BoostRLR: The beauty of Prolog for statistical relational learning

Felix Weitkämper[0000−0002−3895−8279]

Institut für Informatik, Oettingenstr. 67, 80538 München felix.weitkaemper@lmu.de

Abstract. This paper presents an implementation of functional gradient boosting for relational logistic regression in less than 300 lines of Prolog code. This serves as a proof-of-concept for the perfect fit of Prolog as an implementation language for statistical relational learning algorithms, enabling rapid development and transparent code. Furthermore, the efficiency and additional features of modern Prolog engines such as tabling, multithreading and well-documented foreign-function interfaces contribute to the performance of Prolog implementations of such algorithms. As a vibrant, novel application area, statistical relational learning can give a new impetus to Prolog system development and motivate further optimisations and development.

Keywords: Functional gradient boosting · Relational logistic regression · Prolog

1 Introduction

Statistical relational artificial intelligence comprises approaches that combine probabilistic learning and reasoning with variants of first-order predicate logic. Compared to ordinary Bayesian networks or Markov networks, statistical relational languages have some key advantages. The generic relational representation allows features that refer directly to the relations between objects, and since the name and number of domain objects is not predetermined, models can be transferred between differently sized domains. Additionally, by fixing the signature, known symmetries can be enforced when learning the structure or the parameters of a model, which smooths out random fluctuations that may occur in the data. Finally, the compactness and genericity of learned models significantly enhances readability of models for humans. This is a key factor for achieving human-interpretable machine learning models.

The problem of statistical relational artificial intelligence can be approached from both directions; either, statistical models are lifted to make use of a relational representation, or logic programs are extended by probabilities. The latter approach gives rise to probabilistic logic programming (PLP). Since PLP has emerged from the logic programming and inductive logic programming communities, Prolog has played an important role as an implementation language for PLP from the very early days of the field [17]. In this contribution, we want to demonstrate the suitability of Prolog for target formalisms of the other kind, relational statistical models. Our case study concerns functional gradient boosting for relational logistic regression, which we briefly introduce now. A full discussion of the intricacies of relational logistic regression can be found in the original paper [8].

1.1 Relational logistic regression

%

Relational logistic regression We now introduce relational logistic regression in the form used for the functional gradient boosting algorithm of Ramanan et al. [14]. Relational logistic regression is a direct generalisation of logistic regression, a widespread method for probabilistic modelling with Boolean response variables.

In our setting, there is a single *target predicate* Q, whose interpretation depends probabilistically on the interpretation of a number of *extensional predicates.* Let $(x_i)_{i \in 1,\ldots,k}$ be a tuple of variables whose arity corresponds to the arity of the target predicate. To model such probabilistic dependencies, relational logistic regression employs *weighted formulas*, which are triples (φ, w_T, w_F), where φ is a formula and w_T and w_F are real numbers. In relational logistic regression, features are defined by conjunctive formulas $(\varphi_i, w_{T,i}, w_{F,i})_i$ of extensional predicates with weights w_i. Let $\iota(x_i) = a_i$ define a grounding of $Q(x_1, \ldots, x_k)$. Then the probability of $Q(a_1, \ldots, a_k)$ to hold given an interpretation of the feature predicates is defined as follows:

$$\mathbb{P}(Q\left(\boldsymbol{a}\right)) := \sigma\left(w_0 + \sum_i (w_{T,i} t_i + w_{F,i} f_i)\right)$$

where t_i and f_i are the numbers of true and false groundings (respectively) extending ι of the formula φ_i and $\sigma(z) = 1/(1+\exp(-z))$ is the sigmoid function.

Example 1. Consider a situation where people who have more friends are more likely to be considered social, and where we would like to predict whether an individual is social depending on the number of friends they have. In that case, there could be a single atom $\mathrm{Friends(x, y)}$ as a feature, with an associated positive weight w_T and negative weight w_F, as well as an arbitrary intercept weight w_0. The probability of an individual being social is then given by $\sigma(w + w_T t + w_F f)$, where t is the number of friends that individual has and f is the number of potential contacts (domain elements) that the individual is not friends with.

