

Generative Structure Learning for Markov Logic Networks

Quang-Thang DINH and Matthieu EXBRAYAT and Christel VRAIN¹

Abstract. In this paper, we present a generative algorithm to learn Markov Logic Network (*MLN*) structures automatically, directly from a training dataset. The algorithm follows a bottom-up approach by first heuristically transforming the training dataset into boolean tables, then creating candidate clauses using these boolean tables and finally choosing the best clauses to build the *MLN*. Comparisons to the state-of-the-art structure learning algorithms for *MLNs* in two 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*).

1 Introduction

In the Machine Learning community, Markov Logic Networks can be viewed as the most recent cross-fertilization of logic and statistical learning. A Markov Logic Network [15] is a structure, which aims at describing a given world with a set of weighted logical formulas.

Let us first remind the definitions of Markov Networks and Markov Logic Networks. A Markov Network (*MN*) is a graphical model, that consists of a set of vertices representing random variables, these vertices being connected by edges. There exists an edge linking two nodes if the corresponding random variables are conditionally dependent. A weight is associated to each clique of this graph. Given the actual value of a subset of the variables, the value of the remaining ones can be inferred using some sampling technique such as a Markov-Chain Monte-Carlo (*MCMC*) algorithm [15].

A Markov Logic Network (*MLN*) L consists of a set of pairs (F_i, w_i) , where F_i is a formula in first order logic to which a weight w_i is given. This latter is a real number, which expresses the “probability” of this formula to be true, in the sense that the higher w_i , the more likely a grounding of F_i to be true. From a *MLN* and a set of constants $C = \{c_1, c_2, \dots, c_{|C|}\}$ a Markov Network can be generated. The nodes (vertices) of this *MN* correspond to all the ground predicates that can be generated with the predicates appearing in the formulas and C . This set can be restricted when constants and variables are typed. Nodes are connected by features, each feature corresponding to the grounding of a formula F_i . A node takes value 1 if the corresponding ground predicate is true, 0 otherwise; a feature takes value 1 when the corresponding ground formula is true, 0 otherwise.

Both generative and discriminative learning can be applied to Markov Logic Networks. In this paper we focus on generative learning. We propose a heuristic method based on propositionalization, that aims at transforming the database into boolean tables expressing the combinations of literals that exist in the training database. From these boolean tables we extract a set of *template* clauses from

which we build the set of candidate clauses to form the *MLN* structure. These candidate clauses are evaluated according to the weighted pseudo-log-likelihood (*WPLL*) measure.

This paper is organized as follows: in Section 2 we propose a state-of-the-art overview of generative learning of *MLNs*. In Section 3 we present our approach and compare it to related work. Section 4 is devoted to experiments and in Section 5, we sum up and propose some extensions to this work.

2 Generative Learning of Markov Logic Networks

Generative approaches try to induce a global organization of the world and thus, in the case of a statistical approach, optimize the joint probability distribution of all the variables. Concerning *MLNs*, generative approaches optimize the log-likelihood or the pseudo-log-likelihood as proposed in [15]. Weights might be learned using the iterative scaling algorithm [11]. However, using a quasi-Newton optimization method such as the *L-BFGS* algorithm [9] has recently showed to be much faster [16].

Regarding generative structure learning, the original algorithm in *Alchemy* [8] consists of two steps. First, it uses the *CLAUDIEN* [12] system to learn a set of clauses. Then it learns the weights by maximizing pseudo-likelihood [15]. In [6], the authors propose a method that combines ideas from the *ILP* [13] and the feature induction of Markov Networks. This algorithm involves a beam or shortest first search in the space of clauses guided by the weighted pseudo-log-likelihood (*WPLL*) [6] 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 of their fitness to the data. In [10], an algorithm called *BUSL* follows a bottom-up approach in order to reduce the search space. The structure of *BUSL* is composed of three main phases. In the first phase, *BUSL* creates a boolean table *MP* for each predicate P in the domain. In the second phase, *BUSL* applies the *Grow-Shrink Markov Network (GSMN)* algorithm (Bromberg et al, 2006) on *MP* to find every clique of the template network structure, from which it builds candidate clauses. Finally, *BUSL* considers candidate clauses to put into the *MLN* one-by-one, using the *WPLL* measure to choose clauses and the *L-BFGS* algorithm to set parameters. The most recent proposed algorithms are the *ILS* (Iterated Local Search) [3] and the *LHL* (Learning via Hyper-graph Lifting) [7]. *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. In more details, it starts by randomly choosing a unit clause CL_C in the search space. Then it performs a greedy local search to efficiently reach a local optimum CL_S . At this point, a perturbation method is applied

