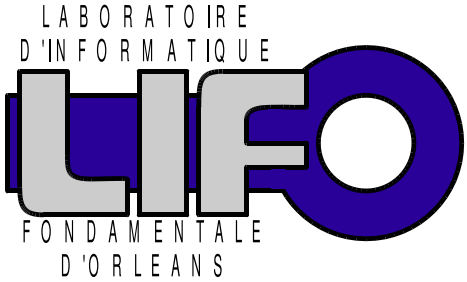

4 rue Léonard de Vinci
BP 6759
F-45067 Orléans Cedex 2
FRANCE
http://www.univ-orleans.fr/lifo

Rapport de Recherche

Discriminative Markov Logic Network Structure Learning based on Propositionalization and χ^2-test

Quang-Thang DINH, Matthieu EXBRAYAT, Christel VRAIN
LIFO, Université d'Orléans

Rapport n^o **RR-2010-03**

Discriminative Markov Logic Network Structure Learning based on Propositionalization and χ^2-test

Quang-Thang DINH, Matthieu EXBRAYAT, Christel VRAIN

LIFO,Bat. 3IA, Université d'Orléans
Rue Lonard de Vinci, B.P. 6759, F-45067 ORLEANS Cedex 2, France

Abstract. In this paper we present a bottom-up discriminative algorithm to automatically learn Markov Logic Network structures. Our approach relies on a new propositionalization method that transforms the learning dataset into an approximative representation in the form of a boolean table. Using this table, the algorithm constructs a set of candidate clauses according to a χ^2 independence test. To compute and choose clauses, we successively use two different optimization criteria, namely log-likelihood (LL) and conditional log-likelihood (CLL), in order to combine the efficiency of LL optimization algorithms together with the accuracy of CLL ones. First experiments show that our approach outperforms existing discriminative MLN structure learning algorithms.

1 Introduction

Inductive Logic Programming (*ILP*) is a research field at the intersection of machine learning and logic programming [26]. It aims at a formal framework as well as practical algorithms for inductively learning relational descriptions from examples and background knowledge. Propositionalization is the process of generating a number of useful attributes or features starting from relational representations and then using traditional propositional algorithms for learning and mining [27]. In the past few years, many approaches to propositionalization have been developed. The majority of these directly transform a relational description into an attribute-value one, though some also consider the intermediate level of multi-instance descriptions [27]. One advantage of such propositionalization is that the whole set of traditional learning algorithms, including neural networks, statistics, support vector machines and so on, can be applied to the approximation. The disadvantage is that the approximation might be incomplete and that some information might get lost in the propositionalization process.

Statistical relational learning (*SRL*) concerns the induction of probabilistic knowledge for multi-relational structured data [28]. Markov Logic Networks (*MLNs*) [18] are a recently developed *SRL* model that generalizes both full first-order logic and Markov Networks [22]. A Markov Network (*MN*) is a graph, where each vertex corresponds to a random variable. Each edge indicates that two variables are conditionally dependent. Each clique of this graph is associated to a weight. In the case of boolean random variables, this weight is a real number, the value of which is directly log-proportional to the probability of the clique (i.e. the conjunction of its random variables) to be true. A Markov Logic Network consists of a set of pairs (F_i,w_i), where F_i is a formula in First Order Logic (*FOL*), to which a weight w_i is associated. The higher w_i, the more likely a grounding of F_i to be true. Given a *MLN* and a set of constants $C = \{c_1, c_2, \ldots c_{|C|}\}$, a *MN* can be generated. The nodes (vertices) of this *MN* correspond to all ground predicates that can be generated by grounding any formula F_i with constants of C. This set can be restricted when constants and variables are typed.

Both generative and discriminative learning can be applied to *MLNs*. In this paper, we propose a propositionalization-based discriminative approach in order to learn both the structure and weights of a *MLN*. This approach consists of three main steps. First, we apply a technique, which is intuitively similar to relational pathfinding [29] and relational cliché [30], searching on the training dataset to form a set of essential features (i.e. the relations to the learning predicate in discriminative fashion) and store it into an approximation boolean table for the propositionalization problem. Second, starting from the approximation table, we compose the set of candidate clauses. Finally, clauses are added into the *MLN*.

This paper is organized as follows: in Section 2 we bring back several notions of First Order Logic, then we propose an overview of *MLNs* learning techniques in Section 3. Our approach is presented in Section 4. Section 5 is devoted to experiments. Section 6 is the conclusion of this paper.

2 Notions

Let us recall here some basic notions of First Order Logic which will be used throughout this paper. We consider a function-free first order language composed of a set $\mathcal{P}$ of predicate symbols, a set C of constants and a set of variables.

Definition 1. *An* atom *is an expression $p(t_1, \ldots, t_k)$, where p is a predicate and t_i are either variables or constants.*

Definition 2. *A* literal *is either a positive or a negative atom; it is* a ground literal *(resp.* variable literal*) when it contains no variable (only variables).*

Definition 3. *A* clause *is a disjunction of literals; a* Horn clause *contains at most a positive literal. A* template clause *composes only of positive literals.*

Definition 4. *Two ground atoms (resp. two variable literals) are said to be connected if they share at least one ground term or argument (one variable).*

Definition 5. *A clause (resp. a ground clause) is* connected *when there is an ordering of its literals $L_1 \wedge \ldots \wedge L_p$, such that for each L_j, $j = 2 \ldots p$, there exists a variable (a constant) occurring both in L_j and in L_i, with $i < j$.*

Definition 6. *A* variabilization *of an expression e (either a ground clause or a conjunction of ground clauses), denoted by $var(e)$, is obtained by assigning a new variable to each constant and replacing all its occurrences by this variable.*

3 Learning of Markov Logic Networks

3.1 Generative learning of MLNs

Generative approaches aim at inducing a global organization of the world described by the predicates, and thus optimize the joint probability distribution of all the variables. Concerning *MLNs*, generative approaches optimize log-likelihood or pseudo-log-likelihood as proposed in [18]. Weights might be learnt using iterative scaling [16]. However, using a quasi-Newton optimization such as *L-BFGS* has recently shown to be much faster [19].