1.2 Functional gradient boosting for relational logistic regression

We approach scalable structure learning for relational logistic regression models from *functional gradient boosting*, which combines families of weak learners that are fast and easy to learn to stronger aggregate learners [6]. In our setting, the training data is given by a single interpretation of the extensional predicates and a finite set of *examples*, that is, groundings of the atomic formula $Q(x_1, \ldots, x_k)$, where Q is the target predicate. Every example is annotated with a truth value (*true* or *false*). The goal of learning a relational logistic regression model from data is to find a set of weighted conjunctive formulas that maximises the likelihood of the individual examples. As it is convenient instead to speak of a *loss function*, we consider the *gradient* as the difference between the actual truth value of an example (where *true* is taken to be 1 and *false* is taken to be 0) and the probability value predicted by the classifier. Thus, if the probability predicted is 0.8 and the example is in fact true, then the gradient would be -0.2, while if the example is in fact false the gradient would be 0.8.

To be able to adjust the intercept probability more easily as new formulas are added, Ramanan et al. [14] suggest learning not just formulas with weights for true and false instances, but a third weight for each formula which is simply added to the intercept.

We proceed to give a high-level overview of the algorithm. At the highest level, the algorithm consists of the following steps:
Until a prescribed number M of clauses is reached:

1. Find the current gradient of each example;
2. find a weighted conjunctive formula that, when added, reduces the (regularised) average square gradient, which will just be called *the score* in the following;
3. add that clause to the regression and repeat.

To find a good weighted clause, we start with a clause with an empty body (which corresponds to a formula that is always true), and weights $(0, 0, 0)$. We then proceed as follows until we either no longer improve the score or we reach a prescribed number L of conjuncts:

1. Enumerate possible atoms to conjoin to the formula; for each, find the optimal weights of the formula augmented by the atom, and calculate the score of the resulting formula;
2. choose the best-scoring literal and add it to the clause;
3. update the weight to that just calculated.

Enumeration is controlled by the user with the classical inductive logic programming technique of mode declarations [11], which will be explained in more detail when discussing our concrete Prolog implementation below.

This leaves open how the scores and optimal weights are computed. Ramanan et al. [14] derive a closed-form matrix representation of the problem as follows: For each triple-weighted formula φ, let C_φ be the matrix whose i-th row is $[1, t_\varphi(j), f_\varphi(j)]$, where $t_\varphi(j)$ and $f_\varphi(j)$ are the numbers of true and false groundings respectively that correspond to the j-th example. Let $\boldsymbol{w}_\varphi$ be a weight vector $[w_0, w_T, w_F]$. Let $\varDelta_j$ be the j-th example gradient of the existing clauses. Let $\lambda > 0$ be a regularisation parameter.

Then Ramanan et al. [14] show that the regularised square error of the model augmented with the triple-weighted formula $(\varphi, \boldsymbol{w})$ can be written compactly as

$$\|Cw - \varDelta\|^2 + \lambda \|w\|^2 .$$

The optimal weights are in fact solutions to the linear system

$$\left(C^{\mathrm{T}} C + \lambda \mathrm{I}\right) w = C^{\mathrm{T}} \Delta$$

or, equivalently, computed by

$$w = \left(C^{\mathrm{T}} C + \lambda \mathrm{I}\right)^{-1} C^{\mathrm{T}} \Delta. \tag{1}$$

2 Prolog as an implementation language

Several factors make Prolog an ideal implementation language for such statistical relational learning approaches. Firstly, there is a direct representation of quantifier-free conjunctive formulas as conjunctive Prolog goals, and of an interpretation of any number of relation symbols on a finite domain as a Prolog knowledge base. This reduces calculating the number of true groundings of a conjunctive formula with respect to an interpretation to simply counting the number of solutions of a goal with respect to a knowledge base. Similarly, integrating background knowledge is seamless in Prolog, since a knowledge base can itself be seen as simply another Prolog program to be consulted and evaluated. Thus, Prolog clauses provide a uniform representation for all variable aspects of the algorithm.