¹ LIFO, Université d’Orléans, Rue Léonard de Vinci, B.P. 6759, F-45067 ORLEANS Cedex 2, France, email: {thang.dinh, matthieu.exbrayat, christel.vrain}@univ-orleans.fr

Algorithm 1 Generative Structure Learning (DB, MLN)

```

Set of template clauses  $STC = \emptyset$ 
for each predicate  $QP$  do
  Form a set of possible literals  $SL$  (Algorithm 2)
  for each literal  $L_{QP} \in SL$  do
    Find  $MB(L_{QP})$  (Algorithm 3)
    Create template clauses  $TC$  (Algorithm 4)
     $STC = STC \cup TC$ 
  end for
end for
Add clauses from  $STC$  into the  $MLN$  (Algorithm 5)
Return( $MLN$ )

```

leading to the neighbor CL'_C of CL_S and then a greedy local search is applied to CL'_C to reach another local optimum CL'_S . The algorithm has to decide whether the search must continue from the previous local optimum CL_C or from the last found local optimum CL'_S . *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 the *WPLL* [7]. Through experiments on two real-world domains, 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 Generative Structure Learning

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 $\mathcal{C}$ of constants and a set of variables. An *atom* is an expression $p(t_1, \dots, t_k)$, 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 $L_1 \vee \dots \vee L_p$, such that for each L_j , $j = 2 \dots p$, 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 composed of true/false ground atoms. We are looking for a set of clauses with their weights. We describe in this section our algorithm for Heuristic Generative Structure learning for *MLNs*, called *HGSM*.

In our approach we use the concept of Markov Blanket (*MB*), where the *MB* of a node is the set of all its neighbors. If two literals L_i and L_j occur in a same clause, then they are conditionally dependent and L_i must be in the *MB* of L_j , and vice versa.

As a basis of our algorithm, candidate clauses will be extracted from template clauses. We define a template clause as a disjunction of positive variable literals. This suggests that a *MLN* (or its template) can be built from the training dataset by first forming such sets of possible variable literals, and then finding links among them. Clauses will thus be generated from the template clauses.

We sketch the global structure of *HGSM* in Algorithm 1. *HGSM* tries to find global existing clauses by considering all predicates in the domain in turn. For a given predicate QP , it correspondingly

Algorithm 2 Form literals (DB, QP)

```

index = -1
for each true ground atom  $tga$  of  $QP$  do
  index = index + 1;  $Chains[index] = MakeChain(tga)$ 
end for
Sort  $Chains$  by decreasing length
 $SL = Variabilize(Chains[0])$ 
 $SL = SL \cup CaptureLiteral(Chains[i]), 1 \leq i \leq maxIndex$ 
Return( $SL$ )

```

builds a set SL of variable literals, then it forms template clauses from several subsets of SL , each of them containing at least an occurrence of QP . To build the set SL of variable literals, *HGSM* 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 3.1. For each literal $L_{QP} \in SL$, *HGSM* generates a set of template clauses from which it extracts a set of relevant candidate clauses to add into the *MLN*. 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 . From each template clause, *HGSM* generates all possible candidate clauses (flipping the sign of each literal) and then keeps the best one according to a given measure (i.e. *WPLL*). When every template clause has been considered, we get a set of the best clauses, a subset of which will form the final *MLN*. We present techniques to determine neighbors in Subsection 3.2 and detail how template clauses are composed in Subsection 3.3. In Subsection 3.4 we present how the set STC of template clauses can be used to learn the final *MLN*.

We must emphasize that our approach is, at a first glance, somewhat similar to the principle underlying *BUSL* [10]. Both of them consist of three main steps: Transforming the relational dataset into a boolean table corresponding to each predicate in the domain, Building candidate clauses using these boolean tables and Putting clauses into the *MLN*. Methods used in the first phase can be viewed as different propositionalization approaches [13] in *ILP*. As it is shown in [13], 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 3.5 we will discuss these differences in more detail.

3.1 Forming Variable Literals

We use a heuristic technique to form variable literals described in Algorithm 2. 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. Such a set is called a *chain*, the length 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 .

The longest chain is first chosen and variabilized to initialize the set SL . It is then completed by capturing all the relations of the other shorter chains. By variabilizing chains in order of decreasing length, reasonably and intuitively, we can expect that the algorithm can reuse more variable literals for the shorter chains, hence the number of variable literals created in SL is as small as possible. In the variabilization process, the algorithm ensures that different constants in a chain are replaced by different variables.