Regarding generative structure learning, the first algorithm proposed in [11] initially uses *CLAUDIEN* [5] to learn the clauses of *MLNs* and then learns the weights by maximizing pseudo-likelihood [18]. In [10], the authors propose a method involving either a beam search

or a shortest first search in the space of clauses guided by a weighted pseudo-log-likelihood (*WPLL*) measure. These two systems follow a top-down paradigm where many potential candidate structures are systematically generated without considering data and then evaluated using a statistical measure evaluating fitness to data. In [13], an algorithm called *BUSL* follows a bottom-up approach in order to reduce the search space. This algorithm uses a propositional Markov Network learning method to construct structure networks that guide the construction of candidate clauses [13]. The structure of *BUSL* is composed of three main phases: Propositionalization, Building clauses and Putting clauses into the *MLN*. In the propositionalization phase, BUSL creates a boolean table *MP* for each predicate P in the domain. When building clauses, *BUSL* applies *Grow-Shrink Markov Network (GSMN)* algorithm (Bromberg et al, 2006) on *MP* to find every clique of the network, from which it builds candidate clauses. Finally, *BUSL* considers clauses to put into the *MLN* one-by-one, using *WPLL* measure for choosing clauses and *L-BFGS* algorithm for setting parameters. The most recent proposed algorithms are Iterated Local Search *(ILS)* [1] and Learning via Hyper-graph Lifting *(LHL)* [9]. *ILS* is based on the iterated local search meta-heuristic that explores the space of structures through a biased sampling of the set of local optima. The algorithm focuses the search not on the full space of solutions but on a smaller subspace defined by the solutions that are locally optimal according to the optimization engine. *LHL* is a different approach, that directly utilizes data in order to construct candidates. From the training dataset, *LHL* builds a hyper-graph from which it forms clauses, that are evaluated using *WPLL* [9]. Through experiments, both *ILS* and *LHL* have shown improvement over the state-of-the-art algorithms. Although, as far as we know, there is no direct comparison between *ILS* and *LHL*.

3.2 Discriminative learning of MLNs

Discriminative learning of *MLNs* relies on the optimization of the *CLL* of query given evidence [18]. Let Y and X be the set of query atoms and evidence atoms, the *CLL* of Y given X is: $\log P(Y = y|X = x) = \log \sum_{j=1}^{n} \log P(Y_j = y_j|X = x)$.

First proposals for *MLN* discriminative learning were based on the voted-perceptron algorithm [20]. The second approach, called *Preconditioned SCG (PSCG)*, based on the scaled conjugate gradient *(SCG)* method, is shown to outperform the previous algorithm both in terms of learning time and prediction accuracy [12]. Recently, a new discriminative weight learning method for *MLNs* based on a max-margin framework was proposed achieving higher *F-scores* than the the *PSCG* method [6].

However, all these algorithms only focus on parameter learning, the structure being supposed given by an expert or previously learned. This can lead to suboptimal results when these clauses do not capture the essential dependencies in the domain in order to improve classification accuracy [3]. To the best of our knowledge, there only exists two systems, that learn the structure of MLNs for a discriminative task. One first uses *ALEPH* [21] to learn a large set of potential clauses, then learns the weights and prunes useless clauses guided by weights [7]. The second method, called Iterated Local Search - Discriminative Structure Learning (*ILS-DSL*), chooses the structure by maximizing *CLL* and sets the parameters by maximizing *WPLL* [3]. Iterated Local Search is used for searching candidate clauses and for every candidate structure, the quasi-Newton optimization method *L-BFGS* is used to set weights optimizing *WPLL*.

Algorithm 1 Structure of DMSP

Input: database DB, Markov logic network MLN, query predicate QP, minWeight
Output: Learned MLN
1. Initialization of the set of candidate clauses: $CanClauses = \emptyset$
2. Propositionalization
Form a set of possible literals SL from DB, QP
Build a boolean table, a column for each element of SL, a row for each ground atom
3. Determine dependent literals and build a set STC of template clauses
4. Put clauses into the learned MLN from STC
$Return(MLN)$

4 DMSP

4.1 DMSP Structure

As inputs to our system, a query predicate QP, a positive real number *minWeight* and a database, called DB in the following, defining positive/negative examples are given. A set of clauses defining background knowledge may also be given. We aim at learning a *MLN* that correctly discriminates between true and false groundings of QP.