More generally, Prolog is a very compact language per se, and allows for the transparent, brief and flexible expression of a variety of search algorithms. Above and beyond the ISO standard, modern Prolog implementations provide valuable extensions that contribute significantly to the implementation. The first and most important is tabling, the comprehensive and disciplined integration of memoisation with the Warren Abstract Machine model of computation implemented in the SLG-WAM abstract machine [3]. By keeping tables of already computed answers in memory, tabling equips Prolog with the functionalities of a deductive database engine [16]. Repeatedly computing the numbers of solutions of goals with respect to a knowledge base is a classic deductive database task that profits from keeping previous results in memory. Tabling was pioneered by XSB, which still provides a particularly mature and efficient implementation [19]. However, tabled evaluation has also been integrated into other wide-spread systems such as SWI-Prolog [24] and YAP [4], which allows its benefits to be combined with other features that can make those systems preferable for a given task.

As part of the main loop of our algorithm, the program scores several candidate extensions of a clause independently and chooses the best one. This immediately suggests parallelisation. Arguably the best-maintained multi-threading engine, which is compatible with all the other language features offered there, is implemented in SWI-Prolog [23]. In previous versions, XSB and YAP had extended their engines with multithreading support, and significant work was put into the efficient and semantically sound combination of tabling and multi-threading in both systems [10, 1]. However, by 2023, multi-threading has been deprecated in both systems (Swift, Theresa and Rocha, Ricardo, personal communication).

Thus, we seemed to face the choice of either prioritising efficient and mature tabling or multi-threading capability when choosing a system for our implementation. Eventually, we made the decision to make our code executable on either XSB or SWI-Prolog using a portable system of dialect flags, allowing the user to choose his system based on personal preference and familiarity or on the characteristics of his dataset.

Further evidence of the suitability of Prolog for this task is given by the existing Java implementation itself [13], which expends many of its more than 80,000 lines of code on implementations of core reasoning functionality (in this case adapted from the Wisconsin Inductive Logic Learner [12]) that is natively available in Prolog.

Existing use of Prolog for statistical relational artificial intelligence has centered on PLP. This includes the implementation of structure learning algorithms such as SLIPCOVER [2] and LIFTCOVER [5], which are probably the closest analogues to our present work. However, they differ not just in their target formalism, but employ a completely different learning methodology unrelated to functional gradient boosting. In addition to implementing learning algorithms themselves, applications to PLP have motivated several auxiliary utilities, such as the recent Prolog encoding of automatic differentiation [18] and a SWI-Prolog pack of Prolog-native matrix operations [15].

3 Implementation

Our Prolog implementation has only 300 lines including empty lines and comments, and we will present what we consider the most insightful components of it here, highlighting the design choices and Prolog features used. The software exists in two different forms: a SWI-Prolog version with no dependencies on foreign functions, ustilising the aforementioned Matrix pack, and a more highly performant version which makes use of C implementations of the linear algebra and is compatible with both XSB and multithreaded SWI-Prolog through dialect flags.

3.1 Input files

The input to the algorithm is provided in the form of four files, namely bias.pl, which contains the mode declarations, settings.pl, which specifies the number of clauses to be learnt, the maximum length of any clause and a regularisation parameter λ, kb.pl, which contains the training domain and the background knowledge and lastly examples.pl, which contains the actual positive and negative examples used for learning.

For a simple running example of family relations, the bias file could look as follows:

```
:- module(bias,[modeb/2]).
modeb(father(_,_),childof(a,a)).
modeb(father(_,_),male(a)).
modeb(father(_,_),siblingof(a,a)).
```

The module declaration never has to be changed by the user, and the mode declarations say that the predicates `childof`/2, `male/1` and `sibling/2` can occur in clauses predicting father/2. The 'a' in the argument position stands for 'any' and means that any variable can be used in that argument position, whether it already occurred earlier in the clause (an *input* variable) or not (an *output* variable). If the arguments should be restricted to either input or output variables, 'a' could be replaced with 'i' or 'o' respectively.