In order to illustrate our approach, let us consider a training database consisting of 13 ground atoms as follows:

stu(Bart), *prof*(Ada), *advBy*(Bart,Ada), *pub*(T1,Bart), *pub*(T1,Ada), *pub*(T2,Ada), *pub*(T2,Bart), *stu*(Betty), *prof*(Alan), *advBy*(Betty,Alan), *advBy*(Bob,Alan), *pub*(T3,Betty), *pub*(T4,Alan).

Let $QP = \{\text{advBy}\}$. Let us start from the true ground atom *advBy*(Bart,Ada). The longest chain that can be built starting from this ground atom contains 7 atoms as $\{\text{advBy}(\text{Bart},\text{Ada}), \text{stu}(\text{Bart}), \text{pub}(\text{T1},\text{Bart}), \text{pub}(\text{T2},\text{Bart}), \text{prof}(\text{Ada}), \text{pub}(\text{T1},\text{Ada}), \text{pub}(\text{T2},\text{Ada})\}$.

The algorithm then variabilizes this chain by assigning each constant to a precise variable to get the following set of literals: $SL = \{\text{advBy}(A,B), \text{stu}(A), \text{pub}(C,A), \text{pub}(D,A), \text{prof}(B), \text{pub}(C,B), \text{pub}(D,B)\}$.

Let us consider now the true ground atom *advBy*(Betty,Alan), then we get the chain: $\{\text{advBy}(\text{Betty},\text{Alan}), \text{stu}(\text{Betty}), \text{prof}(\text{Alan}), \text{advBy}(\text{Bob},\text{Alan}), \text{pub}(\text{T3},\text{Betty}), \text{pub}(\text{T4},\text{Alan})\}$. The above set of literals SL is not sufficient to capture the relation among the two ground atoms $\{\text{advBy}(\text{Betty},\text{Alan}), \text{advBy}(\text{Bob},\text{Alan})\}$, and one more literal *advBy*(E,B) is added to SL .

Let us describe this heuristic technique in more detail here. Having got the initialized set SL , the algorithm scans for every shorter chain to capture additional variable literals. During the scan process the algorithm follows precisely the two following principles:

- **Check for a new relation:** It happens when there is a new relation between two ground atoms in the considered chain. For example, the relation between the two ground atoms $\{\text{advBy}(\text{Betty},\text{Alan}), \text{advBy}(\text{Bob},\text{Alan})\}$ is the new one. As the ground atom *advBy*(Betty,Alan) is already variabilized by the variable literal *advBy*(A,B) and variables C,D are already used for other relations in SL , a new variable E is introduced to form a new variable literal *advBy*(E,B) to variabilize the true ground atom *advBy*(Bob,Alan). It guarantees the link between *advBy*(Betty,Alan) and *advBy*(Bob,Alan) by the shared variable B between *advBy*(A,B) and *advBy*(E,B).

- **Check for the existing relations:** It happens when there are several similar relations (which have occurred in some previous considered chain) but there does not exist any variabilization of it. For example, we assume that the dataset in our example contains one more true ground atom *pub*(T5,Alan). The relation between *advBy*(Betty,Alan) and *pub*(T5,Alan) is similar to the relation between *advBy*(Betty,Alan) and *pub*(T4,Alan), both of them sharing the term Alan at the same positions. The set SL is sufficient to variabilize the relations among the three ground atoms *advBy*(Betty,Alan), *pub*(T3,Betty), *pub*(T4,Alan) (respectively by *advBy*(A,B), *pub*(C,A), *pub*(D,B)) but it is not sufficient to variabilize relations among the four ground atoms *advBy*(Betty,Alan), *pub*(T3,Betty), *pub*(T4,Alan), *pub*(T5,Alan). In this case, a new variable literal *pub*(F,B) is used to variabilize *pub*(T5,Alan).

Repeating this process until the last true ground atom of the given predicate *advBy*, algorithm 2 produces the set of 8 variable literals as: $SL = \{\text{advBy}(A,B), \text{stu}(A), \text{pub}(C,A), \text{pub}(D,A), \text{prof}(B), \text{pub}(C,B), \text{pub}(D,B), \text{adv}(E,B)\}$.

