number of domains show that LazySAT can yield very large memory reductions, and these reductions increase with domain size [38]. For domains whose full instantiations fit in memory, running time is comparable; as problems become larger, full instantiation for MaxWalkSAT becomes impossible.

---

[7] This differs from MaxWalkSAT, which assigns random values to all atoms. However, the LazySAT initialization is a valid MaxWalkSAT initialization, and the two give very similar results empirically. Given the same initialization, the two algorithms will produce exactly the same results.

Table 4. Lazy variant of the MaxWalkSAT algorithm.

---

function LazySAT(KB, DB, $t_{\max}$, $f_{\max}$, $target$, p)
    inputs: KB, a weighted knoweldge base
              DB, database containing evidence
              $t_{\max}$, the maximum number of tries
              $f_{\max}$, the maximum number of flips
              $target$, target solution cost
              p, probability of taking a random step
    output: $soln$, best variable assignment found

for $i \leftarrow 1$ to $t_{\max}$
    $active\_atoms \leftarrow$ 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 $f_{\max}$
        if $cost \leq target$
            return "Success, solution is", $soln$
        $c \leftarrow$ a randomly chosen unsatisfied clause
        if Uniform(0,1) $< p$
            $v_f \leftarrow$ a randomly chosen variable from c
        else
            for each variable v in c
                compute DeltaCost(v), using KB if $v \notin active\_atoms$
            $v_f \leftarrow v$ with lowest DeltaCost(v)
        if $v_f \notin active\_atoms$
            add v_f to $active\_atoms$
            add clauses activated by v_f to $active\_clauses$
        $soln \leftarrow soln$ with v_f flipped
        $cost \leftarrow cost +$ DeltaCost(v_f)
return "Failure, best assignment is", best $soln$ found

We have also used this method to implement a lazy version of MC-SAT that avoids grounding unnecessary atoms and clauses [29].

Lifted Inference The inference methods discussed so far are purely probabilistic in the sense that they propositionalize all atoms and clauses and apply standard probabilistic inference algorithms. A key property of first-order logic is that it allows *lifted* inference, where queries are answered without materializing all the objects in the domain (e.g., resolution [34]). Lifted inference is potentially much more efficient than propositionalized inference, and extending it to probabilistic logical languages is a desirable goal. We have developed a lifted version of loopy belief propagation (BP), building on the work of Jaimovich *et al.* [11]. Jaimovich *et al.* pointed out that, if there is no evidence, BP in probabilistic logical models can be trivially lifted, because all groundings of the same atoms and clauses become indistinguishable. Our approach proceeds by identifying the subsets of atoms and clauses that remain indistinguishable even after evidence is taken into account. We then form a network with *supernodes* and *superfeatures* corresponding to these sets, and apply BP to it. This network can be vastly smaller than the full ground network, with the corresponding efficiency gains. Our algorithm produces the unique minimal lifted network for every inference problem.

We begin with some necessary definitions. These assume the existence of an MLN L, set of constants C, and evidence database E (set of ground literals). For simplicity, our definitions and explanation of the algorithm will assume that each predicate appears at most once in any given MLN clause. We will then describe how to handle multiple occurrences of a predicate in a clause.

Definition 4. *A* supernode *is a set of groundings of a predicate that all send and receive the same messages at each step of belief propagation, given L, C and E. The supernodes of a predicate form a partition of its groundings.*

A superfeature *is a set of groundings of a clause that all send and receive the same messages at each step of belief propagation, given L, C and E. The superfeatures of a clause form a partition of its groundings.*

Definition 5. *A* lifted network *is a factor graph composed of supernodes and superfeatures. The factor corresponding to a superfeature $g(x)$ is $\exp(wg(x))$, where w is the weight of the corresponding first-order clause. A supernode and a superfeature have an edge between them iff some ground atom in the supernode appears in some ground clause in the superfeature. Each edge has a positive integer weight. A* minimal lifted network *is a lifted network with the smallest possible number of supernodes and superfeatures.*

The first step of lifted BP is to construct the minimal lifted network. The size of this network is $O(nm)$, where n is the number of supernodes and m the number of superfeatures. In the best case, the lifted network has the same size as the MLN L; in the worst case, as the ground Markov network $M_{L,C}$.

