Heuristic Method for Discriminative Structure Learning of Markov Logic Networks

Quang-Thang DINH
LIFO, Université d'Orléans, France
Email: thang.dinh@univ-orleans.fr

Matthieu EXBRAYAT
LIFO, Université d'Orléans, France
Email: matthieu.exbrayat@univ-orleans.fr

Christel VRAIN
LIFO, Université d'Orléans, France
Email: christel.vrain@univ-orleans.fr

***Abstract*—Markov Logic Networks (MLNs) combine Markov Networks and first-order logic by attaching weights to first-order formulas and viewing them as templates for features of Markov Networks. Learning a MLN can be decomposed into structure learning and weights learning. In this paper, we present a heuristic-based algorithm to learn discriminative MLN structures automatically, directly from a training dataset. The algorithm heuristically transforms the relational dataset into boolean tables from which to build candidate clauses for learning the final MLN. Comparisons to the state-of-the-art structure learning algorithms for *MLNs* in the three real-world domains show that the proposed algorithm outperforms them in terms of the conditional log likelihood *(CLL)*, and the area under the precision-recall curve *(AUC)*.**

***Keywords*-Markov Logic Network, Structure Learning, Discriminative learning, Relational Learning.**

I. Introduction

Markov Logic Networks (MLNs) [11] are a first-order logic extension of Markov Networks (MNs). A Markov Network 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 $C = \{c_1, c_2, \dots c_{|C|}\}$, 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. Regarding MLN weights learning, generative approaches optimize the log-likelihood or the pseudo-log-likelihood (PLL) [11] using the iterative scaling [10] algorithm or a quasi-Newton optimization method such as the *L-BFGS* [12] algorithm. The PLL of the possible worlds x given by $\log P_w(X = x) = \sum_{l=1}^{n} \log P_w(X_l = x_l|MB_x(X_l))$, where X_l is a ground atom, x_l is the truth value (0 or 1), $MB_x(X_l)$ is the state of the Markov blanket of X_l in x. Discriminative approaches rely on the optimization of the conditional log-likelihood (CLL) of query given evidence [11]. Let *Y* be the set of query atoms and *X* be the set of evidence atoms, CLL of *Y* given *X* is: $\log P(Y = y|X = x) = \log \sum_{j=1}^{n} \log P(Y_j = y_j|X = x)$. Methods for MLN discriminative weights learning were based on the voted-perceptron algorithm [13], the scaled conjugate gradient method (PSCG) [8], the max-margin [4]. 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 [2]. This hence requires to learn MLN structure directly from data. Further, MLN structure learning is a very important task because it allows us to discover novel knowledge. However, it is also a challenging one because of its super-exponential search space, hence only a few practical approaches have been proposed to date. They are the top-down approaches (e.g., Kok and Domingos, 2005), the bottom-up BUSL algorithm (e.g., Mihalkova and Mooney, 2007), the ILS (Iterated Local Search) algorithm (e.g., Biba et al. 2008), the LHL (Learning via Hyper-graph Lifting) algorithm (e.g., Kok and Domingos, 2009) for generative structure learning. For discriminative structure learning, to the best of our knowledge, there only exists two systems. The first one uses the *ALEPH* system to learn a large set of potential clauses, then learns the weights and prunes useless clauses [5]. The second method, called Iterated Local Search - Discriminative Structure Learning (ILS-DSL), chooses the structure by maximizing the CLL and sets the parameters by the L-BFGS algorithm maximizing the PLL [2].

In this paper we propose a heuristic approach, in order to learn discriminatively the structure of a MLN. This approach consists of three main steps. First, for each query predicate, the algorithm applies a heuristic technique to build a set of variable literals, then for each variable literal of the query predicate it heuristically transforms the learning dataset to a boolean table. Second, by applying the Grow-Shrink Markov Network (GSMN) [14] algorithm to these boolean tables, the algorithm extracts a set of *template* clauses. We define a template clause as a disjunction of positive variable literals. Last, candidate clauses are built from the template clauses to add into the MLN.