Algorithm 1 gives the global structure of our method, called *DMSP* (Discriminative MLN Structure learning based on Propositionalization). It can be separated into three steps: Propositionalization, Building a set of candidate clauses and Learning the Markov networks.
• **Propositionalization:** In order to construct an approximation of the database, *DMSP* first forms a set SL of variable literals starting from the learning predicate QP. Then it builds a boolean table, each column corresponding to a variable literal and each row to only one true/false ground atom of QP. We express how the set SL is formed in Subsection 4.2 and describe how entries of the boolean table are filled in Subsection 4.3.
• **Building a set of candidate clauses:** For each literal $L_{QP} \in SL$, the χ^2 test [23] is applied on the boolean table to find a set SDL of variable literals, each of them being dependent on L_{QP}. A set of connected template clauses STC is then built from L_{QP} and every subset $S \subset 2^{SLD}$. Candidate clauses are built from STC by keeping only Horn clauses from any possible combination of variable literals in each connected template clause.
• **Learning the Markov networks:** Each candidate clause is assigned a weight by applying *L-BFGS* weight learning algorithm to learn the weight of a temporary *MLN* composed of the initial clauses and that candidate clause. If the weight of the candidate clause is greater than *minWeight*, the score is measured by computing the *CLL* of the learning predicate given the temporary *MLN* and *DB*. For each connected template clause, *DMSP* keeps at most one Horn clause, which is the one with the highest *CLL* among those having a weight higher than *minWeight*. The final candidate clauses are sorted by increasing number of literals. Candidate clauses having the same number of literals are sorted by decreasing *CLL*. *DMSP* then considers candidate clauses in turn. For each candidate clause c, it uses *L-BFGS* to learn the weights for a *MLN* composed of the initial *MLN* plus the clauses kept at the previous iterations and c. It then computes the current *CLL* measure. Clause c will be added to the current structure whenever the *CLL* measure is improved. If c is not accepted, and if there exists a clause pc in the current structure such that there exists a variable renaming $\theta, pc\theta \subseteq c$. *DMSP* then checks if replacing pc by c allows a higher *CLL*. If it does, pc is replaced by c. Finally, as adding a clause into a *MLN* might drop down the weight of clauses added before, once all the clauses has been considered, *DMSP* tries to prune some clauses of the *MLN*, as was done in [10].

4.2 Generating literals for propositionalization

Let us give here some more definitions, which will be used in this subsection.

Definition 7. *A* g-chain *of ground literals (resp.* v-chain *of variable literals) of length k starting from a ground literal g_1 (variable literal v_1) is an ordered list of* k *ground literals* $< g_1, ..., g_k >$ *(variable* $< v_1, ..., v_k >$*) such that for* $1 < j \leq k$ *the* j*th ground (variable) literal is connected to the* (j-1)*th via a previously unshared constant (variable). It is denoted by g-chain$_k(g_1)$ (v-chain$_k(v_1)$) :* $g\text{-}chain_k(g_1) =< g_1, ..., g_k >$ (v-chain$_k(v_1) =< v_1, ..., v_k >$).

Definition 8. *The link of two connected ground (variable) literals* g *and* s*, denoted by* link(g, s)*, is an ordered list composed of the name of the predicates of* g *and* s *followed by the positions of the shared arguments.*

Example 1. Let *P(a,b)* and *Q(b,a)* be two ground atoms connected by two shared arguments *a* and *b*. Argument *a* occurs respectively at position 1 of P(a,b) and at position 2 of *Q(b,a)* and argument *b* occurs respectively at position 2 of P(a,b) and at position 1 of *Q(b,a)*. We have: *link(P(a,b), Q(b,a)) = {P Q (1 2) (2 1)}*.

Definition 9. *The link of a g-chain* $gc =< g_1, ..., g_k >$ *(resp. v-chain* $vc =< v_1, ..., v_k >$ *) is an ordered list of* $link(g_i, g_{i+1}), 1 \leq i < k$ *(*$link(v_i, v_{i+1}), 1 \leq i < k$*), denoted by* g-link(gc) *(*v-link(vc)*):*
$g\text{-}link(gc) =< link(g_1, g_2)/.../link(g_i, g_{i+1})/.../link(g_{k-1}, g_k) >$
$(v\text{-}link(vc) =< link(v_1, v_2)/.../link(v_i, v_{i+1})/.../link(v_{k-1}, v_k) >)$.

The definitions of *g-chain*, *v-chain* ensure that a *g-chain* or a *v-chain* is also a connected clause.

For each true ground atom *e* of the query predicate *QP* in the database *DB*, we build the set of *g-chains* of length *k* starting from *e*. From them we build the set *SL* of variable literals so that, for each *g-chain*$_k(e)$, there exists at least a *v-chain*$_k(ve)$, where *ve* and *e* are formed from the same predicate, such that there exists an injective substitution $\theta, v\text{-}chain_k(ve)\theta \equiv g - chain_k(e)$.

To find the set *SL* of variable literals, we consider the remark in [24] that two clauses *g* and *s* are equivalent under *OI-subsumption* if and only if *g* and *s* are equal with a renaming of variable. We can find several *g-chains*$_k(e)$, the variabilization of which are equivalent under *OI-subsumption* to a *v-chain*$_k(ve)$, *ve* and *e* are built with the same predicate. In this case, we only keep one variabilization. Moreover, if a *g-link* of some *g-chains*$_k(e)$ is a prefix of another one, it means that there exists at least one *v-chain*$_k(ve)$ and a variable renaming θ such that $g\text{-}chains_k(e)\theta \subseteq v\text{-}chain_k(ve)$, it is no longer considered for variabilizing. During the process of variabilization to form *vc*, we try to reuse variables (also variable literals) which have been used to variabilize previous *v-chains*$_k$ in order to reduce the number of variable literals, hence reduce the search space of the next steps in our method.

Algorithm 2 sketches our idea to build a set *SL* of variable literals given a database *DB*, a query predicate *QP* and a positive integer k (to limit the maximum number of literals per clause). The algorithm considers each true ground atom *tga* of the query predicate *QP* and builds every *g-chain*$_k(tga)$. Function $LinkOf(g\text{-}chain_k(tga))$ performs two operations. First, it creates the *g-link* of *g-chain*$_k(tga)$, a g-chain of length *k* starting from the true ground atom *tga*. We call this g-link *gl*. Second, it checks whether *gl* is already in the set of *g-links SOGL*, containing the *g-links* already built. Variabilizing will occur only if *gl* does not appear in the set *SOGL*. Regarding the variabilization problem, the replacement of constants by variables can be done using various strategies such as *simple variabilization*, *complete*

Algorithm 2 Generating literals (DB, QP, k)