The second and final step in lifted BP is to apply standard BP to the lifted network, with two changes:

1. The message from supernode x to superfeature f becomes

$$\mu_{f\to x}^{n(f,x)-1} \prod_{h\in nb(x)\setminus\{f\}} \mu_{h\to x}(x)^{n(h,x)}$$

   where $n(h,x)$ is the weight of the edge between h and x.
2. The (unnormalized) marginal of each supernode (and, therefore, of each ground atom in it) is given by $\prod_{h\in nb(x)} \mu_{h\to x}^{n(h,x)}(x)$.

The weight of an edge is the number of identical messages that would be sent from the ground clauses in the superfeature to each ground atom in the supernode if BP was carried out on the ground network. The $n(f,x)-1$ exponent reflects the fact that a variable's message to a factor excludes the factor's message to the variable.

   The lifted network is constructed by (essentially) simulating BP and keeping track of which ground atoms and clauses send the same messages. Initially, the groundings of each predicate fall into three groups: known true, known false and unknown. (One or two of these may be empty.) Each such group constitutes an initial supernode. All groundings of a clause whose atoms have the same combination of truth values (true, false or unknown) now send the same messages to the ground atoms in them. In turn, all ground atoms that receive the same number of messages from the superfeatures they appear in send the same messages, and constitute a new supernode. As the effect of the evidence propagates through the network, finer and finer supernodes and superfeatures are created.

   If a clause involves predicates $R_1, \ldots, R_k$, and $N = (N_1, \ldots, N_k)$ is a corresponding tuple of supernodes, the groundings of the clause generated by N are found by joining $N_1, \ldots, N_k$ (i.e., by forming the Cartesian product of the relations $N_1, \ldots, N_k$, and selecting the tuples in which the corresponding arguments agree with each other, and with any corresponding constants in the first-order clause). Conversely, the groundings of predicate R_i connected to elements of a superfeature F are obtained by projecting F onto the arguments it shares with R_i. Lifted network construction thus proceeds by alternating between two steps:

1. Form superfeatures by doing joins of their supernodes.
2. Form supernodes by projecting superfeatures down to their predicates, and merging atoms with the same projection counts.

Pseudocode for the algorithm is shown in Table 5. The projection counts at convergence are the weights associated with the corresponding edges.

   To handle clauses with multiple occurrences of a predicate, we keep a tuple of edge weights, one for each occurrence of the predicate in the clause. A message is passed for each occurrence of the predicate, with the corresponding edge weight. Similarly, when projecting superfeatures into supernodes, a separate count is maintained for each occurrence, and only tuples with the same counts for all occurrences are merged.

   See Singla and Domingos [40] for additional details, including the proof that this algorithm always creates the minimal lifted network.

Table 5. Lifted network construction algorithm.

---

function LNC(L, C, E)
   inputs: L, a Markov logic network
           C, a set of constants
           E, a set of ground literals
   output: M, a lifted network
for each predicate P
   for each truth value t in {*true, false, unknown*}
     form a supernode containing all groundings of P with truth value t
repeat
   for each clause involving predicates $P_1, \ldots, P_k$
     for each tuple of supernodes $(N_1, \ldots, N_k)$,
       where N_i is a P_i supernode
       form a superfeature F by joining $N_1, \ldots, N_k$
   for each predicate P
     for each superfeature F it appears in
       $S(P, F) \leftarrow$ projection of the tuples in F down to the variables in P
       for each tuple s in $S(P, F)$
         $T(s, F) \leftarrow$ number of F's tuples that were projected into s
     $S(P) \leftarrow \bigcup_F S(P, F)$
     form a new supernode from each set of tuples in $S(P)$ with the
       same $T(s, F)$ counts for all F
until convergence
add all current supernodes and superfeatures to M
for each supernode N and superfeature F in M
   add to M an edge between N and F with weight $T(s, F)$
return M

---

6 Learning

In this section, we discuss methods for automatically learning weights, refining formulas, and constructing new formulas from data.

