ECAI 2016
G.A. Kaminka et al. (Eds.)

doi:10.3233/978-1-61499-672-9-1602

Scaling Structure Learning of Probabilistic Logic Programs by MapReduce

Fabrizio Riguzzi[1] and **Elena Bellodi**[2] and **Riccardo Zese**[2]
and **Giuseppe Cota**[2] and **Evelina Lamma**[2]

Abstract. Probabilistic Logic Programming is a promising formalism for dealing with uncertainty. Learning probabilistic logic programs has been receiving an increasing attention in Inductive Logic Programming: for instance, the system SLIPCOVER learns high quality theories in a variety of domains. However, SLIPCOVER is computationally expensive, with a running time of the order of hours. In order to apply SLIPCOVER to Big Data, we present SEMPRE, for "Structure lEarning by MaPREduce", that scales SLIPCOVER by following a MapReduce strategy, directly implemented with the Message Passing Interface.

1 Introduction

Probabilistic Logic Programming (PLP) is an interesting language for Inductive Logic Programming (ILP), because it allows algorithms to better deal with uncertain information. The distribution semantics [5] is an approach to PLP that is particularly attractive for its intuitiveness and for the interpretability of the programs. Various algorithms have been proposed for learning the parameters of probabilistic logic programs under the distribution semantics, such as ProbLog2 [3] and EMBLEM [1]. Recently, systems for learning the structure of these programs have started to appear. Among these, SLIPCOVER [2] performs a beam search in the space of clauses using the log-likelihood as the heuristics.

This system was able to learn good quality solutions in a variety of domains [2] but is usually costly in terms of time.

In this paper, we propose the system SEMPRE for "Structure lEarning by MaPREduce", that is a MapReduce version of SLIPCOVER.

We experimentally evaluated SEMPRE by running it on various datasets using 1, 8, 16 and 32 nodes. The results show that SEMPRE significantly reduces SLIPCOVER running time, even if the speedup is often less than linear because of a (sometimes) relevant overhead.

The paper is organized as follows. Section 2 summarises PLP under the distribution semantics. Section 3 discusses SEMPRE and presents the experiments, while Section 4 concludes the paper.

[1] Dipartimento di Matematica e Informatica – University of Ferrara, Via Saragat 1, I-44122, Ferrara, Italy. Email: fabrizio.riguzzi@unife.it
[2] Dipartimento di Ingegneria – University of Ferrara Via Saragat 1, I-44122, Ferrara, Italy Email: [elena.bellodi, riccardo.zese, giuseppe.cota, evelina.lamma]@unife.it

2 Probabilistic Logic Programming

We introduce PLP focusing on the distribution semantics. We consider Logic Programs with Annotated Disjunctions (LPADs) as the language for their general syntax and we do not allow function symbols; for the treatment of function symbols see [4].

An LPAD is a finite set of annotated disjunctive clauses of the form $h_{i1} : \Pi_{i1}; \ldots; h_{in_i} : \Pi_{in_i} \text{ :- } b_{i1}, \ldots, b_{im_i}.$ where $b_{i1}, \ldots, b_{im_i}$ are literals forming the *body*, $h_{i1}, \ldots h_{in_i}$ are atoms whose disjunction forms the *head* and $\Pi_{i1}, \ldots, \Pi_{in_i}$ are real numbers in the interval $[0, 1]$ s.t. $\sum_{k=1}^{n_i} \Pi_{ik} \leq 1$. If $\sum_{k=1}^{n_i} \Pi_{ik} < 1$, the head contains an extra atom *null* absent from the body of every clause annotated with $1 - \sum_{k=1}^{n_i} \Pi_{ik}$

Given an LPAD P, the grounding $ground(P)$ is obtained by replacing variables with terms from the Herbrand universe in all possible ways. If P does not contain function symbols and P is finite, $ground(P)$ is finite as well. $ground(P)$ is still an LPAD from which we can obtain a normal logic program by selecting a head atom for each ground clause. In this way we obtain a so-called *world* to which we can assign a probability by multiplying the probabilities of all the head atoms chosen. We thus get a probability distribution over worlds from which we can define a probability distribution over the truth values of a ground atom: the probability of an atom q being true is the sum of the probabilities of the worlds where q is true[3].

3 Distributed Structure Learning

SEMPRE parallelizes three operations of the structure learning algorithm SLIPCOVER [2] by employing n workers, one master and $n - 1$ slaves.

The first operation is the scoring of the clause refinements [lines 8-14 in Algorithm 1]: when the revisions for a clause are generated, the master process splits them evenly into n subsets and assigns $n-1$ subsets to the slaves. One subset is handled by the master. Then, SEMPRE enters the *Map phase* [lines 15-25], when each worker scores a set of refinements by means of (serial) EMBLEM [1] which is run over a theory containing only one clause. Then, SEMPRE enters the *Reduce phase* [lines 26-31], where the master collects all sets of scored refinements from the workers and updates the beam of promising clauses and the sets of target and background

[3] We assume that the worlds all have a two-valued well-founded model.