$maxVar = 0$; $mapVar[c_i] = 0, 1 \leq i \leq mc$, where mc is the number of constants.
for each true ground atom tga of QP **do**
  Find every $g - chain_k(tga)$
  if $LinkOf(g - chain_k(tga), SOGL)$ **then**
    $SOGL = SOGL \cup g - link(g - chain_k(tga))$
    $SL = SL \cup Variabilize(g - chain_k(tga), maxVar, mapVar)$
  end if
end for
$Return(SL)$

Algorithm 3 Variabilizing

Input: $g - chain_k(tga) =< g_1(t_1^1, ..., t_{m_1}^1), ..., g_k(t_1^k, ..., t_{m_k}^k) >$ where $t_{m_j}^i$ is the constant at position m_j of ground atom g_i. A maximum number of variable has been used $maxVar$. A list maps constants to its variables $mapVar$ where mapVar[c] = -v implies that the constant c is replaced by variable -v.
Output: $var(g - chain_k(tga))$, maxVar, mapVar.
If $(maxVar = 0)$ then $maxVar = m_1$;
$mapVar[t_i^1] = -i, 1 \leq i \leq m_1$;
for $(i = 2; i \leq k; i++)$ **do**
  for $(j = 1; j \leq m_i; j++)$ **do**
    if $(mapVar[t_j^i] = 0)$ **then**
      $maxVar++$; $mapVar[t_j^i] = -maxVar$;
    end if
  end for
end for
$\theta =< t_1^1/mapVar[t_1^1], ..., t_j^i/mapVar[t_j^i], ..., t_{m_k}^k/mapVar[t_{m_k}^k] >$;
$v - chain = g - chain_k(tga)\theta$;
$RETURN(v - chain, maxVar, mapVar)$;

variabilization, etc. [25]. Here, we use the *simple variabilization strategy* to variabilize each $g - chain_k(tga)$ ensuring that different constants in this $g - chain_k$ are replaced by different variables. In more details, the algorithm uses the same variable literal for all starting true ground atom tga in the process of variabilizing each $g - chain_k(tga)$, and for the others in a g-chain, a new variable is only assigned to the new constant (a constant that has not previously been assigned a variable). Algorithm 3 describes gradually this step.

We detail how to variabilize a g-chain in particular and illustrate step by step the process of generating literals through Example 2 below.

Example 2. Let DB be a database composed of 15 ground atoms as follows:

advBy(ba,ad), stu(ba), prof(ad), pub(t1,ba),
pub(t2,ba), pub(t1,ad), pub(t2,ad), advBy(be,al), advBy(bo,al),
stu(be), prof(al), pub(t3,be), pub(t4,bo), pub(t4,al), pub(t5,al).

Fig. 1. Example of generating literals

Let *k=4, QP={advBy}*. For the sake of simplicity, in this example k will be ommited. Figure 2a shows all possible *g-chains* of true ground atoms *advBy(ba,ad)* and *advBy(be,al)*, Figure 2b exhibits all g-links of *g-chains* shown in Figure 2a and Figure 2c gives variable literals according to the process of variabilization. Corresponding to every *g-chain*, function *LinkOf* creates a g-link. At the beginning, the *g-link* {*advBy stu 1 1*} corresponding to the *g-chain* {*advBy(ba, ad) stu(ba)*} is created. It is the first considered g-chain therefore the g-link is added into the set *SOGL* of g-links and the $g - chain_4$: {*advBy(ba, ad) stu(ba)*} is variabilized to get the set of literals *SL* = {*advBy(-1, -2), stu(-1)*}, where the algorithm uses the minus to denote whose variables and different constants in this g-chain are replaced by different variables, respectively variables *-1, -2* for constants *ba, ad*.

The algorithm next takes into account the *g-chain* {*advBy(ba, ad) pub(t1, ba) pub(t1, ad) prof(ad)*} and creates the g-link *gl*={*advBy pub 1 2/pub pub 1 1/pub prof 2 1*}. Because *gl* is not in the set *SOGL*, *gl* is added into *SOGL* and the *g-chain* is variabilized to get the set of literals *SL*={ *advBy(-1, -2), stu(-1), pub(-3, -1), pub(-3, -2), prof(-2)*}. Considering then the *g-chain* {*advBy(ba, ad) pub(t2, ba) pub(t2, ad) prof(ad)*}, the algorithm also creates the g-link *gl1*={*advBy pub 1 2/pub pub 1 1/pub prof 2 1*} but *gl1* is already present in the set of g-links (*gl1* and *gl* are the same), and then variabilizing for this g-chain is not useful. The three stars sign (***) displayed in Figure 2c suggests that there is no variabilization for the corresponding g-chain. As we can see from Figure 2c, this situation occurs quite frequently in this example database. It must be noted that, in the case of the g-chain {*advBy(be, al) pub(t3, be)* }, the g-link {*advBy pub 1 2*} is included as a prefix of a g-link and thus the algorithm also does not variabilize this g-chain.

Let us consider now the *g-chain* $\{advBy(be, al)\ advBy(bo, al)\ pub(t4, bo)\ pub(t4, al)\}$. The algorithm creates the g-link *gl2*={ *advBy advBy 2 2 /advBy pub 1 2 / pub pub 1 1* }. This g-link is then variabilized because *gl2* has not occurred in the set of g-links. At the beginning of the variabilization step, the variable literal *advBy(-1,-2)* is reused to map the starting ground atom *advBy(be,al)* (as we mentioned before, the algorithm uses the same variable literal for all starting true ground atoms of the query predicate), hence constants *be, al* respectively are mapped to variables *-1, -2*. The two constants *bo* and *t4* are new considering constants, thus they are respectively assigned to new variables *-5* and *-6*. After this process, three new variable literals were created are *advBy(-5,-2)*, *pub(-6,-5)*, *pub(-6,-2)*.