While the algorithm as developed by Ramanan et al. [14] only covers single-target learning, making the explicit head declaration redundant, we include it in the mode declarations in order to facilitate future generalisation to multi-target learning.

The settings file sets the parameters as simple unary clauses, and only the three arguments have to be adapted by the user to set the parameters.

```
:- module(settings,[lambda/1,clause_length/1, clause_num/1]).
lambda(100).
clause_length(2).
clause_num(1000).
```

The knowledge base could be any Prolog program defining the background knowledge, including a tabled predicate defining the training domain. In our case, an excerpt of it could be such:

```
:- table domain/1.
male(fredweasley).
male(jamespotter).
male(harrypotter).
siblingof(fredweasley,ginnyweasley).
childof(jamespotter,harrypotter).
childof(lilypotter,harrypotter).
domain(X) :-
    male(X);
    siblingof(X,_);
    siblingof(_,X);
    childof(X,_);
    childof(_,X).
```

Finally, the examples file is another arbitrary Prolog program defining the positive and negative training examples, in this case containing among others the following facts:

```
positive(father(harrypotter,jamespotter)).
negative(father(harrypotter,lilypotter)).
negative(father(harrypotter,fredweasley)).
```

3.2 A guided tour of program highlights

`b_rlr(+,-)` is the main entry point to the program. It takes the target predicate with (usually anonymous) variables as its first argument and returns the list of weighted clauses as its second argument. So, a typical call would be

```
?- b_rlr(father(_,_),WeightedClauses).
```

A key advantage of using Prolog is that the target and the background knowledge can themselves be immediately processed by the engine. However, a side effect of this feature can be the propagation of unwanted variable bindings through the program. To counteract this, we immediately pass the query through `numbervars/1`, which "freezes" head variables to designated terms which can be "thawed" again later when needed. We implement the main loop of the program using the classic accumulator technique to facilitate tail recursion:

```
b_rlr(Head,WeightedClauses) :-
    numbervars(Head),
    b_rlr(Head,0,[],WeightedClauses).
b_rlr(_,Acc,WeightedClauses,WeightedClauses) :-
    clause_num(Acc).
b_rlr(Head,Acc,OldWeightedClauses,WeightedClauses) :-
    \+clause_num(Acc),
    compute_gradients(Head,OldWeightedClauses,Gradients),
    fit_regression(Head,Gradients,Clause,Weights),
    Acc1 is Acc +1,
    NewWeightedClauses = [(Weights::Clause)|OldWeightedClauses],
    b_rlr(Head,Acc1,NewWeightedClauses,WeightedClauses).
```

The key algorithmic components are the predicates `compute_gradients/3`, which evaluates clauses against the training examples, and `fit_regression/4`, which builds the optimal next clause to add. We discuss the implementation of each in turn.

`compute_gradients(+,+,-)` takes as arguments the query used for testing and the weighted clauses returned e.g. by `b_rlr/2`, and returns the list of gradients, i. e. the list of deviations between the predicted probabilities and the real truth value of the query. It is implemented using `findall/3` over an auxiliary predicate, which returns the gradients of all possible examples upon backtracking over `compute_gradients`. `compute_gradient(+,+,-)` finally computes a single example gradient by performing inference over the weighted clauses.

```
compute_gradient(Head,WeightedClauses,Gradient) :-
    positive(Head),
    inference(Head,WeightedClauses,Prob),
    Gradient is 1 - Prob.
compute_gradient(Head,WeightedClauses,Gradient) :-
    negative(Head),
    inference(Head,WeightedClauses,Prob),
    Gradient is -Prob.
```

