Structure Learning of Probabilistic Logic Programs by MapReduce

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

Dipartimento di Matematica e Informatica – University of Ferrara

Dipartimento di Ingegneria – University of Ferrara
[fabrizio.riguzzi,elena.bellodi,riccardo.zese,
giuseppe.cota,evelina.lamma]@unife.it

ILP 2015

UNIVERSITÀ DEGLI STUDI DI FERRARA
- EX LABORE FRUCTUS -

Probabilistic Logic Programming

- Logic + Probability: useful to model domains with *complex* and *uncertain* relationships among entities
- Probabilistic logic programming languages under the Distribution Semantics
  - Independent Choice Logic (ICL), PRISM, ProbLog, Logic Programs with Annotated Disjunctions (LPADs),...
  - They define a probability distribution over normal logic programs (possible worlds)
  - The distribution is extended to a joint distribution over worlds and queries
  - The probability of a query is obtained from this distribution by marginalization
  - They differ in the definition of the probability distribution

Probabilistic Inductive Logic Programming

- Learn the parameters given examples, a background and the structure of the program: PRISM, LeProbLog, LFI-ProbLog, EMBLEM, ProbLog2
- Learn the structure and the parameters given examples and a background: SLIPCASE, SLIPCOVER, LEMUR
- Problem: execution time in the range of hours for datasets fitting in main memory
- Proposed solution: scale systems by distributed parameter and structure learning by MapReduce.

EMBLEM

- EM over Bdds for probabilistic Logic programs Efficient Mining
- Parameter learning [Bellodi and Riguzzi, IDA 2013] inspired by [Ishihata et al., TR 2008] and [Thon et al., ECML 2008], similar to LFI-ProbLog [Gutmann et al., ECML 2011]
- Input: set of interpretations, target predicates, an LPAD
- BDDs encode the explanations for each ground fact Q for the target predicates
- Hidden variables: the selection of *i-th* head atom from groundings of the clauses used in the proof of Q
- EM algorithm

EMBLEM

- *Expectation:*
  - Expected counts of hidden variables: $E[c_{ik0}|Q]$ and $E[c_{ik1}|Q]$ for all rules C_i and $k = 1, ..., n_i(heads) - 1$, where c_{ikx} is the number of times a binary variable X_{ijk} takes value x $\in \{0, 1\}$, and for all values of $j \in g(i) = \{j|\theta_j \textit{ is a substitution grounding } C_i\}$
  - Expected counts are computed by traversing twice the BDDs
  - The counts for individual examples are summed up top obtain $E[c_{ik0}] = \sum_Q E[c_{ik0}|Q]$ and $E[c_{ik1}] = \sum_Q E[c_{ik1}|Q]$
- *Maximization:*
  - Computes maximum likelihood parameters from the distributions
  - parameters π_{ik} represent $P(X_{ijk} = 1)$ for all $j \in g(i)$ and for all rules C_i, $k = 1, ..., n_i - 1$; π_{ik} = $E[c_{ik1}]$ / ($E[c_{ik0}]$ + $E[c_{ik1}]$)

SLIPCOVER

- Structure LearnIng of Probabilistic logic programs by searChing OVER the clause space
- Two-phase search strategy:
  1. beam search in the space of clauses
  2. greedy search in the space of theories
- Clause search: beam search for each predicate separately. Initialize the beam with top clauses. Obtain refinements by adding a literal from a bottom clause built as in Progol [Muggleton, NGC 1995]. Evaluate refinements through LL by invoking EMBLEM. A fixed-size list of the best clauses is kept
- Theory search: add iteratively clauses from the 1st phase to an initially empty theory, run EMBLEM to compute the corresponding LL and keep the clause if the LL increases.

Distributed Parameter Learning by MapReduce: EMBLEM^{MR}

- We follow the approach of [Chu et al, NIPS 2006] for MapReduce EM: expectations are computed separately for the various examples and then aggregated in the Reduce phase
- in our case n workers from 1 to n. Worker 1 is the “master”, the others the “slaves”
- The Map function is performed by all workers; the Reduce function by the master (the “reducer”)
- The input interpretations I and the input theory T are replicated among all workers, the examples E are evenly divided among the n workers

EMBLEM^{MR}

- Each worker builds the BDDs for its examples. All the mappers stay active keeping the BDDs in memory
- The Expectation step is executed in parallel by sending the current values of the parameters to each mapper *m*, which computes the expectations for each of its examples
- The vector of expectations are sent back to the master that aggregates by component-wise sum them and performs Maximization

SEMPRE

- Structure lEarning by MaPREduce
- Parallelizes SLIPCOVER by employing *n* workers, one master and $n - 1$ slaves. All the workers initially receive all the input data
- First parallel operation: scoring the clause refinements: the revisions *Refs* for a clause are split evenly among the workers.
- Each worker returns the set of refinements with their log-likelihood (LL). Scoring is performed using (serial) EMBLEM
- In the Reduce phase the master updates the beam of promising clauses
- Second parallel operation: scoring the theory refinements with EMBLEM^{MR}

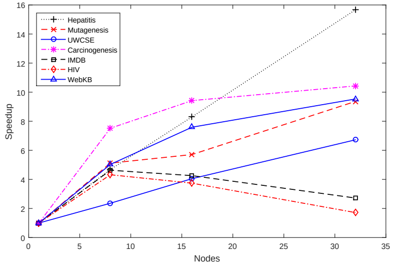
Experiments

- SEMPRE was implemented in Yap Prolog using the `lammpi` library by Nuno A. Fonseca and Vitor Santos Costa.
- Datasets: Hepatitis [Khosravi et al., ML 2012], Mutagenesis [Srinivasan et al., AI 1996], UWCSE [Kok and Domingos, ICML 2005], Carcinogenesis [Srinivasan et al., ILP 1997], IMDB [Mihalkova and Mooney, ICML 2007], HIV [Beerenwinkel et al., JCB 2005] and WebKB [Craven and Slattery, ML 2001]
- Machines with an Intel Xeon Haswell E5-2630 v3 (2.40GHz) CPU with 8GB of memory allocated to the job.

Experiments

	1	8	16	32
Hepatitis	19,867	4,246	2,392	1,269
Mutagenesis	14,784	2,887	2,587	1,579
UWCSE	12,758	5,401	3,152	1,899
Carcinogenesis	170	23	18	16
IMDB	481	104	113	177
HIV	508	118	136	295
WebKB	2,441	486	322	256

SEMPRE execution time (in seconds) as the number of slaves varies.



SEMPRE speedup (ratio of the running time of 1 worker to the running time of *n* workers)

Conclusions

- The speedup is always larger than 1 and grows with the number of workers except for HIV and IMDB with 16 and 32 processors
- Remarkable speedup both in parameter and structure learning
- Most time spend in the beam search of clause refinements: for UWCSE the time for clause search is around 94% of the total, for WebKB around 96%.
- Average time to handle a refinement is small, around 23ms for UWCSE and 80ms for WebKB
- Hence it is more reasonable to distribute the refinements to workers
- Overall, SEMPRE is able to exploit the availability of processors in most cases

Future Works

- Further investigate scaling, especially to datasets not fitting main memory
- Exploit distribution in-memory schemes such as Apache Spark

THANKS FOR LISTENING

AND

ANY QUESTIONS ?

UNIVERSITÀ DEGLI STUDI DI FERRARA
- EX LABORE FRUCTUS -