In the following, we present our method in Section II, Section III is devoted to experiments and Section IV is the conclusion of this paper.

II. Proposal of a Heuristic Method

Let us first recall some basic notions of first order logic and make precise the task at hand. 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. An *atom* is an expression $p(t_1, \ldots, t_k)$, where p is a predicate and t_i are either variables or constants. A *literal* is either a positive or a negative atom; it is a *ground literal* when it contains no variable; it is a *variable literal* when it contains all variables. A *clause* is a disjunction of literals; a *Horn clause* contains at most a positive literal. Two ground atoms are said to be connected if they share at least one ground term (or argument). A clause (resp. a ground clause) is *connected* when there is an ordering of its literals $L_1 \vee \ldots \vee L_p$, such that for each L_j, $j = 2 \ldots p$, there exists a variable (resp. a constant) occurring both in L_j and L_i, $i < j$. A *variabilization* of a ground clause e, denoted by $var(e)$, is obtained by assigning a new variable to each constant and replacing all its occurrences accordingly.

We have as inputs a database DB composed of true/false ground atoms and a query predicate QP (several query predicates can be 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. We describe in this section our algorithm for Heuristic Discriminative Structure learning for *MLNs*, called *HDSM*.

In a MLN, if two variable literals L_i and L_j occur in a same clause, then they are conditionally dependent and L_i must be in the Markov blanket of L_j (or L_i is one of the neighbors of L_j), and vice versa. As a basis of our algorithm, a MLN is built from the training dataset by first forming a set of possible variable literals, and then finding links among them (in the discriminative fashion, it means finding neighbors of each variable literal of the query predicate QP). Template clauses are generated from each query variable literal and its neighbors. Candidate clauses will be extracted from template clauses. It is obvious that the set of possible variable literals is as small as possible to save time for constructing candidate clauses. However, this set is also large enough to be able to describe relations of ground atoms in the dataset as well as to compose good clauses.

We sketch the global structure of HDSM in Algorithm 1. HDSM tries to find existing clauses containing query predicate QP by building a set SL of variable literals, then forming template clauses from SL, each of them has at least an occurrence of QP. To build the set SL of variable literals, HDSM constructs the largest possible set of connected ground atoms corresponding to every true ground atom of QP, then heuristically variabilizes them. We describe the way this set SL is built in Subsection II-A. For each literal $L_{QP} \in SL$, HDSM generates a set of template clauses from which it extracts a set of relevant candidate clauses. A template clause is built from the variable literal L_{QP} and its *neighbors*. Once every predicate has been considered we get a set of template clauses STC. To find the neighbors of the variable literal L_{QP}, HDSM correspondingly transforms the relational dataset into a boolean table in order to apply the GSMN algorithm. We present techniques to build the boolean table in Subsection II-B and detail how template clauses are composed in Subsection II-C. In Subsection II-D we present how the set STC can be used to learn the MLN.

Algorithm 1 HDSM(DB, QP, MLN)

```
Set of template clauses $STC = \emptyset$
for each query predicate $QP$ do
  Heuristically form a set $SL$ of possible variable literals
  for each variable literal $L_{QP} \in SL$ do
    Build a boolean table $BT(L_{QP})$
    Create Template Clauses $\rightarrow$ STC
  end for
end for
Add clauses from $STC$ into $MLN$
$Return(MLN)$
```

Algorithm 2 Form literals (DB, QP)

```
$i = -1$; $mI$ = the number of true ground atoms of QP
for each true ground atom $tga$ of $QP$ do
  $i = i + 1$; $Chains[i] = MakeChain(tga)$
end for
Sort $Chains$ by decreasing $width$
$SL = Variablize(Chains[0])$
$SL = SL \cup CaptureLiteral(Chains[j])$, $1 \leq j \leq mI$
$Return(SL)$
```

We must emphasize that our approach is, at a first glance, somewhat similar to the principle underlying the BUSL [9] algorithm. Both of them consist of three mains steps: Transforming the relational dataset into the boolean tables, building candidate clauses using these boolean tables and putting clauses into the MLN. Methods using in the first phase can be viewed as different propositionalization approaches [15] in *ILP*. As it is shown in [15], they are a kind of incomplete reduction, hence the quality of the boolean table affects the results of the next steps of both approaches. Our approach differs from BUSL not only in the first step but also in the remaining ones. In Subsection II-E we will then discuss these differences in more details.