`inference(+,+,-)` computes the probability of a ground instance of the target being true as predicted by the weighted clauses. It does so by calculating the numbers of true and false groundings and applying the sigmoid function to convert weights to probabilities. All the mathematics required for this computation is implemented by fast core Prolog arithmetic.

```
inference(Head,WeightedClauses,Prob) :-
    ground(Head),
    aggregate_all(sum(S),
                 (member(([W0,W1,W2]::(Head <= Body)),WeightedClauses),
                    calculate_groundings(Body,T,F),
                    S is W0 + W1*T + W2*F),CompoundWeight),
    sigmoid(CompoundWeight,Prob).
sigmoid(W,S) :-
    S is (exp(W) / (exp(W) + 1)).
```

`calculate_groundings(+,-,-)` counts the number of true and false groundings of a clause body. This is done by counting the solutions to the goal, and then deducting it from the overall number of possible solutions. Since forming a list and then considering its length is a very inefficient way to count, we use a dedicated counting predicate count/2. In SWI-Prolog, this can rely on an efficient specialised implementation of `aggregate_all(count,_,_)`, while we provide a dedicated low-level XSB implementation in a count module that we include with our program, courtesy of David S. Warren (personal communication).

```
calculate_groundings(Body,T,F) :-
    term_variables(Body,VarsList),
    count(Body, T),
    length(VarsList,Exponent),
    count_domain(All),
    Alltups is All ** Exponent,
    F is Alltups - T.
count_domain(All) :-
    count(domain(_),All).
```

We now turn to `fit_regression(+,-,-,-)`, which implements the inner loop. It relies on `extend_body(+,+,+,+,+,-,-,-)`, which finds the best-scoring extension of a body by a mode-conforming literal. `extend_body/8` employs a `find_all` over the scored literals, chooses the best-scoring one and then iterates this until either no further literal improves the score or the maximum number of clauses has been reached.

```
fit_regression(Head,Gradients,Clause,Weights) :-
    extend_body(Head,true,1.0E10,[0,0,0],0,Gradients,Body,Weights),
    Clause = (Head <= Body).

extend_body(Head,Body,Score0,OldWeights,Acc,Grads,NewBody,ClWghts) :-
    findall((Body,Literal)-Score-Weights,
            (possible_literal(Head,Body,Literal),
             unnumbervars((Head,Body,Literal),
                          (ReHead,ReBody,ReLiteral)),
             score(ReHead,ReBody,Grads,ReLiteral,Weights,Score)),Ls),
    Extensions = [Body-Score0-OldWeights|Ls],
    argminlist(Extensions,Body2,Score2,Weights2),
    Acc1 is Acc + 1,
    clause_length(Max),
    (   (Body2 = Body
        ; Acc1 = Max
        )
    ->  Body2 = NewBody,
        ClWghts = Weights2
        ;     extend_body(Head,Body2,Score2,Weights2,
                          Acc1,Grads,NewBody,ClWghts)).
```

As this predicate scores a range of possible literals independently of each other, it is predestined for parallelisation. Therefore, we allow the user to toggle parallel execution under SWI-Prolog using a command-line flag, which replaces the above definition of extend_body with a restructured one that makes use of SWI-Prolog's built-in `concurrent_maplist` predicate.

`possible_literal(+,+,-)` generates mode-conforming extensions by one literal on backtracking, where the actual parsing of the mode declarations is taken on by `instantiation/3`.

```
possible_literal(Head,Body,Literal) :-
    unnumbervars((Head,Body),(ReHead,ReBody)),
    modeb(Head,Schema),
    instantiation(Schema,(ReHead,ReBody),Literal).
```

`score(+,+,+,+,-,-)` computes the optimal weights and the corresponding score of a body and literal. It is implemented using findalls over `score_aux(+,+,+,-,-)`, which returns the numbers of true and false groundings of all possible examples upon backtracking. It then passes the results of the findalls to the dedicated C-implemented function `opt_weight_score` (where flags deal with the slightly different foreign function interfaces of XSB and SWI-Prolog), which implements Equation 1.

```
score_aux(Head,Body,Literal,T,F) :-
    copy_term((Head,Body,Literal),(Head2,Body2,Literal2)),
    example(Head2),
    calculate_groundings((Body2,Literal2),T,F).
```

For ease of maintenance, the C code implementing Equation 1 itself is collected in a single C file and uses a specialised Cholesky decomposition for better numerical stability. Additional custom-made files then pass the computations to the foreign function interfaces of XSB and SWI-Prolog respectively.