3.2 Finding Neighbors of a Variable Literal

The next step of our approach performs the task of finding relevant connections among variable literals in the set SL . For this purpose we find the set of neighbors (Markov Blanket) of a variable literal. For this purpose we propose, as in [10], to use the Grow-Shrink Markov Network algorithm (GSMN) [5]; it is based on the Pearson's conditional independence chi-square (χ^2) statistic test (see [1] for details of its computation) to determine whether two variables are condition-

Algorithm 3 Find Markov Blanket (DB, SL , L_{QP})

```

MatrixForGS =  $\emptyset$ 
Find  $SLPL(L_{QP})$ 
for each true/false ground atom  $qga$  of  $QP$  do
  fillchar(OneRowOfMatrix, 0)
  Find  $Slpga(qga)$ 
  for each  $lpga \in Slpga(qga)$  do
    if  $lpga$  fits some  $LPL \in SLPL(L_{QP})$  then
      for each literal  $L$  in  $LPL$  do
        OneRowOfMatrix[L] = 1
      end for
    end if
  end for
  MatrixForGS.append(OneRowOfMatrix)
end for
GSMN(MatrixForGS)
Return( $MB(L_{QP})$ )

```

ally independent or not.

In order to apply the GSMN algorithm, we first build a boolean table, called *MatrixForGS*. From this boolean table, *GSMN* can easily arrange contingency tables needed to compute the (χ^2) tests. *MatrixForGS* is organized as follows: each column corresponds to a variable literal; each row corresponds to a true/false ground atom of the considered predicate. *MatrixForGS*[r][c] being true means that there exists at least a linked-path vc of connected variable literals containing variable literal at column c, a linked-path of connected ground atom gc starting from the ground atom at row r, and a variabilization of gc such that $var(gc) \subseteq vc$. Here, a linked-path is related to the relational pathfinding [14] and the relational cliché [18]. Figure 1 presents a part of such a boolean table in our example.

Let us consider a connected clause $C = A_1 \vee A_2 \vee \dots \vee A_k$ where k is the maximum number of literals per clause. Since the clause is connected, from some literal A_i we can reach some other literal A_j with at most k links. For example, 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: the link $\{P(x), Q(x,y)\}$ by the term x and the link $\{Q(x,y), R(y,z)\}$ by the term y . This implies that to find links from a given variable literal L_{QP} to the others, we only need to consider the subset of SL that can be reached from L_{QP} with at most k links. We can reasonably expect that this subset is much smaller than SL when the dataset is large, hence the size of the boolean table is reduced considerably. Without considering the time and the space, k can be set as large as needed. Our algorithm uses this value as a parameter to limit the search space. For the sake of simplicity, we henceforth omit this parameter in the presentation.

In the following, a *linked-path* of variable literals denotes a connected clause and $SLPL(L_{QP})$ denotes the set of all linked-paths of variable literals starting from L_{QP} .

Algorithm 3 describes the steps for finding the Markov Blanket of a variable literal L_{QP} . For each true/false ground atom qga of QP , *HGSM* creates a row in the *MatrixForGS*. It builds every linked-path $lpga$ of true ground atoms starting from qga . If $lpga$ satisfies a linked-path of variable literals LPL in $SLPL(L_{QP})$, then for each variable literal L in LPL , value at row qga column L in the *MatrixForGS* is set to *true*. $Slpga(qga)$ denotes the set of all $lpga$. During the process for filling boolean values, the algorithm heuristically, greedily focuses on variable literals that are not yet filled with true values to decide whether they are true or not. Algorithm 3 returns the Markov Blanket of the input literal L_{QP} .

Literals Groundings	AdvBy (A,B)	Stu (A)	Pub (C,A)	Pub (D,A)	Prof (B)	Pub (C,B)	Pub (D,B)	AdvBy (E,B)
advBy(Bart,Ada)	1	1	1	1	1	1	1	0
advBy(Betty,Alan)	1	1	1	1	1	1	1	1
advBy(Bob,Alan)	1	0	0	0	1	1	1	1
advBy(Betty,Bob)	0	1	1	1	0	0	0	0
advBy(Alan,Bart)	0	0	1	1	0	1	1	0
...	...	...	...	...	...	...	...	...

