Bandit-Based Monte-Carlo Structure Learning of Probabilistic Logic Programs

Nicola Di Mauro[1], Elena Bellodi[2], and Fabrizio Riguzzi[3]

[1] Dipartimento di Informatica – University of Bari "Aldo Moro"
Via Orabona, 4, 70125 Bari, Italy
[2] Dipartimento di Ingegneria – University of Ferrara
Via Saragat 1, I-44122, Ferrara, Italy
[3] Dipartimento di Matematica e Informatica – University of Ferrara
Via Saragat 1, I-44122, Ferrara, Italy
nicola.dimauro@uniba.it, elena.bellodi@unife.it,
fabrizio.riguzzi@unife.it

Abstract. Probabilistic logic programming allows to model domains with complex and uncertain relationships among entities. While the problem of learning the parameters of such programs has been considered by various authors, the problem of learning their structure is yet to be explored in depth. In this work we present an approximate search method based on a one-player game approach, called LEMUR. It relies on the Monte-Carlo tree search UCT algorithm that combines the precision of tree search with the generality of random sampling. LEMUR works by modifying the UCT algorithm in a similar fashion to FUSE, that considers a finite unknown horizon and deals with the problem of having a huge branching factor. The proposed system has been tested on the UW-CSE and Hepatitis datasets and has shown better performances than those of SLIPCASE and LSM.

1 Introduction

Probabilistic Logic Programming (PLP) is gaining popularity due to its ability to represent domains with many entities connected by complex and uncertain relationships. One of the most fertile approaches to PLP is the distribution semantics [1], that is at the basis of several languages such as Probabilistic Logic Programs, Independent Choice Logic, PRISM, Logic Programs with Annotated Disjunctions (LPADs) and ProbLog.

Various algorithms for learning the parameters of probabilistic logic programs under the distribution semantics have been developed, such as PRISM, LFI-ProbLog and EMBLEM. Less systems have been proposed for learning the structure of these programs. Among these, SLIPCASE [2] performs a beam search in the space of possible theories using the log-likelihood (LL) of the examples as the heuristics. The beam is initialized with a number of simple theories that are repeatedly revised using theory revision operators: the addition/removal of a literal from a rule and the addition/removal of a whole rule. Each refinement is

scored by learning the parameters with EMBLEM [3] and using the log-likelihood of the examples returned by it.

Since SLIPCASE search space is extremely large, in this paper we investigate the application of a new approximate search method. In particular, we propose to search in the space of possible theories using a Monte Carlo Tree Search (MCTS) algorithm [4]. MCTS has been originally and extensively applied to Computer Go and recently used in Machine Learning as in FUSE (Feature UCT Selection) [5], that performs feature selection, and BAAL (Bandit-Based Active Learner) [6], that focuses on active learning with small training sets. In this paper, similarly to FUSE, we propose the system LEMUR (*LEarning with a Monte carlo Upgrade of tRee search*) relying on UCT, the tree-structured multi-armed bandit algorithm originally introduced in [7].

We tested LEMUR on a dataset modeling a university domain, UW-CSE, and a dataset modeling a medical domain regarding hepatitis. We compared it with SLIPCASE and LSM, a system for structure learning of Markov Logic Networks [8] and we show that LEMUR achieves higher areas under the Precision Recall and ROC curves.

2 Probabilistic Logic Programming

We present an introduction to PLP focusing on the distribution semantics. We use LPADs as the language for their general syntax.

Logic Programs with Annotated Disjunctions [9] consist of a finite set of annotated disjunctive clauses C_i of the form $h_{i1} : \Pi_{i1}; \ldots; h_{in_i} : \Pi_{in_i} : -b_{i1}, \ldots, b_{im_i}$. $h_{i1}, \ldots h_{in_i}$ are logical atoms and $b_{i1}, \ldots, b_{im_i}$ are logical literals, $\{\Pi_{i1}, \ldots, \Pi_{in_i}\}$ are real numbers in the interval $[0, 1]$ such that $\sum_{k=1}^{n_i} \Pi_{ik} \leq 1$. $b_{i1}, \ldots, b_{im_i}$ is called the *body* and is indicated with $body(C_i)$. Note that if $n_i = 1$ and $\Pi_{i1} = 1$ the clause corresponds to a non-disjunctive clause. If $\sum_{k=1}^{n_i} \Pi_{ik} < 1$ the head of the annotated disjunctive clause implicitly contains an extra atom *null* that does not appear in the body of any clause and whose annotation is $1 - \sum_{k=1}^{n_i} \Pi_{ik}$. We denote by $ground(T)$ the grounding of an LPAD T.