A. Forming Variable Literals

Finding variable literals is difficult since the database is only a set of ground literals (no templates are given), hence a heuristic technique is used. Given a training database DB and a predicate QP, Algorithm 2 returns a set SL of variable literals. For each true ground atom *tga* of QP in DB, the set of ground atoms of DB connected to *tga* is built by the function *MakeChain*. Such a set is called a *chain*, the *width* of a chain being the number of ground atoms in it. The set SL is then built so that for each chain, there exists a variabilization of it with all variable literals belonging to SL. It must be noted that the function *MakeChain* will stop whenever all the ground atoms connected to *tga* are already in the chain, hence it does not take so much time even when the dataset is large.

Regarding this variabilization problem, the replacement of constants by variables can be achieved using various strategies such as *simple variabilization*, *complete variabilization*, etc. Here, we use the *simple variabilization strategy* to variabilize each chain ensuring that different constants in a chain are replaced by different variables.

In more details, the algorithm initially variabilizes the first element chains[0](the largest one) using function *Variabilize*. Different constants in chains[0] are replaced with different variables. The resulting variable literals form the basis of the set *SL*. The remaining elements of *chains* are then explored by decreasing *width*. Function *CaptureLiteral* heuristically carries their variabilization ensuring that there exists a variabilization s.t. $var(chain(j)) \subseteq SL$, $1 \leq j \leq mI$, where mI is the number of true ground atoms of QP in the DB. Each new element of *chains* being less wide than the preceding ones, we can reasonably expect that the number of new variables, and thus of variable literals introduced in SL decreases rapidly. In other words, the existing patterns of variable literals already stored in SL are more likely to fit the new ones, hence the algorithm keeps the set SL as small as possible.

B. Building Boolean Table

To facilitate the presentation, we use here concepts of *link*, *g-chain*, and *v-chain*. The link of two atoms *g* and *s*, denoted by *link(g,s)*, is a set of its shared arguments. A *g-chain* (*v-chain*) of ground (variable) literals starting from a ground (variable) literal g_1 is an ordered list of the ground (variable) literals $<g_1, ..., g_k, ...>$ such that $\forall i > 1$, $link(g_{i-1}, g_i) \neq \emptyset$ and any shared argument in the $link(g_{i-1}, g_i)$ is not in the $link(g_{j-1}, g_j), 1 < j < i$. The definitions of g-chain and v-chain ensures that a g-chain or a v-chain is also a connected clause. This is related to the relational pathfinding [16] and the relational cliché [17].

The next step in our approach transforms the relational database into a boolean table. For the task of discriminative learning, this boolean table needs to catch as much information related to the query predicate as possible. This boolean table, called *Matrix*, is organized as follows: each column corresponds to a variable literal; each row corresponds to a true/false ground atom of the query predicate. Let us assume that the data concerning a given ground atom q_r of QP is stored in row *r*. Let us assume that column c corresponds to a given variable literal vl_c. *Matrix[r][c]* = true means starting from q_r we can reach a literal that is variabilized as vl_c. In other words, there exists at least a v-chain vc containing the variable literal vl_c, a g-chain gc_r starting from the ground atom q_r, and a variabilization of gc_r such that $vc \subseteq var(gc_r)$.

Let us consider a connected clause $C = A_1 \vee A_2 \vee \cdots \vee A_k$, where the number of literals in *C* is *k*. Since the clause is connected, from any literal $A_i, 1 \leq i \leq k$ we can reach some other literal $A_j, 1 \leq j \leq k, j \neq i$ with at most *k* links. For instance, considering the clause $P(x) \vee !Q(x,y) \vee R(y,z)$, *R(y,z)* can be reached from *P(x)* through two links: *link(P(x),Q(x,y))* and *link(Q(x,y),R(y,z))*. This implies that to find information related to a query variable literal L_{QP} of the query predicate QP, we only need to consider the subset of variable literals appearing in the set $SVC = \{v\text{-}chains(L_{QP})\}$, which is much smaller than the complete set SL, especially when the database is large.