Having repeated this process until the last true ground atom of the query predicate *advBy*, we get the set of 10 variable literals as follows:
SL={ *advBy(-1,-2) stu(-1) pub(-3,-1) pub(-3,-2) prof(-2) pub(-4,-2) pub(-4,-1) adv(-5,-2) pub(-6,-5) pub(-6,-2)* }.

We end this subsection by introducing the following lemma:

Lemma 1. *Let DB, QP, and k respectively be a database, a query predicate and a maximum length. The set SL of variable literals created by Algorithm 2 is the minimum set such that for each ground atom* e *of* QP*, for each* $g - chain_k(e)$*, there always exists at least a variabilization:* $var(g - chain_k(e)) \subseteq SL$.

Proof: Assume that the set SL of variable literals created by Algorithm 2 is not the minimum set. This means that there is a variable literal $vl \in SL$ such that: for each true ground atom e, for each $g - chain_k(e)$, there always exists at least a variabilization $var(g - chain_k(e)) \subseteq SL \backslash vl$. Following the process of variabilization in Algorithm 2, there exists at least some $g\text{-}chain_k(e)$ such that $g\text{-}chain_k(e)$ is variabilized and $vl \in var(g\text{-}chain_k(e))$. The positions of variable literals appearing in $var(g - chain_k(e))$ are fixed. Beside, different variables in $var(g - chain_k(e))$ map to different constants in $g - chain_k(e)$, therefore vl can not be replaced by the other element in SL, so that we can not remove vl from SL.

Algorithm 4 Build propositional task(DB, SL, L_{QP}, k)

Input: database DB, set SL of variable literals, learning variable literal L_{QP}, length of link k
Output: transformed matrix $Matrix$
$Matrix = \emptyset$;
Find the set of v-links: $SVL = \{v - link(v - chain_k(L_{QP}))\}$;
for each true/false ground atom qga of QP **do**
  fillchar($OneRowOfMatrix, 0$);
  Find the set of g-links: $SGL = \{g - link(g - chain_k(qga))\}$;
  for each g-link $gl \in SGL$ **do**
    if $\exists vl \in SVL$ s.t. $gl \equiv vl$ **then**
      Fill $OneRowOfMatrix[L] = 1, \forall L$, L is a variable literal appearing in vl
    end if
  end for
  $Matrix.append(OneRowOfMatrix)$
end for
$Return(Matrix)$

Ground atoms	advBy (-1,-2)	stu (-1)	pub (-3,-1)	pub (-3,-2)	prof (-2)	pub (-4,-2)	pub (-4,-1)	adv (-5,-2)	pub (-6, -5)	pub (-6,-2)
advBy(bo,al)	1	0	1	1	1	1	1	1	1	1
advBy(ba,ad)	1	1	1	1	1	1	1	0	1	1
advBy(be,al)	1	1	1	1	1	1	1	1	1	1
advBy(ba,be)	0	1	1	1	1	1	1	1	0	1
advBy(ad,ba)	0	0	1	1	0	1	1	0	1	1
advBy(ad,be)	0	0	1	1	0	1	1	0	0	1
. . .	...	...	...	...	...	...	...	...	...	...

Table 1. An example of the boolean table

When k tends to infinity, Algorithm 2 tends to generate a minimum set of variable literals which subsumes the whole database under *OI-subsumption*.

4.3 Building the propositional problem

The second step in propositionalization consists in transforming the first order learning problem into a propositional one. We build a boolean table, called *Matrix*, organized as follows: each column corresponds to a variable literal; each row correspond to a true/false ground atom of the query predicate. Matrix[r][c] is true means that there exists at least a v-chain vc containing variable literal at column c, a g-chain gc starting from the ground atom at row r, and a variabilization of gc such that $var(gc) \subseteq vc$.

Algorithm 4 sketches the steps to fill values of entries of table *Matrix*. For each variable literal L_{QP} of the learning predicate QP, DMSP finds the set SVL of v-links of $v - chains_k$ starting from L_{QP}. For each true/false ground atom qga of QP, it finds every g-link gl of $g - chains_k$ starting from qga. If there exists some $vl \in SVL$ such that vl and gl are similar, then for the row of the Matrix corresponding to the ground atom qga, value at every column L is set to true where L is a variable literal occurring in vl.

We illustrate this step by Example 3.

Example 3. Continuing from Example 2, let $L_{QP} = advBy(-1, -2)$. Algorithm 4 first finds the set SVL of v-links of *v-chains* starting from *advBy(-1,-2)*. For instance, for the *v-chain*

$< advBy(-1,-2)\ stu(-1) >$, it creates a v-link *vl1* ={*advBy stu 1 1*}, and for the *v-chain* $< advBy(-1,-2)\ pub(-3,-1)\ pub(-3,-2)\ prof(-2) >$, it creates a v-link *vl2*={*advBy pub 1 2 / pub pub 1 1 / pub prof 2 1*}. Let us consider now the ground atom *advBy(bo,al)*. Because $advBy(bo, al)$ is the true ground atom, value at column *advBy(-1, -2)* and row *advBy(bo,al)* of *Matrix* is *1 (true)*. Algorithm 4 also finds for every *g-link* of *g-chains* starting from *advBy(bo,al)*. For the *g-chain* $< advBy(bo, al)\ pub(t4, bo)\ pub(t4, al)\ prof(al) >$, it creates a g-link *gl*={*advBy pub 1 2 / pub pub 1 1 / pub prof 2 1*}. This *gl* is similar to *vl2*, so that values at columns *advBy(-1,-2)*, *pub(-3,-1)*, *pub(-3,-2)* and *prof(-2)* are filled by 1. There does not exist any *vc = v-chain(advBy(bo,al))* containing a ground atom of predicate *stu* such that *v-link(vc)* is already in SVL, so that value at column *stu(-1)* and row *advBy(bo,al)* is 0. For the false ground atom *advBy(ad,ba)*, the value at column *advBy(-1,-2)* is 0. The algorithm repeats this process until all true/false ground atoms of the query predicate *advBy* have been considered to produce the approximation boolean table.