In the stand-alone Prolog implementation used for teaching, the predicate `opt_weight_score` is implemented in a Prolog file making use of Riguzzi's matrix library [15].

4 Experimental evaluation

The main considerations for our implementation were adaptability, extendibility and compactness of the code. However, since some planned extensions such as multi-target learning require a significant number of iterations of structure learning to be executed, performance was still an important consideration.

We here present some initial experiments that merely show the benefit of several of the Prolog features we exploit. While it would of course also be interesting to benchmark our implementation against the Java reference [13], we have so far been unable to execute that code successfully.

All the experiments we present in Table 1 here use the family database we mentioned above, which has just 40 facts, 5 positive examples and 21 negative examples. While benchmarking on bigger training problems is work for the immediate future, in the meantime we can scale the difficulty of the problem by using different settings. We compare 5 different configurations: The vanilla pure Prolog version we use for teaching ("vanilla"), with some manual memoising but without tabling, and a pure Prolog version with tabling ("tabled") (both executed under SWI-Prolog); the version we explained above, with linear algebra implemented in C, on XSB ("XSB") and on SWI-Prolog (there with ("SWI-mt") and without multi-threading ("SWI")). All experiments were conducted on a laptop running Ubuntu 20.04, with 15,3 GiB RAM and an Intel i7 1,9 GHz * 8 core processor. We use SWI-Prolog 9.3.7 with optimisation enabled and XSB 5.0. As execution times differ quite a bit between runs, they should only be taken as a rough estimate. This shows that at least for this small dataset, the superior tabling performance in XSB outweighs the downsides of multithreading, but that as long as the clause length is sufficient for enough parallel scoring to be required, multithreading does have a positive impact. We would expect this to grow with the complexity of the individual scoring task, making multithreading more competitive on larger knowledge bases.

Table 1. Execution times of the experiments in seconds, with a 4 minute time-out.

settings	vanilla	tabled	SWI	SWI-mt	XSB
6, 20	6.6	2.5	1.1	0.77	0.43
6, 200	>240	22.9	7.1	6.6	3.0
2, 1000	122	39.4	24.4	28.1	9.6

5 Discussion and conclusion

Our implementation has captured the algorithm exactly as described [14], while requiring less than 300 lines in comparison to the only alternative Java implementation's more than 80,000 lines of code [13] (a reduction in code length by a factor of more than 250). While some of the linear algebra has been outsourced to C code, even this is very compact, with less than another 300 lines of code including separate parameter-passing from both SWI-Prolog and XSB. By utilising tabling, multithreading and the foreign-function interface, our implementation has nonetheless taken significant steps towards computational efficiency.

However, our main motivation in reimplementing this algorithm in Prolog was to have a research prototype with which to implement variations and extensions of the original algorithm. Among the extensions that are currently in progress is multi-target learning in the original setting of Kazemi et al. [8], who use relational logistic regression as an aggregation function within lifted Bayesian networks.

There is a vast variety of search techniques in use for learning the structure of Bayesian networks [9], many of which are excellently suited to implementation in Prolog. Our Prolog rendering of learning and inference for relational logistic regression can very easily be augmented by adding a structure learning algorithm on top, which calls the entry point of the program inside its loop. Integration into a single Prolog program also allows tabling to act across different iterations of the structure search, maximising efficiency.