6.1 Markov Network Learning

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 [26]. Another alternative is iterative scaling [6]. Features can also be learned from data, for example by greedily constructing conjunctions of atomic features [6].

6.2 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. This is essential when the training data is a single network, such as the Web. The gradient of the log-likelihood with respect to the weights is

$$\frac{\partial}{\partial w_i} \log P_w(X{=}x) = n_i(x) - \sum_{x'} P_w(X{=}x') \, n_i(x')$$ (6)

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. In the generative case, even approximating these expectations tends to be prohibitively expensive or inaccurate due to the large state space. Instead, we can maximize the pseudo-likelihood of the data, a widely-used alternative [1]. 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

$$\log P_w^*(X{=}x) = \sum_{l=1}^{n} \log P_w(X_l{=}x_l | MB_x(X_l))$$ (7)

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 [19], pseudo-likelihood yields efficient learning of MLN weights even in domains with millions of ground

atoms [32]. 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.

6.3 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* of Y given X is

$$P(y|x) = \frac{1}{Z_x} \exp \left(\sum_{i \in F_Y} w_i n_i(x, y) \right) = \frac{1}{Z_x} \exp \left(\sum_{j \in G_Y} w_j g_j(x, y) \right) \qquad (8)$$

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 conditional log-likelihood is

$$\frac{\partial}{\partial w_i} \log P_w(y|x) = n_i(x, y) - \sum_{y'} P_w(y'|x) n_i(x, y')$$
$$= n_i(x, y) - E_w[n_i(x, y)] \qquad (9)$$

In the conditional case, we can approximate the expected counts $E_w[n_i(x, y)]$ using either the MAP state (i.e., the most probable state of y given x) or by averaging over several MC-SAT samples. The MAP approximation is inspired by the voted perceptron algorithm proposed by Collins [2] for discriminatively laerning hidden Markov models. We can apply a similar algorithm to MLNs using MaxWalkSAT to find the approximate MAP state, following the approximate gradient for a fixed number of iterations, and averaging the weights across all iterations to combat overfitting [36]. We get the best results, however, by applying a version of the scaled conjugate gradient algorithm [24]. We use a small number of MC-SAT samples to approximate the gradient and Hessian matrix, and use the inverse diagonal Hessian as a preconditioner. See Lowd and Domingos [21] for more details and results.

6.4 Structure Learning and Clustering

The structure of a Markov logic network is the set of formulas or clauses to which we attach weights. While these formulas are often specified by one or more experts, such knowledge is not always accurate or complete. In addition

to learning weights for the provided clauses, we can revise or extend the MLN structure with new clauses learned from data. We can also learn the entire structure from scratch. The problem of discovering MLN structure is closely related to the problem of finding frequent subgraphs in graphs. Intuitively, frequent subgraphs correspond to high-probability patterns in the graph, and an MLN modeling the domain should contain formulas describing them, with the corresponding weights (unless a subgraph's probability is already well predicted by the probabilities of its subcomponents, in which case the latter suffice). More generally, MLN structure learning involves discovering patterns in hypergraphs, in the form of logical rules. The inductive logic programming (ILP) community has developed many methods for this purpose. ILP algorithms typically search for rules that have high accuracy, high coverage, etc. However, since an MLN represents a probability distribution, much better results are obtained by using an evaluation function based on pseudo-likelihood [13]. Log-likelihood or conditional log-likelihood are potentially better evaluation functions, but are much more expensive to compute. In experiments on two real-world datasets, our MLN structure learning algorithm found better MLN rules than the standard ILP algorithms CLAUDIEN [5], FOIL [30], and Aleph [41], and than 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.

Recently, Mihalkova and Mooney [23] 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.