Figure 1. A part of the boolean table

Algorithm 4 Create template clauses (L_{QP} , $MB(L_{QP})$, k)

```

TC = ∅
for j = 2 to k do
  for each subset (j - 1)-elements  $S \subseteq MB(L_{QP})$  do
    if CheckLink( $L_{QP}$ ,  $S$ ) then
       $c = \text{CreateTempClause}(L_{QP}, S)$ 
    end if
     $TC = TC \cup c$ 
  end for
end for
Return(TC)

```

Let us continue our example by starting from the variable literal $advBy(A,B)$. From $advBy(A,B)$, the algorithm can reach any element in the set $\{stu(A), pub(C,A), pub(D,A), prof(B), pub(C,B), pub(D,B), advBy(E,B)\}$. In this case, it is the whole set SL , but as we mentioned above, it could be much smaller than SL when the dataset is large. *MatrixForGS*, each column corresponds to an element in this set, each row corresponds to a true/false ground atom of the predicate $advBy$. Figure 1 shows several rows of the boolean table *MatrixForGS* in our example starting from the variable literal $advBy(A,B)$. We consider, for example, the row corresponding to the false ground atom $advBy(Betty,Bob)$. There exists a linked-path of true ground atoms $\{advBy(Betty,Bob), stu(Betty)\}$ that satisfies the linked-path of variable literals $\{advBy(A,B), stu(A)\}$, and a linked-path of true ground atoms $\{advBy(Betty,Bob), publication(T3,Betty)\}$ that satisfies the two linked-path of variable literals $\{advBy(A,B), pub(C,A)\}$ and $\{advBy(A,B), pub(D,A)\}$. The values at columns corresponding to variable literals $stu(A)$, $pub(C,A)$, $pub(D,A)$ are thus set to *true*. The others are set to *false*.

By applying the *GSMN* algorithm on this boolean table we get the MB of the variable literal $advBy(A,B)$ is $\{stu(A), prof(B), advBy(E,B)\}$.

3.3 Creating Template Clauses

A set of template clauses is created from the starting variable literal L_{QP} and its neighbors, given by its Markov Blanket $MB(L_{QP})$, computed by Algorithm 3. The *Pseudo-code* of this task is detailed in Algorithm 4. At each iteration, by fixing the number of literals per clause j , $2 \leq j \leq k$, where k is the maximum number of variable literals per clause, *HGSM* collects every subset S of $(j - 1)$ -elements ($S \subseteq MB(L_{QP})$) and L_{QP} to form a template clause c . The boolean function $CheckLink(L_{QP}, S)$ ensures that each variable literal in c can reach the other ones. Algorithm 4 returns a set of template clauses TC .

We finish this step by illustrating the set of template clauses created from the starting variable literal $advBy(A,B)$ and its neighbors in

Algorithm 5 Add clauses to the MLN (DB,MLN,STC,modeClause)

```

AddUnitClauseIntoMLN()
LearnWeightsWPLL(MLN, DB)
BestScore = measureWPLL(MLN, DB)
CanClauses = ∅
for each template clause  $Tc \in STC$  do
   $Clauses = \text{CreateClauses}(Tc, modeClause)$ 
  for each clause  $c_i \in Clauses$  do
     $LearnWeightsWPLL(c_i, MLN, DB)$ 
     $c_i.gain = \text{measureWPLL}(c_i, MLN, DB)$ 
  end for
   $CanClauses = CanClauses \cup c$  where  $c.gain = \max\{c_i.gain\}$ 
end for
Choose BestClauses from CanClauses
for each clause  $c$  in BestClauses do
   $LearnWeightsWPLL(c, MLN, DB)$ 
   $NewScore = \text{measureWPLL}(c, MLN, DB)$ 
  if  $NewScore > BestScore$  then
    Add clause  $c$  into MLN
     $BestScore = NewScore$ 
  end if
end for
PruneClauseFromMLN(MLN, DB)
Return(MLN)

```

Subsection 3.2. Among template clauses, we found $advBy(A, B) \vee stu(A)$, $advBy(A, B) \vee prof(B)$, $advBy(A, B) \vee advBy(E, B)$, $advBy(A, B) \vee stu(A) \vee prof(B)$, etc.

3.4 Adding Clauses into the MLN

Algorithm 5 describes the task of putting clauses into the *MLN* from the set of template clauses STC . For each template clause $Tc \in STC$ of length n , *HGSM* generates a set $Clauses$ of clauses by flipping the sign of each variable literal. Depending on the goal (or on time constraints), *HGSM* can restrict the search to *Horn clauses* or consider all *arbitrary* clauses (2^n possible clauses). In [15, 6], the authors have shown that it is useful to start the generative learning of a *MLN* structure by adding all unit clauses. Thus for each clause $c_i \in Clauses$, *HGSM* first learns weights for a *MLN* composed of all unit clauses *plus* c_i , and then computes the *WPLL* of this *MLN*. Then *HGSM* chooses the clause in $Clauses$ that leads to the best *WPLL* and adds it into the set $CanClauses$ of candidate clauses. As proposed in [15, 6], we eliminate several clauses from $CanClauses$ by keeping only clauses improving *WPLL* measures and having a weight greater than a given threshold; this allows time saving during the next steps. The final set of candidate clauses is stored into $BestClauses$.