An *atomic choice* is a triple (C_i, θ_j, k) where $C_i \in T$, θ_j is a substitution that grounds C_i and $k \in \{1, \ldots, n_i\}$ identifies the head atom. $C_i\theta_j$ corresponds to a multivalued random variable X_{ij} and an atomic choice (C_i, θ_j, k) to an assignment $X_{ij} = k$. A set of atomic choices κ is *consistent* if only one head is selected from a ground clause. A *composite choice* κ is a consistent set of atomic choices. The *probability* $P(\kappa)$ *of a composite choice* κ is the product of the probabilities of the individual atomic choices, i.e. $P(\kappa) = \prod_{(C_i,\theta_j,k)\in\kappa} \Pi_{ik}$. A *selection* σ is a composite choice that, for each clause $C_i\theta_j$ in $ground(T)$, contains an atomic choice (C_i, θ_j, k). A selection σ identifies a normal logic program w_σ defined as $w_\sigma = \{(h_{ik} \leftarrow body(C_i))\theta_j | (C_i, \theta_j, k) \in \sigma\}$, which is called a *world* of T. Since selections are composite choices, we can assign a probability to worlds: $P(w_\sigma) = P(\sigma) = \prod_{(C_i,\theta_j,k)\in\sigma} \Pi_{ik}$. We denote by S_T the set of all selections and by W_T the set of all worlds of a program T. A composite choice κ identifies a

set of worlds $\omega_\kappa = \{w_\sigma | \sigma \in S_T, \sigma \supseteq \kappa\}$. We define the set of worlds identified by a set of composite choices K as $\omega_K = \bigcup_{\kappa \in K} \omega_\kappa$.

We consider only *sound* LPADs where each possible world has a total well-founded model, so $w_\sigma \models Q$ means a query Q is true in the well-founded model of the program w_σ. Let $P(W)$ be the distribution over the worlds. The probability of a query Q given a world w is $P(Q|w) = 1$ if $w \models Q$ and 0 otherwise. The probability of Q is then:

$$P(Q) = \sum_{w \in W_T} P(Q, w) = \sum_{w \in W_T} P(Q|w)P(w) = \sum_{w \in W_T : w \models Q} P(w) \qquad (1)$$

Example 1. The following LPAD T models the fact that if somebody has the flu and the climate is cold, there is the possibility that an epidemic or a pandemic arises:

$C_1 = epidemic : 0.6; pandemic : 0.3 : -flu(X), cold.$
$C_2 = cold : 0.7.$
$C_3 = flu(david).$
$C_4 = flu(robert).$

T has 18 instances, the query $Q = epidemic$ is true in 5 of them and its probability is $P(epidemic) = 0.6 \cdot 0.6 \cdot 0.7 + 0.6 \cdot 0.3 \cdot 0.7 + 0.6 \cdot 0.1 \cdot 0.7 + 0.3 \cdot 0.6 \cdot 0.7 + 0.1 \cdot 0.6 \cdot 0.7 = 0.588$.

Since it is unfeasible to enumerate all the worlds where Q is entailed, inference algorithms find in practice a covering set of *explanations* for Q, i.e. a set of composite choices K such that Q is entailed in a world w_σ iff $w_\sigma \in \omega_K$. In order to compute $P(Q)$ by a summation as in (1), the explanations have first to be made mutually exclusive with respect to each other. To this purpose Binary Decision Diagrams (BDDs) are used.

EMBLEM [3] performs parameter learning using an Expectation Maximization (EM) algorithm. It takes as input a set of interpretations, i.e., sets of ground facts describing a portion of the domain. The user has to indicate which predicates of the domain are *target*: the corresponding ground atoms will form the examples. For each of these, a BDD encoding its explanations is built. EMBLEM then maximizes the LL for the positive and negative target examples with an EM cycle, until it has reached a local maximum. The E-step computes the expectations of the latent variables directly over BDDs. The M-step updates the parameters for all clauses.