4.4 Comparing to BUSL

The outline of our method, at a first glance, is similar to the generative structure learning algorithm *BUSL* [13]. Nevertheless, it differs deeply in all three steps: the way propositionalization is performed, the way to compose the set of candidate clauses and the way to put clauses into the learned *MLN*:

• **Propositionalization:** The approximation tables respectively constructed by *BUSL* and our method are different in the meaning of columns, hence in the meaning of values of entries. Each column in the table *MP* of *BUSL* is a *TNode* which can be either a single literal or a conjunction of several literals, while each column in the table *Matrix* of *DMSP* is a variable literal. For instance, starting from the ground atom *student(a)*, knowing *advBy(b,a)* and then *pub(t, b)*, *BUSL* would produce a *TNode t = AdvBy(B,A), Pub(T, B)* while *DMSP* would produce two separated variable literals *l1 = AdvBy(B,A)* and *l2 = Pub(T, B)*. The number of *TNodes* in *BUSL* can be very high, depending on the number of atoms allowed per *TNode*, the size of the database and the links existing between ground atoms. On the contrary, *DMSP* produces just a minimum set of variable literals, enough for reflecting all possible links between ground atoms. For the *r-th* ground atom of learning predicate, *MP[r][t] = true* if and only if the conjunction of the set of literals in t is true, while *Matrix[r][l] = true* if there exists at least a $v\text{-}chain_k$ starting from the *r-th* ground atom and containing l. These differences influence the performance when applying χ^2-test and *GSMN*.

• **Composing set of candidate clauses:** *BUSL* uses *GSMN* to determine edges amongst *TNodes* and composes candidate clauses from cliques of *TNodes*. *DMSP* uses just the χ^2 test in order to get more links amongst variable literals. Moreover, candidate clauses in *BUSL* must contain all the literals appearing in a *TNode*, meaning that, concerning our example, both *AdvBy(B,A)* and *Pub(T, B)* occur together in the clause. This might not be flexible enough as it might occur that a relevant clause contains only one of these two literals.

• **Adding clauses into *MLN*:** *BUSL* uses likelihood for both setting parameters and choosing clauses, this can lead to sub-optimal results given prediction tasks. *DMSP* also sets the parameters by maximum likelihood but chooses clauses by maximizing the *CLL* of the query predicates instead of the joint likelihood of all predicates. The difference is also in the order clauses are taken into account. *BUSL* uses the order of decreasing *WPLL* while *DMSP* uses two orders; first the order of increasing the number of literals per clause and then the order of decreasing *CLL*. The different orders lead to different structures.

5 Experiments

5.1 Datasets

We use three publicly-available datasets [11] called *IMDB*, *UW-CSE* and *CORA* respectively in order of increasing number of constants as well as increasing number of true ground atoms in the dataset. *IMDB* dataset describes a movie domain containing 1540 ground atoms of 10 predicates and 316 constants. In this dataset, we predict the probability of pairs of person occurring in the relation WorkedUnder. *UW-CSE* dataset describes an academic department consisting of 2673 ground atoms of 15 predicates and 1323 constants. We have chosen the discriminative task of predicting who is *advisor* of who. *CORA* dataset is a collection of citations to computer science papers including 70367 true/false ground atoms of 10 predicates and 3079 constants. We learn four discriminative MLNs, respectively according to four predicates: *sameBib, sameTitle, sameAuthor, sameVenue*.

5.2 Systems and Methodology

DMSP is implemented over the Alchemy package [11]. We ourself perform experiments to answer the following questions:
• Does *DMSP* outperform the state-of-the-art discriminative systems? (1)
• Can we compare *DMSP* to the state-of-the-art generative systems? (2)
• Does *DMSP* perform better than *BUSL* in discriminative tests (only for some query predicate) in terms of *CLL* and *AUC* measures? (3)
• What should be done to improve *DMSP*? (4)
We choose three algorithms to compare to *DMSP*, which are *ILS-DSL*, *ISL* and *BUSL*. To answer question 1, we compare *DMSP* to the state-of-the-art discriminative system *ISL-DSL*. To answer question 2, we choose to run the state-of-the-art generative system *ILS* and also refer to the results of LHL published in [9]. For question 3, we configure *BUSL* to run only for single learning predicates. Comparative results will help us to indicate points for question 4.

For all domains, we performed *5-fold cross-validation*. We measured *CLL* and area under the precision-recall curve *(AUC)*. The *CLL* of a query predicate is the average log-probability over all its groundings given evidence. The precision-recall curve is computed by varying the threshold above which a ground atom is predicted to be true. Parameters for *ILS-DSL*,*BUSL* and *ILS* were respectively set as in [3], [13] and [1]. To guarantee the fairness of comparison, we set the maximum number of literals per clause to 5 for all systems as it is shown in [3]. We used the package provided in [4] to compute *AUC*. We ran our tests on a Dual-core AMD 2.4 GHz CPU - 4GB RAM machine.

5.3 Results

We performed inference on the learned *MLN* for each dataset and for each test fold, using *Lazy-MC-SAT* algorithm. *Lazy-MC-SAT* returns the probability for every grounding of the learning predicate on the test fold, which is used to compute the average *CLL* over all the groundings and the relative *AUC*.