The learned *MLN* is initialized with the set of unit clauses. Clauses in $BestClauses$ are sorted by decreasing gain, and then checked one-by-one to test whether they can be put into the learned *MLN*: when a clause allows improving the *WPLL* it is added into the *MLN*. As adding a clause into the *MLN* might drop down the weight of the clauses added before, once every clause in $BestClauses$ has been considered, we try to prune some clauses of the *MLN*, as was done in [6].

3.5 Comparing to BUSL

The outline of our method, at a first glance, is similar to the generative structure learning algorithm *BUSL* [10]. Nevertheless, it dif-

fers deeply in all three steps: the way propositionalization is performed, the way to build the set of candidate clauses and the way to put clauses into the learned *MLN*:

- **Propositionalization:** The boolean tables respectively constructed by *BUSL* and *HGSM* are different in the meaning of columns, hence in the meaning of values of entries. Each column in the table *MP* of *BUSL* corresponds to a *TNode* which can be either a single variable literal or a conjunction of several variable literals, while each column in the table *MatrixForGS* of *HGSM* corresponds to 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 *HGSM* 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 between ground atoms. On the contrary, *HGSM* produces just a set of variable literals, enough to reflect all possible links between ground atoms. For the r -th ground atom of the target predicate, $MP[r][t] = true$ if and only if the conjunction of the set of variable literals in t is true, while $MatrixForGS[r][l] = true$ if there exists at least a linked-path of ground atoms starting from the r -th true ground atom and containing a true ground atom of l . These differences influence the performance when applying the *GSMN* algorithm.

- **Composing the set of candidate clauses:** *BUSL* uses *GSMN* to determine edges amongst *TNodes* and composes candidate clauses from cliques of *TNodes*. *HGSM* uses just the MB of the considered 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 *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.

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

4 EXPERIMENTS

4.1 Datasets

We used *UW-CSE* and *CORA*, the two publicly-available datasets from <http://alchemy.cs.washington.edu/>. The *UW-CSE* dataset, prepared by Richardson and Domingos (2006), describes an academic department. The published dataset consists of 15 predicates as *Student(person)*, *Professor(person)*, *AdvisedBy(person, person)*, *Publication(paper, person)*, etc. The dataset contains a total of 2673 true ground atoms with the remainder assumed to be *false*. The *CORA* dataset is a collection of citations of computer science papers, created by Andrew McCallum, and later processed by Singla and Domingos (2006) into 5 folds. The dataset contains a total of 70367 true and false ground atoms with the others assumed to be false.

4.2 Systems and Methodology

HGSM has been implemented on top of the Alchemy package [8]. We used the APIs implementation of *L-BFGS* to learn the maximum *WPLL* weights. We ourselves perform experiments to answer the following questions:

1. Does *HGSM* outperform *BUSL*?
2. Does *HGSM* perform better the state-of-the-art generative systems?
3. How does *HGSM* perform for only Horn clauses?

To answer question 1, we directly compare *HGSM* to *BUSL*. To answer question 2, we compare *HGSM* to the state-of-the-art generative system *ILS* and also refer to the results of *LHL* published in [7]. Finally, for question 3, we configured *HGSM* to limit to Horn clauses, named *HGSM-H*.

For all domains, we performed *5-fold cross-validation*. For each system on each test set, we measured the *CLL* and the area under the precision-recall curve (*AUC*) for every predicate. The advantage of the *CLL* is that it directly measures the quality of the probability estimates produced. The advantage of the *AUC* is that it is insensitive to true negatives (i.e., ground atoms that are false and were predicted to be false). In other words, the *AUC* is useful because it demonstrates how well the algorithm predicts the few positives in the data. The *CLL* of a predicate is the average over all its groundings of the ground atoms log-probability given evidence. The precision-recall curve for a predicate is computed by varying the *CLL* threshold above which a ground atom is predicted to be true. Parameters for *ILS* and *BUSL* were set as in [3]. 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*. For each test set, we ran each system on a Dual-core AMD 2.4 GHz CPU - 4GB RAM machine.