Recently, Weitkämper [22, 21] has demonstrated the asymptotic consistency of relational logistic regression when training and test domains are of different size, as long as the weight parameters are scaled commensurately with the size of the domain. Such scaling can easily be inserted into our implementation by querying domain sizes before performing inference. It also allows for learning from several training domains of different sizes, applying a different scaling factor to each training domain depending on its size. The Prolog_DB package shipped with XSB offers a very elegant declarative approach to this problem, whose computational efficiency and compatibility with the SWI-Prolog ecosystem we have yet to evaluate.

Ultimately, despite its clear advantages, using Prolog directly will always limit the general appeal of an implementation. Hence, the recently presented Prolog-Python bridge Janus [20], implemented in a widely compatible way by both XSB and SWI-Prolog, can prove a key contribution towards the feasibility of using Prolog as an implementation language for machine learning algorithms.

A blueprint for this could be srlearn [7], which serves a similar function for those functional gradient boosting algorithms implemented in Java. A possible challenge to overcome would be the integration of the C code, which could be simplified by reimplementing it in Python using numpy. In the other direction, Janus also facilitates access to Python's enormous data analysis and visualisation ecosystem directly from Prolog.

It is painful to see that so much effort that has gone into seamlessly and efficiently integrating parallelisation and tabling within YAP and XSB has been made inaccessible by the deprecation of both Prolog's multithreading capabilities. Part of the reason for this seems to be a lack of clear applications to justify the maintenance burden of a separate mutlithreaded engine. From this perspective, we present (relational) machine learning as a vibrant field that can benefit from precisely this combination of features.

Therefore, we believe that adopting transparent implementations of cutting-edge relational learning algorithms can open up new fields of application to modern Prolog engines, giving their developers new stimuli for their optimisation. Indeed, in addition to the new counting predicate the work presented here has already motivated improvements to the core XSB engine that significantly sped up the call of core built-ins. Most importantly though, Prolog can help overcome the implementation bottleneck that hampers progress in statistical relational learning and serves as a huge barrier to entry for researchers outside large, well-established research groups with many years of tooling and implementation craft passed down through generations of students.

References

1. Areias, M., Rocha, R.: Table space designs for implicit and explicit concurrent tabled evaluation. Theory Pract. Log. Program. **18**(5-6), 950–992 (2018). https://doi.org/10.1017/S147106841800039X
2. Bellodi, E., Riguzzi, F.: Structure learning of probabilistic logic programs by searching the clause space. Theory Pract. Log. Program. **15**(2), 169–212 (2015). https://doi.org/10.1017/S1471068413000689
3. Chen, W., Warren, D.S.: Tabled evaluation with delaying for general logic programs. J. ACM **43**(1), 20–74 (1996). https://doi.org/10.1145/227595.227597
4. Costa, V.S., Rocha, R., Damas, L.: The YAP prolog system. Theory Pract. Log. Program. **12**(1-2), 5–34 (2012). https://doi.org/10.1017/S1471068411000512
5. Fadja, A.N., Riguzzi, F.: Lifted discriminative learning of probabilistic logic programs. Mach. Learn. **108**(7), 1111–1135 (2019). https://doi.org/10.1007/S10994-018-5750-0
6. Friedman, J.H.: Greedy function approximation: a gradient boosting machine. Annals of statistics **29**, 1189–1232 (2001)
7. Hayes, A.L.: srlearn: A python library for gradient-boosted statistical relational models. CoRR **abs/1912.08198** (2019), http://arxiv.org/abs/1912.08198, presented at StarAI 2020
8. Kazemi, S.M., Buchman, D., Kersting, K., Natarajan, S., Poole, D.: Relational logistic regression. In: Baral, C., Giacomo, G.D., Eiter, T. (eds.) Principles of Knowledge Representation and Reasoning: Proceedings of the Fourteenth International Conference, KR 2014, Vienna, Austria, July 20-24, 2014. AAAI Press (2014)