For each variable literal L_{QP} of QP, the algorithm finds the set SVC of $v\text{-}chains(L_{QP})$ from the set SL. Each column of the boolean table *Matrix* corresponds to a precise variable literal appearing in *SVC*. For each true/false ground atom qga of QP, the algorithm fills values for the corresponding row of the boolean table. It will take so much time if we try to find all g-chains starting from qga then check the fitness of each g-chain to every v-chain be cause the number of g-chains is much greater than the number of v-chains. Here, we perform inversely by first finding the set of v-chains then finding g-chains guided by every v-chain. For each $v\text{-}chain(L_{QP})$, the algorithm searches for every *g-chain(qga)* to check whether it fits that $v\text{-}chain(L_{QP})$ (i.e. there exists a variabilization such that $v\text{-}chain(L_{QP}) \subseteq var(g\text{-}chain(qga))$. If it does, value at every column L is set to 1 (*true*) where L is a variable literal occurring in the $v\text{-}chain(L_{QP})$. By this way, for each $v\text{-}chain(L_{QP})$, the algorithm has already known the chain of continuous predicates, thus finding the set of g-chains guided by this $v\text{-}chain(L_{QP})$ is much faster than finding all arbitrary g-chains of qga.

C. Creating Template Clauses

Let us remind that there exists a link between two variable literals if they are conditional dependent. As a consequence, we aim at building the Markov blanket of each query variable literal L_{QP}. For this purpose we propose, as in [9], to use the Grow-Shrink Markov Network algorithm (GSMN) [14], which is based on the Pearson's conditional independence chi-square (χ^2) test to determine whether two variables are conditional independent or not. The algorithm applies GSMN on the boolean table *Matrix* to find the Markov blanket $MB(L_{QP})$ of the query variable literal L_{QP}.

Having got $MB(L_{QP})$, the algorithm composes template clauses. As we have mentioned above, a template clause is simply a disjunction of positive literals. A set of template clauses is created from the query variable literal L_{QP} and its neighbors, i.e. the variable literals forming its Markov blanket $MB(L_{QP})$. Each template clause is built from L_{QP} and a subset $S \subseteq 2^{MB(L_{QP})}$ such that this template clause is also a connected clause.

D. Adding Clauses into the MLN

Candidate clauses are considered in turn. We build a MLN consisting of this clause and the initial MLN given as input (which might be empty). A weight learning algorithm is then applied on the resulting MLN. The score of the weighted MLN is then measured by computing either its CLL or WPLL (depending on the testing purpose) given the database DB. Because every clause created from a template clause composes the similar cliques of the network, for each connected template clause, HDSM only keeps at most one clause, which is the one associated to the MLN having the highest score.

The final candidate clauses are sorted by decreasing score and considered in turn in this order. Each of them is added to the current MLN (for the first clause considered, this MLN is initiated to the input MLN). The weights of the resulting MLN are learned and this latter is scored using the chosen measure. Each clause that improves the score is kept in the current MLN, otherwise it is discarded.

Finally, as adding a clause into the MLN might drop down the weight of clauses added before, once all clauses has been considered, HDSM tries to prune some clauses of the MLN. As was done in [7], this pruning is based on their weight: a clause with a weight less than a given *minWeight* is discarded from the MLN if removing it increases the MLN's score.