Table 2 presents average *CLL*, *AUC* measures for learning predicates over test folds, for all algorithms estimating on three datasets. They are average values of learning predicates; *WorkedUnder* for *IMDB*, *AdvisedBy* for *UW-CSE* and *SameBib*, *SameTitle*, *SameAuthor*, *SameVenue* for *CORA*. It must be noted that, while we used the same parameter setting,

Algorithms →		DMSP		ISL-DSL		ISL		BUSL	
Datasets	Predicates	CLL	AUC	CLL	AUC	CLL	AUC	CLL	AUC
IMDB	WorkedUnder	**-0.022±0.011**	**0.382**	-0.029±0.009	0.311	-0.036±0.010	0.329	-0.325±0.171	0.129
UW-CSE	AdvisedBy	**-0.016±0.014**	**0.264**	-0.028±0.019	0.194	-0.031±0.015	0.187	-0.044±0.015	0.204
CORA	SameBib	**-0.136±0.012**	0.420	-0.141±0.011	**0.461**	-0.173±0.015	0.346	-0.325±0.017	0.229
	SameTitle	**-0.085±0.016**	**0.524**	-0.134±0.015	0.427	-0.144±0.014	0.415	-0.284±0.013	0.418
	SameAuthor	**-0.132±0.015**	**0.549**	-0.188±0.016	0.500	-0.234±0.013	0.369	-0.356±0.013	0.347
	SameVenue	**-0.109±0.011**	0.375	-0.132±0.014	0.237	-0.145±0.014	0.250	-0.383±0.015	**0.476**

Table 2. CLL, AUC measures

our results do slightly differ from the ones in [3]. This comes from the fact that we conducted inference using *Lazy-MC-SAT* instead of *MC-SAT*.

First, comparing *DMSP* and *ILS-DSL*, we can notice that *DMSP* performs better both in terms of *CLL* and *AUC* for all datasets, except the *AUC* value for learning predicate *SameBib* for *CORA* dataset. Since *CLL* determines the quality of the probability predictions output by the algorithm, our algorithm outperforms this state-of-the-art discriminative algorithm in the sense of the ability to predict correctly the query predicates given evidences. Since *AUC* is useful to predict the few positives in the data, we can conclude that *DMSP* enhances the ability of predicting the few positives. This is the answer for question 1.

Second, we take a look at *DMSP* and *ILS*. *DMSP* gets better values in both *CLL* and *AUC* for all predicates for all datasets. Referring to results of *LHL* [9], *DMSP* gets better *CLL* values and slightly worse *AUC* values. However, as described in [9], in the process of evaluation the authors has omitted from datasets several equality predicates and evaluated groundings for two predicates *Actor* and *Director* (IMDB), two predicates *Student* and *Professor* (UW-CSE) together, which is a bit harder than we did. In spite of that, with the better *CLL* values, we believe in the domination of *DMSP* compared to the state-of-the-art generative structure learning for MLNs. This is the answer for question 2.

Third, let us consider the results of *DMSP* and *BUSL*. *DMSP* does improve highly *CLL* values and is almost better in *AUC* values (only smaller for learning predicate *SameVenue* for *Cora* dataset). We can answer for the question 3 that *DMSP* dominates *BUSL* in terms of *CLL* and *DMSP* is competitive to *BUSL* in terms of *AUC*. However, as we mentioned above, *DMSP* and *BUSL* have the differences in the all three steps. From these results, we can not estimate precisely how each step affects the dominant of *DMSP*. We will further investigate in this issue.

Last, let us consider algorithms all together. For all three datasets, *DMSP* achieves the best *CLL* and only gets two smaller *AUC* values for *SameBib* and *SameVenue (CORA)*. It must be noted that, *DMSP* dominates the remainders on *CLL* values not only on averages but also for every test folds for all datasets. It thus offers the best ability to predict correctly the query predicates given evidences. Concerning *AUC*, *DMSP* performs only poorer for learning predicate *SameBib* than *ILS-DSL*, and for learning predicate *SameVenue* than *BUSL*, it enhances ability to predict the few positives in the test dataset. The smaller *AUC* of *DMSP* for predicates *SameBib* and *SameVenue* can be due to the approach *DMSP* optimizes the *CLL* during structure learning that may lead to overfitting.

Regarding consuming-time, *DMSP* runs somewhat faster than *BUSL* but slower than both *ILS* and *ILS-DSL*. We do not present the consuming-time of all algorithms here because, in theory, to set weights for clauses, all algorithms has involved *L-BFGS*, hence the times all depend on the performance of *L-BFGS*. *DMSP* and *ILS-DSL* have to take more time than *ILS* and *BUSL* for inference to compute *CLL* values. In practice, as revealed in [35], the

presence of a challenging clause like $AdvisedBy(s,p) \wedge AdvisedBy(s,q) \rightarrow SamePerson(p,q)$ will have great impact on optimization as well as on inference. Time-consuming therefore depends mostly on the number of candidate clauses and the occurrence of literals together in each clause. From practice we also verify that the time consuming for finding candidate clauses is much less than the time for weights learning and inference. *ILS-DSL* is accelerated by using the same approach as shown in [12] for setting the parameters of *L-BFGS* that optimize the *WPLL*, and using heuristics to make the execution of *Lazy-MC-SAT* tractable in a limited time. This is the reason why *ILS-DSL* is the fastest system, but it is maybe one of reason to make less *CLL* and AUC. *BUSL* reduces consuming-time by considering only cliques in the structure networks. *DMSP* saves time by solving only Horn clauses and involving inference only when the weight of the considering clause is greater than *minWeight*. This is why in this estimation, we only consider the *CLL* and AUC measures in evaluation. We would like to notice that the *MLN* produced by *DMSP* might be an advantage, from the logic point of view, as a Horn-clause *MLN* might integrate easier in further processing than a *MLN* based on arbitrary-clauses.

These comparative results let us believe in the prospect of $DSML$. However, we need to conduct further experiments in order to estimate thoroughly $DMSP$ from which to improve our algorithm. In more details, we plan to do the following tasks:

• Study a strategy to learn *MLN* improving both two measures *CLL* and AUC in a reasonable time. It must also overcome overfitting.