4.3 Results

Having learned the *MLN*, we performed inference for every predicate on the test folds for both datasets, using the *Lazy-MC-SAT* algorithm. *Lazy-MC-SAT* produces probability for every grounding of the query predicate on the test fold. These probability values were used to compute the average *CLL* over all the groundings and the related *AUC*. The *CLL* and *AUC* of a test fold are averaged over all predicates of the domain.

Table 1. CLL, AUC measures

ALGORITHM	UW-CSE		CORA	
	CLL	AUC	CLL	AUC
BUSL	-0.358 ± 0.019	0.291 ± 0.021	-0.341 ± 0.014	0.435 ± 0.010
ILS	-0.180 ± 0.015	0.257 ± 0.019	-0.131 ± 0.011	0.501 ± 0.018
HGSM-H	-0.138 ± 0.017	0.292 ± 0.022	-0.108 ± 0.013	0.701 ± 0.014
HGSM	-0.103 ± 0.020	0.311 ± 0.014	-0.087 ± 0.016	0.762 ± 0.015

Table 1 reports the *CLL* and *AUC* measures for all approaches on both databases. These are the average of *CLLs* and *AUCs* over all test folds. It must be noted that, while we used the same parameter setting, our results do slightly differ from the ones in [3]. This comes from the fact that we conducted inference using the *Lazy-MC-SAT* instead of the *MC-SAT*. Table 2 exposes the average number of candidate clauses (*NOC*) and final learned clauses (*FC*) and the training time (in minutes) (*TT*) over all train folds.

First, let us compare *HGSM* respectively to *BUSL* and *ILS*. *HGSM* outperforms *BUSL* and *ILS* in terms of both *CLL* and *AUC*. We would like to emphasize that *HGSM* dominates them not only on average values, but also for each test fold of each dataset. However, *HGSM*

Table 2. Number of clauses and training time

Alg	UW-CSE			CORA		
	NOC	FC	TT	NOC	FC	TT
BUSL	362.00	37.00	360.85	188.60	17.75	3412.08
ILS	3104.20	23.40	191.84	1755.00	18.25	1730.08
HGSM-H	570.20	23.20	281.43	114.60	17.00	1963.74
HGSM	1520.00	23.60	521.40	143.40	20.25	3849.52

is slower than *BUSL* and *ILS* is the fastest system. We can answer to question 1 that *HGSM* performs better than *BUSL* in the sense of *CLL* and *AUC*. Referring to results of *LHL* [7], *HGSM* gets better *CLL* values and worse *AUC* values for *CORA* dataset. In spite of the lower *AUC*, with the better results than *ILS* and the better *CLL* values than *LHL* for the large dataset *CORA*, 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. This is the answer to question 2. Since the *CLL* determines the quality of the probability predictions output by the algorithm, *HGSM* outperforms *ILS* and *BUSL* in the sense of the ability to predict correctly the query predicates given evidences. Since the *AUC* is insensitive to the large number of true negatives (i.e., ground atoms are false and predicted to be false), *HGSM* enhances the ability to predict the few positives in the data.

Last, let us compare all systems together. *HGSM* is the best system in terms of *CLL* and *AUC* measures, and *ILS* is the best one in terms of runtime. However, in theory, all algorithms have involved the *L-BFGS* to set weights for clauses, hence the times all depend on the performance of the weights learning algorithm. In practice, as revealed in [17], the presence of a challenging clause like $AdvisedBy(s, p) \wedge AdvisedBy(s, q) \rightarrow SamePerson(p, q)$ for the *UW-CSE* dataset will have a great impact on optimization as well as on inference. Runtime 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 used for finding candidate clauses is much less than the time for weight learning and inference. From Table 2 we can see that, although *BUSL* and *HGSM* evaluate fewer candidates than *ILS*, they are both slower than *ILS*. It is because *BUSL* and *HGSM* change the *MLN* completely at each step, calculating the *WPLL* thus becomes very expensive. In *ILS* this does not happen because at each step the *L-BFGS* is initialized with the current weights (and zero weight for a new clause) and it converges in a few iterations [2]. Regarding *HGSM* and *BUSL*, for the *Cora* dataset, despite *HGSM* evaluates fewer candidate clauses than *BUSL*, it gives better *CLL* and *AUC* and unfortunately it runs slower than *BUSL*. This implies that the set of candidate clauses created by *HGSM* is better than the one created by *BUSL*, furthermore our method to create candidate clauses is better than the one in *BUSL*. This issue also urges us to apply a method like it is done in *ILS* to accelerate *HGSM*. It is very interesting that *HGSM-H* takes much less time than *HGSM* does with arbitrary clauses while it also outperforms both *ILS* and *BUSL* in terms of both *CLL* and *AUC*. *HGSM-H* gets only a little loss in the sense of *CLL* and *AUC* compared to *HGSM* with arbitrary clauses. From the logic point of view, a Horn-clause *MLN* might integrate easier in further processing than a *MLN* based on arbitrary-clauses. These results give us belief in restricting our system to solve with *Horn clauses* in a more acceptable runtime. This is the answer to question 3.