Algorithm 1. Function SEMPRE

```
1: function SEMPRE(I, n, NInt, NS, NA, NI, NV, ϵ, δ)
2:     IBs =InitialBeams(I, NInt, NS, NA)                    ▷ Clause search
3:     TC ← [], BC ← []
4:     for all (PredSpec, Beam) ∈ IBs do
5:         Steps ← 1, NewBeam ← []
6:         repeat
7:             while Beam is not empty do
8:                 if MASTER then
9:                     Refs ←ClauseRefinements((Cl, Literals), NV)
10:                    Split evenly Refs into n subsets
11:                    Send Refs_j to worker j
12:                else                                       ▷ the j-th slave
13:                    Receive Refs_j from master
14:                end if
15:                for all (Cl', Literals') ∈ Refs_j do
16:                    (LL'', {Cl''}) ←EMBLEM(I, {Cl'}, ϵ, δ)
17:                    NewBeam_j ←Insert((Cl'', Literals'), LL'')
18:                    if Cl'' is range restricted then
19:                        if Cl'' has a target predicate in the head then
20:                            TC ←Insert((Cl'', Literals'), LL'')
21:                        else
22:                            BC ←Insert((Cl'', Literals'), LL'')
23:                        end if
24:                    end if
25:                end for
26:                if MASTER then
27:                    Collect all the sets NewBeam_j from workers
28:                    Update NewBeam, TC, BC
29:                else                                       ▷ the j-th slave
30:                    Send the set NewBeam_j to master
31:                end if
32:            end while
33:            Beam ← NewBeam, Steps ← Steps + 1
34:        until Steps > NI
35:    end for
36:    if MASTER then
37:        T ← ∅, TLL ← −∞                                    ▷ Theory search
38:        repeat
39:            Remove the first couple (Cl, LL) from TC
40:            (LL', T') ←EMBLEM^MR(I, T ∪ {Cl}, n, ϵ, δ)
41:            if LL' > TLL then
42:                T ← T', TLL ← LL'
43:            end if
44:        until TC is empty
45:        T ← T ⋃_{(Cl,LL)∈BC} {Cl}
46:        (LL, T) ←EMBLEM^MR(I, T, n, ϵ, δ)
47:        return T
48:    end if
49: end function
```

clauses (TC and BC respectively): the scored refinements are inserted in order of LL into these lists.

The second parallelized operation is parameter learning for the theories with the target clauses. In this phase [lines 37-44], each clause from TC is tentatively added to the theory, which is initially empty. In the end, it contains all the clauses that improved its LL (search in the space of theories). In this case, parameter learning may be quite expensive since the theory contains multiple clauses, so a MapReduce version of EMBLEM called EMBLEM$^{\mathrm{MR}}$ is used.

The third parallelized operation is the final parameter optimization for the theory including also the background clauses [lines 45-46]. All the background clauses are added to the theory previously learned and the parameters of the resulting theory are learned by means of EMBLEM$^{\mathrm{MR}}$.

SEMPRE was implemented in Yap Prolog using the `lam_mpi` library for interfacing Prolog with the Message Passing Interface (MPI) framework.

SEMPRE was tested on the following seven real world datasets: Hepatitis, Mutagenesis, UWCSE, Carcinogenesis, IMDB, HIV and WebKB. All experiments were performed on GNU/Linux machines with an Intel Xeon Haswell E5-2630 v3 (2.40GHz) CPU with 8GB of memory allocated to the job.

Figure 1 shows the speedup of SEMPRE as a function of the number of workers. The speedup is always larger than 1 and grows with the number of workers, except for HIV and IMDB, where there is a slight decrease for 16 and 32 workers due to the overhead; however, these two datasets were the smallest and less in need of a parallel solution.

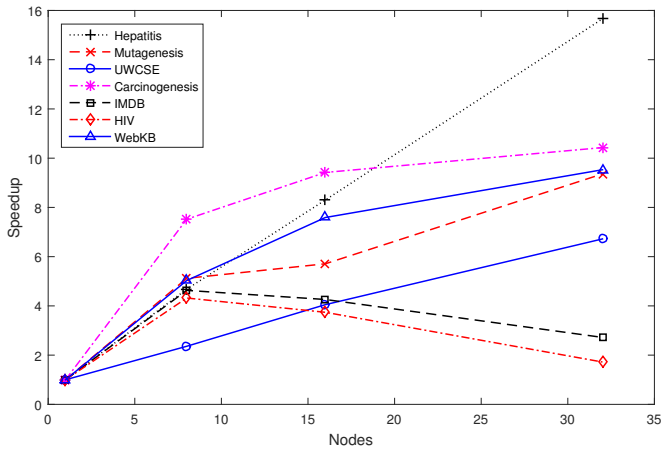

Figure 1. SEMPRE speedup.

4 Conclusions

The paper presents the algorithm SEMPRE for learning the structure of probabilistic logic programs under the distribution semantics. SEMPRE is a MapReduce implementation of SLIPCOVER, exploiting modern computing infrastructures for performing learning in parallel. The results show that parallelization is indeed effective at reducing running time, even if in some cases the overhead may be significant.

Acknowledgement This work was supported by the "GNCS-INdAM".

REFERENCES

[1] Elena Bellodi and Fabrizio Riguzzi, 'Expectation Maximization over Binary Decision Diagrams for probabilistic logic programs', *Intelligent Data Analysis*, **17**(2), 343–363, (2013).

[2] Elena Bellodi and Fabrizio Riguzzi, 'Structure learning of probabilistic logic programs by searching the clause space', *Theory and Practice of Logic Programming*, **15**(2), 169–212, (2015).

[3] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Sht. Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt, 'Inference and learning in probabilistic logic programs using weighted boolean formulas', *Theory and Practice of Logic Programming*, **15**(3), 358–401, (2015).

[4] Fabrizio Riguzzi and Terrance Swift, 'Well-definedness and efficient inference for probabilistic logic programming under the distribution semantics', *Theory and Practice of Logic Programming*, **13**(Special Issue 02 - 25th Annual GULP Conference), 279–302, (2013).

[5] Taisuke Sato, 'A statistical learning method for logic programs with distribution semantics', in *12th International Conference on Logic Programming, Tokyo, Japan*, ed., Leon Sterling, pp. 715–729, Cambridge, Massachusetts, (1995). MIT Press.