Another key problem in MLN learning is discovering hidden variables (or inventing predicates, in the language of ILP). Link-based clustering is a special case of this, where the hidden variables are the clusters. We have developed a number of approaches for this problem, and for discovering structure over the hidden variables [14–16]. The key idea is to cluster together objects that have similar relations to similar objects, cluster relations that relate similar objects,

and recursively repeat this until convergence. This can be a remarkably effective approach for cleaning up and structuring a large collection of noisy linked data. For example, the SNE algorithm is able to discover thousands of clusters over millions of tuples extracted from the Web and form a semantic network from them in a few hours.

7 Applications

Markov logic has been applied to a wide variety of link mining problems, including link prediction (predicting academic advisors of graduate students [32]), record linkage (matching bibliographic citations [37]), link-based clustering (extracting semantic networks from the Web [15]), and many others. (See the repository of publications on the Alchemy Web site (alchemy.cs.washington.edu) for a partial list.) In this section we will discuss two illustrative examples: collective classification of Web pages and optimizing word of mouth in social networks (a.k.a. viral marketing).

7.1 Collective Classification

Collective classification is the task of inferring labels for a set of objects using their links as well as their attributes. For example, Web pages that link to each other tend to have similar topics. Since the labels now depend on each other, they must be inferred jointly rather than independently. In Markov logic, collective classification models can be specified with just a few formulas and applied using standard Markov logic algorithms. We demonstrate this on WebKB, one of the classic collective classification datasets [3].

WebKB consists of labeled Web pages from the computer science departments of four universities. We used the relational version of the dataset from Craven and Slattery [3], which features 4165 Web pages and 10,935 Web links. Each Web page is marked with one of the following categories: student, faculty, professor, department, research project, course, or other. The goal is to predict these categories from the Web pages' words and links.

We can start with a simple logistic regression model, using only the words on the Web pages:

$$\texttt{PageClass}(\texttt{p}, +\texttt{c})$$
$$\texttt{Has}(\texttt{p}, +\texttt{w}) \Rightarrow \texttt{PageClass}(\texttt{p}, +\texttt{c})$$

The '+' notation is a shorthand for a set of rules with the same structure but different weights: the MLN contains a rule and the corresponding weight for each possible instantiation of the variables with a '+' sign. The first line, therefore, generates a unit clause for each class, capturing the prior distribution over page classes. The second line generates thousands of rules representing the relationship between each word and each class. We can encode the fact that classes are mutually exclusive and exhaustive with a set of hard (infinite-weight) constraints:

$$\texttt{PageClass}(\texttt{p}, +\texttt{c1}) \wedge (+\texttt{c1} \neq +\texttt{c2}) \Rightarrow \neg\texttt{PageClass}(\texttt{p}, +\texttt{c2})$$
$$\exists\texttt{c}\,\texttt{PageClass}(\texttt{p}, \texttt{c})$$

In Alchemy, we can instead state this property of the `PageClass` predicate in its definition using the '!' operator: `PageClass(page, class!)`, where page and class are type names. (In general, the '!' notation signifies that, for each possible combination of values of the arguments without '!', there is exactly one combination of the arguments with '!' for which the predicate is true.)

To turn this multi-class logistic regression into a collective classification model with joint inference, we only need one more formula:

$$\texttt{Linked}(\texttt{u1}, \texttt{u2}) \wedge \texttt{PageClass}(\texttt{+c1}, \texttt{u1}) \wedge \texttt{PageClass}(\texttt{+c2}, \texttt{u2})$$

This says that linked Web pages have related classes.

We performed leave-one-out cross-validation, training these models for 500 iterations of scaled conjugate gradient with a preconditioner. The logistic regression baseline had an accuracy of 70.9%, while the model with joint inference had an accuracy of 76.4%. Markov logic makes it easy to construct additional features as well, such as words on linked pages, anchor text, etc. (See Taskar *et al.* [42] for a similar approach using relational Markov networks.)

7.2 Viral Marketing

Viral marketing is based on the premise that members of a social network influence each other's purchasing decisions. The goal is then to select the best set of people to market to, such that the overall profit is maximized by propagation of influence through the network. Originally formalized by Domingos and Richardson [8], this problem has since received much attention, including both empirical and theoretical results.