E. Discussion

The outline of our method, at a first glance, presents several similarities with the generative structure learning algorithm *BUSL* [9]. Nevertheless, it differs deeply in the ways to create the boolean tables, to compose the set of candidate clauses and to put clauses into the MLN:

Creation of the boolean table: The boolean 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 for a *TNode* which can be either a single literal or a conjunction of several literals, while each column in the table *Matrix* of HDSM for a variable literal. For instance, starting from the ground atom *stu(a)*, knowing *advBy(b,a)* and then *pub(t, b)*, BUSL would produce three TNodes *t1* = {*stu(A)*}, *t2* = {*advBy(B,A)*} and *t3* = {*advBy(C,A), pub(D,C)*}, while HDSM would produce tree separated variable literals *l1* = {*stu(A)*}, *l2* = *advBy(B,A)* and *l3* = *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 amongst ground atoms. On the contrary, HDSM produces a set of variable literals, enough for reflecting all possible links amongst ground atoms. For the *r-th* ground atom of the 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 g-chain starting from the *r-th* ground atom and containing a ground atom of *l*. These differences influence the performance when applying χ^2-test and the GSMN algorithm.

Composition of the set of candidate clauses: BUSL composes candidate clauses from cliques of TNodes hence it could miss some clauses that are not in the cliques. HDSM uses just the MB of the considering variable literal in order to get a little more clauses. Moreover, candidate clauses in BUSL must contain all the literals appearing in a TNode, meaning that, concerning our example, both *advBy(C,A)* and *pub(D,C)* of the TNode *t3* 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. On the contrary, HDSM just composes clauses from variable literals.

Addition of clauses into the *MLN*: For each clique, BUSL creates all possible candidate clauses then removes the duplicated clauses and finally considers them one-by-one to put into the MLN. HDSM just keeps at most one clause for a template clause in the set of candidate clauses.

III. Experiment

A. Datasets

We use three publicly-available datasets [1] 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*.

B. Systems and Methodology

HDSM is implemented over the Alchemy package using the L-BFGS [12] algorithm maximizing PLL to set weights and the CLL measure to choose clauses. We ourself perform experiments to answer the following questions:

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

Table I
CLL, AUC MEASURES

Algorithms →		ISL		BUSL		HDSM-W	
Datasets	Predicates	CLL	AUC	CLL	AUC	CLL	AUC
IMDB	WorkedUnder	-0.036±0.006	0.312	-0.225±0.011	0.129	-0.035±0.007	0.315
UW-CSE	AdvisedBy	-0.031±0.005	0.187	-0.044±0.006	0.204	-0.029±0.008	0.215
CORA	SameBib	-0.173±0.005	0.346	-0.325±0.009	0.229	-0.154±0.011	0.394
	SameTitle	-0.144±0.009	0.415	-0.284±0.009	0.418	-0.127±0.006	0.411
	SameAuthor	-0.234±0.007	0.369	-0.356±0.008	0.347	-0.176±0.007	0.410
	SameVenue	-0.145±0.006	0.427	-0.383±0.010	0.276	-0.121±0.007	0.327

Algorithms →		ILS-DSL		HDSM-H		HDSM-A	
Datasets	Predicates	CLL	AUC	CLL	AUC	CLL	AUC
IMDB	WorkedUnder	-0.029±0.007	0.311	-0.028±0.006	0.323	-0.028±0.008	0.325
UW-CSE	AdvisedBy	-0.028±0.006	0.194	-0.023±0.004	0.231	-0.025±0.010	0.230
CORA	SameBib	-0.141±0.009	0.461	-0.140±0.008	0.480	-0.140±0.011	0.480
	SameTitle	-0.134±0.010	0.427	-0.110±0.007	0.502	-0.108±0.010	0.498
	SameAuthor	-0.188±0.008	0.560	-0.155±0.008	0.581	-0.146±0.009	0.594
	SameVenue	-0.132±0.009	0.297	-0.115±0.009	0.338	-0.115±0.011	0.342

Table II
RUNTIMES(HOUR)

Algorithms →	ILS	ILS-DSL	HDSM-H	HDSM-A	HDSM-W	BUSL
IMDB	0.34	0.49	0.56	1.09	0.42	0.38
UW-CSE	2.28	4.30	7.00	9.30	8.56	8.05
CORA	28.83	30.41	33.71	50.15	38.24	34.52

1) How does HDSM carry out with Horn clauses instead of arbitrary clauses?
2) Is HDSM really better than the state-of-the-art algorithms for discriminative MLN structure learning?
3) Is HDSM really better than the state-of-the-art algorithms for generative MLN structure learning?
4) Is the boolean tables created by HDSM better than the one created by BUSL?