3 LEMUR

MCTS [4] aims at finding optimal decisions in a domain by taking random samples in the decision space and building a search tree in an incremental and asymmetric manner. In each iteration, first a *tree policy* is used in order to find the most urgent node of the tree to expand, trying to balance exploitation and exploration. Then a *simulation* phase is conducted from the selected node, by adding a new child node (obtained with a move from the selected node) and

using a *default policy* that suggests the sequence of actions ("simulation") to be chosen from this new node. Finally, the simulation result is *backpropagated* upwards to update the statistics of the nodes.

The goal of MCTS is to approximate the true values of the moves that may be taken in a given node of the tree. In [7] the authors used the UCT formula in order to implement the tree policy. In particular, the choice of a child node is treated as a multi-armed bandit problem, i.e. the value of a child node is the expected reward approximated by Monte Carlo simulations. Bandit problems are sequential decision problems where the goal is to choose amongst K arms of a multi-armed bandit slot machine in order to maximize the cumulative reward by taking the optimal action, based on past rewards. In the famed UCT MCTS algorithm, a child node j (a machine) is selected to maximize the following UCT formula:

$$v_j = \begin{cases} \overline{X}_j + 2C\sqrt{\frac{2\ln n}{n_j}} & \text{if } n_j > 0 \\ \text{FPU}_j & \text{otherwise} \end{cases} \tag{2}$$

where $\overline{X}_j$ is the average reward from arm j, n is the number of times the current node has been visited, n_j the number of times child j has been visited and $C > 0$ is a constant. When $n_j = 0$, *first-play urgency* (FPU) [10] is used, that assigns a fixed value to unvisited nodes and the UCT value to visited nodes. By tuning the fixed value, early exploitations are encouraged.

When applying the UCT for learning the structure of probabilistic logic programs we consider each logic theory as a bandit problem, where each legal theory revision[4] is an arm with unknown reward.

The tree policy is implemented as follows. During each iteration, LEMUR starts from the root of the tree corresponding to the initial theory. At each node, LEMUR selects one move, corresponding to a possible theory revision, according to the UCT formula. LEMUR then descends to the selected child node and selects a new move until it reaches a leaf. The tree search part ends by creating a new leaf in the tree. Then LEMUR starts the Monte Carlo simulation phase to score the theory at this leaf. One random sequence of revisions is applied starting from the leaf theory until a *finite unknown horizon* is reached: LEMUR stops the simulation after k steps, where k is a uniformly sampled random integer smaller than d and d is an input parameter. Once the horizon is reached, LEMUR produces a reward value Δ.

The nodes visited during this random simulation are not saved, while the nodes visited in the tree policy are saved with their statistics: the visit count n_j, the average reward X_j and the score L_j. n_j and X_j are initially set to 0, while the initial value of the score is obtained by learning the parameters of the corresponding theory with EMBLEM and by using the log-likelihood (LL) of the training examples as L_j.

In the simulation phase, all the visited nodes are scored by computing their LL using EMBLEM as in the tree policy, and the reward Δ corresponds to the maximum score obtained in this random descent. Now, this reward is then

[4] In this paper we consider specializations only as theory revisions.

backpropagated up the sequence of nodes selected for this iteration to update the node statistics: for each node j, its visit count is incremented and its average reward X_j is updated according to Δ. In order to have the values of Δ and thus of X_j within $[0, 1]$, the LL l_j of a node j computed by EMBLEM is normalized as $\Delta = 1/(1 - l_j)$.

UCT does not tell how to choose among nodes that are not explored yet. Typically, UCT visits each unvisited move once before revisiting any. FPU modifies MCTS by assigning a fixed value to unvisited nodes. Instead of fixing the FPU value, we used a different approach in LEMUR: for each unvisited node j ($n_j = 0$), we set its FPU_j value to the mean of the values of the visited sibling nodes.

4 Experimental validation

LEMUR has been tested on two real world datasets: UW-CSE and Hepatitis. The UW-CSE dataset[5] [11] contains information about the Computer Science department of the University of Washington, and is split into five mega-examples, each containing facts for a particular research area. The goal is to predict the target predicate `advisedby(X,Y)`, namely the fact that a person `X` is advised by another person `Y`. The Hepatitis dataset[6] [12] contains information on the laboratory examinations of hepatitis B and C infected patients. The goal is to predict the type of hepatitis of a patient, so the target predicate is `type(patient,type)` where `type` can be `b` or `c`. Positive examples for a type are considered as negative examples for the other type.