A standard dataset in this area is the Epinions web of trust [31]. Epinions.com is a knowledge-sharing Web site that allows users to post and read reviews of products. The "web of trust" is formed by allowing users to maintain a list of peers whose opinions they trust. We used this network, containing 75,888 users and over 500,000 directed edges, in our experiments. With over 75,000 action nodes, this is a very large decision problem, and no general-purpose utility maximization algorithms have previously been applied to it (only domain-specific implementations).

We modeled this problem as an MLDN (Markov logic decision network) using the predicates `Buys(x)` (person x purchases the item), `Trusts(x_1, x_2)` (person x_1 trusts person x_2), and `MarketTo(x)` (x is targeted for marketing). `MarketTo(x)` is an *action predicate*, since it can be manipulated directly, whereas `Buys(x)` and `Trusts(x_1, x_2)` are *state predicates*, since they cannot. The utility function is represented by the unit clauses `Buys(x)` (with positive utility, representing profits from sales) and `MarketTo(x)` (with negative utility, representing the cost of marketing). The topology of the social network is specified by an evidence database of `Trusts(x_1, x_2)` atoms.

The core of the model consists of two formulas:

$$\texttt{Buys}(\texttt{x}_1) \wedge \texttt{Trusts}(\texttt{x}_2, \texttt{x}_1) \Rightarrow \texttt{Buys}(\texttt{x}_2)$$
$$\texttt{MarketTo}(\texttt{+x}) \Rightarrow \texttt{Buys}(\texttt{x})$$

The weight of the first formula represents how strongly x_1 influences x_2, and the weight of the second formula represents how strongly users are influenced by marketing. In addition, the model includes the unit clause Buys(x) with a negative weight, representing the fact that most users do not buy most products. The final model is very similar to that of Domingos and Richardson [8] and yields comparable results, but Markov logic makes it much easier to specify. Unlike previous hand-coded models, our MLDN can be easily extended to incorporate customer and product attributes, purchase history information, multiple types of relationships, products, actors in the network, marketing actions, etc. Doing so is a direction for future work. See Nath and Domingos [25] for additional details.

8   The Alchemy System

The inference and learning algorithms described in the previous sections are publicly available in the open-source Alchemy system [17]. Alchemy makes it possible to define sophisticated probabilistic models over relational domains with a few formulas, learn them from data, and use them for prediction, understanding, etc. From the user's point of view, Alchemy makes it easier and quicker to develop link-mining applications by taking advantage of the Markov logic language and the existing library of algorithms for it. From the researcher's point of view, Alchemy makes it possible to easily integrate new algorithms with a full complement of other algorithms that support them or make use of them, and to make the new algorithms available for a wide variety of applications without having to target each one individually.

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 6 compares Alchemy with Prolog and BUGS [22], one of the most popular toolkits for Bayesian modeling and inference.

Table 6. A comparison of Alchemy, Prolog and BUGS.

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

9 Conclusion and Directions for Future Research

Markov logic offers a simple yet powerful representation for link mining problems. Since it generalizes first-order logic, Markov logic can easily model the full relational structure of link mining problems, including multiple relations and attributes of different types and arities, relational concepts such as transitivity, and background knowledge in first-order logic. And since it generalizes probabilistic graphical models, Markov logic can efficiently represent uncertainty in the attributes, links, cluster memberships, etc. required by most link mining applications.

The specification of standard link mining problems in Markov logic is remarkably compact, and the open-source Alchemy system (available at alchemy.cs.washington.edu) provides a powerful set of algorithms for solving them. We hope that Markov logic and Alchemy will be of use to link mining 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. More details on Markov logic and its applications can be found in Domingos and Lowd [7].

Directions for future research in Markov logic include further increasing the scalability, robustness and ease of use of the algorithms, applying it to new link mining problems, developing new capabilities, etc.

10 Acknowledgements

References

1. J. Besag. Statistical analysis of non-lattice data. *The Statistician*, 24:179–195, 1975.
2. 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.
3. M. Craven and S. Slattery. Relational learning with statistical predicate invention: Better models for hypertext. *Machine Learning*, 43(1/2):97–119, 2001.