5 CONCLUSION

We have presented a heuristic algorithm for the generative learning of *MLN* structure. For each predicate in the domain, the algorithm first heuristically builds a set of variable literals from which to heuristically transform the relational dataset into boolean tables, each of its columns corresponds to a variable literal. Then candidate clauses are built from these boolean tables, which are added into the *MLN* as long as they improve its measure (i.e. *WPLL*). Comparisons show that the proposed algorithm outperforms the state-of-the-art algorithms. From the execution-time point of view, we also noticed that our algorithm runs much faster on Horn clauses than on arbitrary ones, at the price of a small loss in terms of *CLL* and *AUC*. In the future we plan to improve the performance of our algorithm in terms of measures and of execution time. We also plan to apply it to richer and more complex domains and also to compare directly to *LHL*.

ACKNOWLEDGEMENTS

We would like to thank Marenglen Biba for his assistance on ILS and ILS-DSL. We also thank the anonymous reviewers for their comments which helped us to improve this paper considerably.

REFERENCES

- [1] A. Agresti, *Categorical Data Analysis (Second Edition)*, John Wiley and Sons, 2002.
- [2] M. Biba, S. Ferilli, and F. Esposito, ‘Discriminative structure learning of markov logic networks’, in *ILP ’08*, pp. 59–76, (2008).
- [3] M. Biba, S. Ferilli, and F. Esposito, ‘Structure Learning of MLNs through Iterated Local Search’, in *ECAI 2008*, pp. 361–365, (2008).
- [4] J. Davis and M. Goadrich, ‘The relationship between precision-recall and roc curves’, in *ICML’06*, pp. 233–240, (2006).
- [5] B. Facundo, D. Margaritis, and V. Honavar, ‘Efficient markov network structure discovery using independence tests’, pp. 449–484. *SIAM DM*, (2006).
- [6] S. Kok and P. Domingos, ‘Learning the structure of markov logic networks’, in *ICML ’05*, pp. 441–448, (2005).
- [7] S. Kok and P. Domingos, ‘Learning markov logic network structure via hypergraph lifting’, in *ICML ’09*, pp. 505–512, (2009).
- [8] S. Kok, M. Sumner, M. Richardson, P. Singla, H. Poon, D. Lowd, J. Wang, and P. Domingos, ‘The alchemy system for statistical relational ai.’, Technical report, Univ. of Washington., (2009).
- [9] D. C. Liu and J. Nocedal, ‘On the limited memory bfgs method for large scale optimization’, *Math. Program.*, (3), 503–528, (1989).
- [10] L. Mihalkova and R. J. Mooney, ‘Bottom-up learning of markov logic network structure’, in *ICML ’07*, pp. 625–632, (2007).
- [11] S. D. Pietra, V. D. Pietra, and J. Lafferty, ‘Inducing features of random fields’, Technical report, (1995).
- [12] D. L. Raedt and L. Dehaspe, ‘Clausal discovery’, *Mach. Learn.*, 99–146, (1997).
- [13] L. D. Raedt, *Logical and relational learning*, Springer, 2008.
- [14] B. L. Richards and R. J. Mooney, ‘Learning relations by pathfinding’, *AAAI*, (1992).
- [15] M. Richardson and P. Domingos, ‘Markov logic networks’, *Mach. Learn.*, 107–136, (2006).
- [16] F. Sha and F. Pereira, ‘Shallow parsing with conditional random fields’, in *NAACL ’03*, (2003).
- [17] J. Shavlik and S. Natarajan, ‘Speeding up inference in markov logic networks by preprocessing to reduce the size of the resulting grounded network’, *IJCAI*, 1951–1956, (2009).
- [18] G. Silverstein and M. Pazzani, ‘Relational clichés: Constraining constructive induction during relational learning’, *The 8-th International Workshop on ML*, (1991).