To answer question 1, we configure to run HDSM twice for performing respectively with Horn clauses (HDSM-H) and with arbitrary clauses (HDSM-A). To answer question 2, we compare HDSM to the state-of-the-art discriminative system ILS-DSL [2]. To answer question 3, we choose to run the state-of-the-art generative system ILS [1] and also refer to the results of LHL published in [6]. To answer question 4, we implement HDSM using the L-BFGS algorithm to set weights and the WPLL measure to choose clauses, called HDSM-W. HDSM-W also creates template clauses from cliques and considers all possible clauses from a template clause. We configure BUSL to run only for single predicates. In this case, BUSL and HDSM-W are different only in the step of building boolean tables, hence we can compare the "good" of the boolean tables created by them.

For all the domains, we performed *5-fold cross-validation*. We measured the CLL and the 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 the ILS-DSL and the ILS were respectively set as in [1], [2]. We set the maximum number of literals per clause to 5 for all the systems as it is shown in [2]. We used the package provided in [3] to compute *AUC*. We ran our tests on a Dual-core AMD 2.4 GHz CPU - 4GB RAM machine.

C. Results

We performed inference on the learned MLN for each test fold, for each dataset using the *Lazy-MC-SAT* algorithm. It 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 I presents the average CLL, AUC measures for the learning predicates over test folds for all the algorithms estimating on the three datasets. It must be noted that, while we used the same parameter setting, our results do slightly differ from the ones in [2]. This comes from the fact that we conducted inference using the Lazy-MC-SAT instead of the MC-SAT algorithm, and in the training process ILS-DSL only uses one of the training folds for computing the CLL [2]. Table II exposes the average runtimes over train folds for the datasets IMDB and UW-CSE, over four learning predicates for the CORA dataset.

First, we compare HDSM-H to HDSM-A. We can easy realize that HDSM-A performs only better than HDSM-H in several predicates in terms of both CLL and AUC. HDSM-A is even worse than HDSM-H for the predicate advisedBy (UW-CSE). It is a bit strange because HDSM-A and HDSM-H share the same set of template clauses and HDSM-A considered more clauses than HDSM-H does. We can explain this issue that, in our method, for each template clause the algorithm keeps at most one candidate clause with the highest score, thus the one keeping by HDSM-A with the score greater than or equal the score of the one keeping by HDSM-H. The set of candidate clauses of HDSM-A is hence different to the one of HDSM-H. Therefore, we can conclude that, in the method of HDSM, the order of template clauses and the order of candidate clauses affect the final learned MLN. In addition, when candidate clauses are considered in turn, the set of candidates clauses of HDSM-A is not always better than the one of HDSM-H because the algorithm has to learn the weights again for each considering candidate clause, and it may affect the weights of clauses added before into the MLN. It is interesting that HDSM performing with Horn clauses is much faster than performing with arbitrary clauses while it gets a little loss in the CLL and AUC measures. This is the answer for the question 1.

Second, we compare HDSM to ILS-DSL. Both versions of HDSM perform better than ILS-DSL in terms of both CLL and AUC. 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 HDSM enhances the ability of predicting the few positives in the data. The question 2 is answered.

Third, we compare HDSM to ILS. HDSM gets really better values in both CLL and AUC for all predicates for all datasets. Referring to results of LHL [6], HDSM gets better CLL values and worse AUC values. As described in [6], in the process of evaluation the authors have evaluated groundings for the two predicates *Actor* and *Director* (IMDB), the two predicates *Student* and *Professor* (UW-CSE) together, which is a bit harder than we did. In spite of that, with the better results than ILS and the better CLL values for CORA dataset than LHL, we believe in the domination of our method compared to the state-of-the-art generative structure learning algorithms for MLNs, especially for the task of classification.

