The Workshops of the Thirtieth AAAI Conference on Artificial Intelligence
Declarative Learning Based Programming: Technical Report WS-16-07

RELOOP: A Python-Embedded Declarative Language for Relational Optimization

Martin Mladenov Danny Heinrich* Leonard Kleinhans* Felix Gonsior Kristian Kersting
Computer Science Department, TU Dortmund University
{fn.ln}@cs.tu-dortmund.de

Abstract

We present RELOOP, a domain-specific language for relational optimization embedded in Python. It allows the user to express relational optimization problems in a natural syntax that follows logic and linear algebra, rather than in the restrictive standard form required by solvers, and can automatically compile the model to a lower-order but equivalent model. Moreover, RELOOP makes it easy to combine relational optimization with high-level features of Python such as loops, parallelism and interfaces to relational databases. RELOOP is available at http://www-ai.cs.uni-dortmund.de/weblab/static/RLP/html/ along with documentation and examples.

Introduction

"Democratization of data" does not mean dropping a huge spreadsheet on everyone's desk and saying "good luck." It means making machine learning and AI methods usable in a way that people can easily instruct machines to have a "look" at the data and help them to understand and act on it. Therefore, it is not surprising that probabilistic logical languages are currently provoking much new AI research with tremendous theoretical and practical implications, see e.g. (Getoor and Taskar 2007; De Raedt 2008; De Raedt et al. 2008; De Raedt and Kersting 2010; Russell 2015) and references therein for overviews. However, instead of looking at AI through the glasses of probabilities over possible worlds, we may also approach it using optimization. That is, we have a preference relation, i.e., some objective function over possible worlds, and we want a best possible world according to the preference. Consider for example a typical data analyst solving a machine learning problem for a given dataset. She selects a model for the underlying phenomenon to be learned, formats the raw data according to the chosen model, tunes the model parameters by minimizing some objective function induced by the data and the model assumptions, and may iterate the last step as part of model selection and validation.

This is an instance of the declarative "Model + Solver" paradigm that was and is prevalent in AI (Geffner 2014), natural language processing (Rush and Collins 2012), machine learning (Sra, Nowozin, and Wright 2011), and data mining (Guns, Nijssen, and De Raedt 2011): instead of outlining how a solution should be computed, we specify what the problem is in terms of some high-level modeling language and solve it using general solvers.

Arguably, the "Model + Solver" paradigm involves more than just the specification and optimization of an objective function subject to constraints. Before optimization can take place, a large effort is needed to not only formulate the model but also to put it into the right form. We must often build models before we know what individuals are in the domain and, therefore, before we know what variables and constraints exist. As prominently witnessed by the NLP community (Riedel and Clarke 2006; Yih and Roth 2007; Clarke and Lapata 2008; Martins, Smith, and Xing 2009; Riedel, Smith, and McCallum 2012; Cheng and Roth 2013; Singh et al. 2015; Kordjamshidi, Roth, and Wu 2015), we must often solve optimization problems over relational domains where we have to reason about a varying number of objects and relations among them, without enumerating them. Hence modeling should facilitate the formulation of declarative, relational knowledge. Moreover, this not only concerns the syntactic form of the model but also the solvers; the efficiency with which the problem can be solved is to a large extent determined by the way the model is formalized and compiled for the solver.

An initial step towards such a general relational optimization framework are relational linear programs (Kersting, Mladenov, and Tokmakov 2015), a simple framework combining linear and logic programming. They are compact templates defining the linear objective and the linear constraints through the logical concepts of individuals, relations, and quantified variables. This contrasts with mainstream LP template languages such as AMPL (Fourer, Gay, and Kernighan 2002), which mix imperative and linear programming, as well as CVXPY (Diamond and Boyd 2015), which combines matrix notation and (object-oriented) Python, and in turn allows a more intuitive representation of optimization problems over relational domains. Since the templates are instantiated multiple times to construct the solver model, the model is likely to exhibit symmetries, which can be detected and exploited automatically to speed up solving.

*Contributed equally to the development of RELOOP.

Copyright © 2016, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

In this paper, we present RELOOP, a domain-specific language for relational optimization embedded in Python based on (Kersting, Mladenov, and Tokmakov 2015). RELOOP allows users to express relational optimization problems in a natural syntax that follows logic and linear algebra, rather than in the restrictive standard form required by solvers, and can efficiently reduce the model to a lower-order but equivalent model whenever symmetry is present. Another important feature of RELOOP is that it makes it easy to combine relational optimization with high-level features of Python such as loops, parallelism and interfaces to relational databases. This allows the user to take shortcuts around problems which are less elegantly expressed declaratively in (relational) logic.

We proceed as follows. We start off with an introduction to RELOOP. To this end, we present three applications from AI and machine learning, taking a tour of the RELOOP's features along the way. Before concluding, we discuss RELOOP's general architecture, going into details for one of its ingredients, namely grounding.

RELOOP: Relational Optimization in Python

We now illustrate RELOOP using three AI applications: Sudoku, Stackelberg games, and LP-SVMs. For more details on some of the mathematical underpinnings, we refer to (Kersting, Mladenov, and Tokmakov 2015).