• Compare the influence of two approximation boolean tables to the performance of *DMSP* and *BUSL*, as well as the effect of χ^2-test and *GSMN* on these two tables, from which we can improve our propositionalization, not only for this task but also for *ILP* community.

• Compare *DMSP* directly to *LHL* and also compare all algorithms to richer and more complex domains. It also includes the task of spreading our propositionalization method to a generative fashion, the task of integrating a discriminative weight learning algorithm into *DMSP*, in order to estimate our method more throughly.

6 Conclusion

Contributions presented in this paper are a novel algorithm for the discriminative learning of *MLN* structure in general and a propositionalization method in particular. The discriminative MLN structure learning algorithm performs a bottom-up approach which learns a *MLN* automatically and directly from a training dataset by first building an approximation problem from which it forges candidate clauses. Comparative results show that the proposed algorithm dominates the state-of-the-art *MLN* structure learning algorithms.

References

1. Biba, M., Ferilli, S., Esposito, F.: Structure Learning of Markov Logic Networks through Iterated Local Search. ECAI 2008. IOS Press (2008)
2. Muggleton, S., Building, W., Road, P.: Inverse entailment and Progol. New Generation Computing Journal. Pp 245-286. SpringerLink (1995)
3. Biba, M., Ferilli S., Esposito, F.: Discriminative Structure Learning of Markov Logic Networks. ILP '08. Pp 59–76. Springer-Verlag (2008)
4. Davis, J., Goadrich, M.: The relationship between Precision-Recall and ROC curves. ICML'06. Pp 233–240. ACM (2006)
5. Raedt, D. L., Dehaspe, L.: Clausal Discovery. Mach. Learn. Volume 26. 1997.
6. Huynh, N. T., Mooney, R. J.: Max-Margin Weight Learning for MLNs. ECML 2009.

7. Huynh, N. T., Mooney, R. J.: Discriminative structure and parameter learning for Markov logic networks. ICML '08. Pp 416–423. ACM (2008)
8. Henry, K., Bart, S., Yueyen, J.: A General Stochastic Approach to Solving Problems with Hard and Soft Constraints. Pp 573–586. American Mathematical Society (1996)
9. Kok, S., Domingos, P.: Learning Markov logic network structure via hypergraph lifting. ICML '09. Pp 505–512. ACM (2009)
10. Kok, S., Domingos, P.: Learning the structure of MLNs. ICML '05. ACM (2005)
11. Kok, S., Sumner, M., Richardson, M., Singla, P., Poon, H., Lowd, D., Wang, J., Domingos, P.: The Alchemy system for statistical relational AI. Dept. of Comp. Sci. and Eng., Univ. of Washington. `http://alchemy.cs.washington.edu.` (2009)
12. Lowd, D., Domingos P.: Efficient Weight Learning for MLNs. PKDD 2007. (2007)
13. Mihalkova, L., Mooney, R. J.: Bottom-up learning of MLN structure. ICML 2007.
14. Mihalkova, L., Richardson, M.: Speeding up Inference In Statistical Relational Learning by Clustering Similar Query Literals. ILP-09. (2009)
15. Møller, M. F.: A scaled conjugate gradient algorithm for fast supervised learning. Neural Netw. Volume 6. Pp 525–533. Elsevier Science Ltd (1993)
16. Pietra, S. D., Pietra, V. D., Lafferty, J.: Inducing Features of Random Fields. Carnegie Mellon University (1995)
17. Poon, H., Domingos, P.: Sound and efficient inference with probabilistic and deterministic dependencies. AAAI (2006)
18. Richardson, M., Domingos, P.: Markov logic networks. Mach. Learn. Vol. 62 (2006)
19. Sha, F., Pereira, F.: Shallow parsing with CRFs. NAACL '03. (2003)
20. Singla, P., Domingos, P.: Discriminative training of MLNs. AAAI'05. (2005)
21. Ashwin, S.: The Aleph manual. `http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph`. (2001)
22. Bishop, M. C.: Pattern Recognition and Machine Learning. Springer, 2007.
23. Facundo, B., Margaritis, D., Honavar, V.: Efficient Markov Network Structure Discovery Using Independence Tests. SIAM Data Mining (2006).
24. Raedt, D. L.: Logical and Relational Learning. Springer, 2008.
25. Jorge, A. M. G.: Iterative Induction of Logic Programs - An approach to logic program synthesis from incomplete specifications. PhD thesis. Univ. of Porto. 1998.
26. Kersting, K.: An inductive logic programming approach to SRL. PhD thesis. ISBN: 1-58603-674-2. IOS Press, 2006.
27. Raedt, L. D.: Logical and relational learning. ISBN: 9783540200406. Springer, 2008.
28. Getoor, L., Taskar, B.: Introduction to Statistical Relational Learning. The MIT Press, 2007. ISBN, 0262072882. (2007).
29. Richards, B. L., Mooney, R. J.: Learning Relations by Pathfinding. AAAI 1992.
30. Silverstein, G., Pazzani, M.: Relational clichés: Constraining constructive induction during relational learning. The 8-th International Workshop on ML. (1991).
31. Agresti, A.: Categorical Data Analysis (SE). John Wiley and Sons, Inc. (2002)
32. Poon, H., Domingos, P., Sumner, M.: A general method for reducing the complexity of relational inference and its application to MCMC. AAAI'08. Pp 1075–1080. AAAI Press (2008)
33. Singla, P., Domingos, P.: Memory-efficient inference in relational domains. AAAI'06. (2006)
34. Liu, D. C., Nocedal, J.: On the limited memory BFGS method for large scale optimization. Journal Math. Program., 1989. (1989).
35. Shavlik, J. and Natarajan, S.: Speeding up inference in Markov logic networks by preprocessing to reduce the size of the resulting grounded network. IJCAI 2009.