Last, we compare all systems together. Regarding runtime, ILS is the fastest system, then are ILS-DSL, HDSM-H, BUSL and HDSM-A. The runtime (of each system) includes the time for finding candidate clauses, the time for learning weight for each candidate clause and the time for choosing clauses for the final MLN. From practice we verify that the time spent for finding candidate clauses is much less than the time for learning weights and for inference to compute the measure (i.e. CLL). To set weights for clauses, all system have involved the L-BFGS algorithm, the runtime thus depends on the performance of this weights learning algorithm. BUSL and HGSM change the MLN completely at each step, thus calculating the WPLL (requires to learn weights by L-BFGS) becomes very expensive. In ILS and ILS-DSL, this does not happen because at each step L-BFGS is initialized with the current weights (and zero weight for a new clause) and it converges in a few iterations [1], [2]. ILS-DSL also use some tactics to ignore a candidate clause whenever it needs so much time for inference [2]. We plan to accelerate our systems in a more reasonable time as it has been done in ILS-DSL, especially, find an effective solution to filter candidate clauses. It is very interesting that HDSM-H takes much less time than HDSM does while it gets only a little loss in the sense of CLL and AUC. We also would like to notice that the MLN produced by HDSM-H 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.

IV. Conclusion and Future Work

Contribution of this paper is an algorithm, HDSM, to learn heuristically, discriminatively the structure of MLNs. Comparative results show that HDSM performing with Horn clauses is really better than performing with arbitrary clauses because of a reasonable time for leaning, a little loss in terms of CLL and AUC measures and the advantages of Horn clauses (in the logic point of view). These comparisons specially gives us belief in developing HDSM performing with Horn clauses as well as in improving performance of HDSM in terms of measures and of execution time. Indeed, considering that candidate clauses were selected due to their maximum CLL, their weights learned by L-BFGS, we will seek further research on a way to associate both these values to choose clauses. Concerning execution, we are thinking of a more integrated organization of the successive steps. We also plan to apply the discriminative weights learning algorithm based on max-margin to evaluate the contribution of the discriminative optimization method in our approach.

References

[1] Biba, M., Ferilli, S., Esposito, F.: Structure Learning of MLNs through Iterated Local Search. ECAI 2008.

[2] Biba, M., Ferilli S., Esposito, F.: Discriminative Structure Learning of MLNs. ILP 2008.

[3] Davis, J., Goadrich, M.: The relationship between Precision-Recall and ROC curves. ICML 2006.

[4] Huynh, N. T., Mooney, R. J.: Max-Margin Weight Learning for MLNs. ECML 2009.

[5] Huynh, N. T., Mooney, R. J.: Discriminative structure and parameter learning for MLNs. ICML 2008.

[6] Kok, S., Domingos, P.: Learning MLN structure via hypergraph lifting. ICML 2009.

[7] Kok, S., Domingos, P.: Learning the structure of MLNs. ICML 2005.

[8] Lowd, D., Domingos P.: Efficient Weight Learning for MLNs. PKDD 2007.

[9] Mihalkova, L., Mooney, R. J.: Bottom-up learning of MLN structure. ICML 2007.

[10] Pietra, S. D., Pietra, V. D., Lafferty, J.: Inducing Features of Random Fields. 1995.

[11] Richardson, M., Domingos, P.: Markov logic networks. Mach. Learn. Vol. 62 (2006).

[12] Sha, F., Pereira, F.: Shallow parsing with CRFs. NAACL 2003.

[13] Singla, P., Domingos, P.: Discriminative training of MLNs. AAAI 2005.

[14] Facundo, B., Margaritis, D., Honavar, V.: Efficient MN Structure Discovery Using Independence Tests. SIAM DM 2006.

[15] Raedt, L. D.: Logical and relational learning. Springer, 2008.

[16] Richards, B. L., Mooney, R. J.: Learning Relations by Pathfinding. AAAI 1992.

[17] Silverstein, G., Pazzani, M.: Relational clichés: Constraining constructive induction during relational learning. The 8-th International Workshop on ML. (1991).