9. Kitson, N.K., Constantinou, A.C., Guo, Z., Liu, Y., Chobtham, K.: A survey of bayesian network structure learning. Artif. Intell. Rev. **56**(8), 8721–8814 (2023). https://doi.org/10.1007/S10462-022-10351-W
10. Marques, R., Swift, T., Cunha, J.C.: A simple and efficient implementation of concurrent local tabling. In: Carro, M., Peña, R. (eds.) Practical Aspects of Declarative Languages, 12th International Symposium, PADL 2010, Madrid, Spain, January 18-19, 2010. Proceedings. Lecture Notes in Computer Science, vol. 5937, pp. 264–278. Springer (2010). https://doi.org/10.1007/978-3-642-11503-5_22
11. Muggleton, S.H.: Inverse entailment and progol. New Gener. Comput. **13**(3&4), 245–286 (1995). https://doi.org/10.1007/BF03037227
12. Natarajan, S., Kunapuli, G., O'Reilly, C., Maclin, R., Walker, T., Page, D., Shavlik, J.: Ilp for bootstrapped learning: A layered approach to automating the ilp setup problem (2009), https://dtai-static.cs.kuleuven.be/events/ilp-mlg-srl/USBStick/papers/ILP09-8.pdf, presented as a poster at the 19th Conference of Inductive Logic Programming.
13. Ramanan, N., Khot, T., Natarajan, S.: Rlr_boost (2018), https://github.com/nandhiniramanan5/
14. Ramanan, N., Kunapuli, G., Khot, T., Fatemi, B., Kazemi, S.M., Poole, D., Kersting, K., Natarajan, S.: Structure learning for relational logistic regression: an ensemble approach. Data Min. Knowl. Discov. **35**(5), 2089–2111 (2021). https://doi.org/10.1007/S10618-021-00770-8
15. Riguzzi, F.: Matrix (2023), https://github.com/friguzzi/matrix
16. Sagonas, K., Swift, T., Warren, D.S.: XSB as an efficient deductive database engine. In: Snodgrass, R.T., Winslett, M. (eds.) Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, USA, May 24-27, 1994. pp. 442–453. ACM Press (1994). https://doi.org/10.1145/191839.191927
17. Sato, T.: A statistical learning method for logic programs with distribution semantics. In: Sterling, L. (ed.) Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, Tokyo, Japan, June 13-16, 1995. pp. 715–729. MIT Press (1995)
18. Schrijvers, T., van den Berg, B., Riguzzi, F.: Automatic differentiation in prolog. Theory Pract. Log. Program. **23**(4), 900–917 (2023). https://doi.org/10.1017/S1471068423000145
19. Swift, T., Warren, D.S.: XSB: extending prolog with tabled logic programming. Theory Pract. Log. Program. **12**(1-2), 157–187 (2012). https://doi.org/10.1017/S1471068411000500
20. Swift, T., Andersen, C.: The janus system: Multi-paradigm programming in prolog and python. In: Pontelli, E., Costantini, S., Dodaro, C., Gaggl, S.A., Calegari, R., d'Avila Garcez, A.S., Fabiano, F., Mileo, A., Russo, A., Toni, F. (eds.) Proceedings 39th International Conference on Logic Programming, ICLP 2023, Imperial College London, UK, 9th July 2023 - 15th July 2023. EPTCS, vol. 385, pp. 241–255 (2023). https://doi.org/10.4204/EPTCS.385.24
21. Weitkämper, F.: Probabilities of the third type: Statistical relational learning and reasoning with relative frequencies. CoRR **abs/2202.10367** (2024), https://arxiv.org/abs/2103.15140
22. Weitkämper, F.: Scaling the weight parameters in Markov logic networks and relational logistic regression models. CoRR **abs/2103.15140** (2024), https://arxiv.org/abs/2103.15140

23. Wielemaker, J.: Native preemptive threads in swi-prolog. In: Palamidessi, C. (ed.) Logic Programming. pp. 331–345. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
24. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: Swi-prolog. Theory Pract. Log. Program. **12**(1-2), 67–96 (2012). https://doi.org/10.1017/S1471068411000494