LEMUR has been compared to SLIPCASE for learning probabilistic logic programs and to LSM [8] for learning Markov Logic Networks. On both UW-CSE and Hepatitis we applied a five-fold cross-validation. We drew a Precision-Recall curve and a Receiver Operating Characteristics curve and computed the Area Under the Curve (AUCPR and AUCROC respectively) using the methods reported in [13]. The UCT in LEMUR has been iterated 1000 times and its parameters have been set as $C = 0.0001$ and $d = 5$ for UW-CSE, $C = 0.0001$ and $d = 3$ for Hepatitis. Furthermore, LEMUR has been restricted to learn theories with at most 3 rules for UW-CSE and 8 rules for Hepatitis.

Table 1 shows the AUCPR and AUCROC averaged over the folds for all algorithms and datasets. As can be seen, LEMUR achieves higher areas than SLIPCASE and LSM.

References

1. Sato, T.: A statistical learning method for logic programs with distribution semantics. In: International Conference on Logic Programming, MIT Press (1995) 715–729

[5] `http://alchemy.cs.washington.edu/data/uw-cse`
[6] `http://www.cs.sfu.ca/~oschulte/jbn/dataset.html`

System	*Hepatitis*		*UW-CSE*	
	AUCROC	AUCPR	AUCROC	AUCPR
LEMUR	0.76 ± 0.06	0.81 ± 0.04	0.95 ± 0.02	0.28 ± 0.06
SLIPCASE	0.66 ± 0.06	0.71 ± 0.05	0.89 ± 0.03	0.03 ± 0.01
LSM	0.52 ± 0.06	0.53 ± 0.04	0.52 ± 0.06	0.07 ± 0.02

Table 1. Results of the experiments in terms of average Area Under the PR and ROC Curves on the Hepatitis and UW-CSE datasets. The standard deviations are also shown.

2. Bellodi, E., Riguzzi, F.: Learning the structure of probabilistic logic programs. In: Proc. ILP. Volume 7207 of LNCS., Springer (2012) 61–75
3. Bellodi, E., Riguzzi, F.: Expectation Maximization over binary decision diagrams for probabilistic logic programs. Intelligent Data Analysis **17** (2013) 343–363
4. Browne, C., Powley, E.J., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of monte carlo tree search methods. IEEE Trans. Comput. Intellig. and AI in Games **4** (2012) 1–43
5. Gaudel, R., Sebag, M.: Feature selection as a one-player game. In: Proc. 27th Int. Conf. Mach. Learn. (2010) 359–366
6. Rolet, P., Sebag, M., Teytaud, O.: Boosting active learning to optimality: A tractable monte-carlo, billiard-based algorithm. In Buntine, W., Grobelnik, M., Mladeni, D., Shawe-Taylor, J., eds.: Machine Learning and Knowledge Discovery in Databases. Volume 5782 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2009) 302–317
7. Kocsis, L., Szepesvári, C.: Bandit based monte-carlo planning. In: Proceedings of the 17th European conference on Machine Learning. ECML'06, Berlin, Heidelberg, Springer-Verlag (2006) 282–293
8. Kok, S., Domingos, P.: Learning markov logic networks using structural motifs. In Fürnkranz, J., Joachims, T., eds.: ICML, Omnipress (2010) 551–558
9. Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic programs with annotated disjunctions. In: International Conference on Logic Programming. Volume 3131 of LNCS., Springer (2004) 195–209
10. Gelly, S., Wang, Y.: Exploration exploitation in Go: UCT for Monte-Carlo Go. In: NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop. (2006)
11. Kok, S., Domingos, P.: Learning the structure of markov logic networks. In: Proceedings of the 22nd international conference on Machine learning. ICML '05, New York, NY, USA, ACM (2005) 441–448
12. Khosravi, H., Schulte, O., Hu, J., Gao, T.: Learning compact markov logic networks with decision trees. Mach. Learn. **89** (2012) 257–277
13. Davis, J., Goadrich, M.: The relationship between precision-recall and roc curves. In: Proceedings of the 23rd international conference on Machine learning. ICML '06, New York, NY, USA, ACM (2006) 233–240