4. P. Damien, J. Wakefield, and S. Walker. Gibbs sampling for Bayesian non-conjugate and hierarchical models by auxiliary variables. *Journal of the Royal Statistical Society, Series B*, 61:331–344, 1999.

5. L. De Raedt and L. Dehaspe. Clausal discovery. *Machine Learning*, 26:99–146, 1997.

6. 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.

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

8. P. Domingos and M. Richardson. Mining the network value of customers. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 57–66, San Francisco, CA, 2001. ACM Press.

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

10. W. R. Gilks, S. Richardson, and D. J. Spiegelhalter, editors. *Markov Chain Monte Carlo in Practice.* Chapman and Hall, London, UK, 1996.

11. 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*, pages 191–199, Vancouver, Canada, 2007. AUAI Press.

12. 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.

13. 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.

14. 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.

15. S. Kok and P. Domingos. Extracting semantic networks from text via relational clustering. In *Proceedings of the Nineteenth European Conference on Machine Learning*, pages 624–639, Antwerp, Belgium, 2008. Springer.

16. S. Kok and P. Domingos. Hypergraph lifting for structure learning in Markov logic networks. In *Proceedings of the Twenty-Sixth International Conference on Machine Learning*, Montreal, Canada, 2009. ACM Press.

17. 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.

18. N. Lavrač and S. Džeroski. *Inductive Logic Programming: Techniques and Applications.* Ellis Horwood, Chichester, UK, 1994.

19. D. C. Liu and J. Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45(3):503–528, 1989.

20. J. W. Lloyd. *Foundations of Logic Programming.* Springer, Berlin, Germany, 1987.

21. 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.

22. 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.

23. 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.

24. M. Møller. A scaled conjugate gradient algorithm for fast supervised learning. *Neural Networks*, 6:525–533, 1993.

25. A. Nath and P. Domingos. A language for relational decision theory. In *Proceedings of the International Workshop on Statistical Relational Learning*, Leuven, Belgium, 2009.

26. J. Nocedal and S. Wright. *Numerical Optimization*. Springer, New York, NY, 2006.

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

28. 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.

29. H. Poon, P. Domingos, and M. Sumner. A general method for reducing the complexity of relational inference and its application to MCMC. In *Proceedings of the Twenty-Third National Conference on Artificial Intelligence*, pages 1075–1080, Chicago, IL, 2008. AAAI Press.

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

31. M. Richardson and P. Domingos. Mining knowledge-sharing sites for viral marketing. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 61–70, Edmonton, Canada, 2002. ACM Press.

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

33. S. Riedel. Improving the accuracy and efficiency of MAP inference for Markov logic. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence*, pages 468–475, Helsinki, Finland, 2008. AUAI Press.

34. J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

35. D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82:273–302, 1996.

36. 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.

37. 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.

38. P. Singla and P. Domingos. Memory-efficient inference in relational domains. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, pages 488–493, Boston, MA, 2006. AAAI Press.

39. 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.

40. P. Singla and P. Domingos. Lifted first-order belief propagation. In *Proceedings of the Twenty-Third National Conference on Artificial Intelligence*, pages 1094–1099, Chicago, IL, 2008. AAAI Press.

41. A. Srinivasan. The Aleph manual. Technical report, Computing Laboratory, Oxford University, 2000.

42. B. Taskar, P. Abbeel, and D. Koller. Discriminative probabilistic models for relational data. In *Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence*, pages 485–492, Edmonton, Canada, 2002. Morgan Kaufmann.

43. S. Wasserman and K. Faust. *Social Network Analysis: Methods and Applications*. Cambridge University Press, Cambridge, UK, 1994.

44. W. Wei, J. Erenrich, and B. Selman. Towards efficient sampling: Exploiting random walk strategies. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence*, pages 670–676, San Jose, CA, 2004. AAAI Press.

45. M. Wellman, J. S. Breese, and R. P. Goldman. From knowledge bases to decision models. *Knowledge Engineering Review*, 7:35–53, 1992.

46. 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.