RELOOP Walkthrough Example: Sudoku

Sudoku presents one with a 9x9 square grid subdivided into 3x3 boxes with 3x3 squares each, cf. Fig. 1a. The grid is partially populated with clues, and your task is to fill in the rest of the grid with integers from 1 through 9 so that each integer appears only once in each row, column, and major 3-by-3 square. To solve such a puzzle one can employ an assignment linear program (LP) as shown in Fig. 1b. This LP has a decision variable $\texttt{fill}(X, Y, N)$ for every triple $(X, Y, N) \in \{1, \ldots, 9\}^3$. The intent of these variables, which are bounded between 0 and 1, is to indicate if the number N is filled in the empty square with coordinates (X, Y). E.g., having $\texttt{fill}(9, 7, 1) = 1$ in our solution represents that the square in the ninth row and seventh column receives the number 1. The constraints of the Sudoku problem can naturally be expressed as linear constraints as shown in the Figure. This LP happens to be totally unimodular, thus solving it via the simplex method always results in a solution where the variables are either 0 or 1. This allows us to read off a valid assignment by looking at the variables which receive a value of 1 in the solution.

First, we import the necessary RELOOP components:

```
from reloop.languages.rlp import *
from reloop.languages.rlp.grounding.block import
    BlockGrounder
from reloop.languages.rlp.logkb import PyDatalogLogKb
from reloop.solvers.lpsolver import CvxoptSolver
```

Let us shortly explain what these are. In order to create a relational LP model (RLP), we need three objects—a logical knowledge base, a solver and a grounder. The LogKB interface provides the RLP with means to query the relational database/reasoning engine where the data is stored. Currently, RELOOP supports pyDatalog, PostgreSQL, SWI Prolog and ProbLog[1]. Here, we will use pyDatalog. The solver interface interfaces the RLP to a linear programming solver such as glpk, CXOPT or Gurobi. Finally, the grounder is an object that implements a strategy of parsing the RLP constraints and querying the LogKB in order to convert the RLP to matrix form, which the solver understands.

We now instantiate the three objects in question:

```
logkb = PyDatalogLogKb()
grounder = BlockGrounder(logkb)
solver = CvxoptSolver(solver_solver='glpk')
```

The option solver_solver = 'glpk' is a passthrough argument that tells CVXOPT to use glpk, since we need to solve the Sudoku LP with a simplex method. With this, we are ready to instantiate the model:

```
model = RlpProblem("play sudoku for fun and profit",
                   LpMaximize, grounder, solver)
```

The model takes as arguments a grounder (the LogKB is accessed through the grounder), a solver and a sense of the objective, either LpMinimize or LpMaximize. For this example it does not really matter.

Before we start defining constraints, we declare our predicates, constants, and logical variables. These symbols are defined with

```
I, J, X, U, V = sub_symbols('I', 'J', 'X', 'U', 'V')
```

Now we can define the predicates. Reloop has two different kinds of predicates: **numeric** predicates (essentially functions) that return a numeric value, e.g. $\texttt{pred}('\texttt{a}', '\texttt{b}') \mapsto 100$. In the LogKB, this numerical atom is stored as $\texttt{pred}('\texttt{a}', '\texttt{b}', 100)$. **Boolean** predicates that return a boolean value, e.g., $\texttt{pred}('\texttt{a}', '\texttt{b}', 100) \mapsto \texttt{True}$. In our sudoku LP, we have the following predicates:

```
num = boolean_predicate("num", 1)
boxind = boolean_predicate("boxind", 1)
box = boolean_predicate("box", 4)
initial = boolean_predicate("initial", 3)
fill = numeric_predicate("fill", 3)
```

More precisely, predicate declarations in RELOOP take two arguments—a predicate name and an arity. A predicate can be a variable predicate, meaning that each of its atoms in the Herbrand basis are decision variables. If a predicate is not a variable, it must be interpreted in the knowledge base. The variable predicate in our running example is `fill`:

```
model.add_reloop_variable(fill)
```

The other predicates will be interpreted in the knowledge base. Since we leave the knowledge base discussion for the end of this example, let us briefly mention what these predicates are supposed to mean.

[1]https://dtai.cs.kuleuven.be/problog/

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

(a) Sudoku example taken from Wikipedia

$$
\begin{aligned}
\underset{\mathtt{fill}\in\mathbb{R}^{9\times 9\times 9}}{\text{minimize}} \quad & 0 \\
\text{subject to} \quad & \forall X, N \in [1,9]: \quad \textstyle\sum_{N\in[1,9]} \mathtt{fill}(X,Y,N) = 1\ , \\
& \forall Y, N \in [1,9]: \quad \textstyle\sum_{N\in[1,9]} \mathtt{fill}(X,Y,N) = 1\ , \\
& \forall X, N \in [1,9]: \quad \textstyle\sum_{N\in[1,9]} \mathtt{fill}(X,Y,N) = 1\ , \\
& \forall B \in \mathtt{Box}, N \in [1,9]: \quad \textstyle\sum_{(X,Y)\in B} \mathtt{fill}(X,Y,N) = 1\ . \\
& \forall X, Y, N \in [1,9]: \quad 1 \geq \mathtt{fill}(X,Y,N) \geq 0\ ,
\end{aligned}
$$

(b) Assignment Linear Program for solving Sudoku

Figure 1: (a) The objective of Sudoku is to fill a 9-by-9 grid with integers from 1 through 9 so that each integer appears only once in each row, column, and major 3-by-3 square. The grid is partially populated with clues, and your task is to fill in the rest of the grid. There are many approaches to solving Sudoku puzzles. The approach in (b) shows an assignment LP doing the job.

The predicate `num` evaluates to true if the argument is an integer from 1 to 9. This is used for grid coordinates and for numbers to fill in the squares. `boxind` holds the numbers from 1 to 3. This is used for the coordinates of the boxes. E.g., `boxind(1,1)` is the upper-left box in the grid, while `boxind(3,3)` is the lower-right. The predicate `box` takes 4 arguments—the two coordinates of a square and the two coordinates of a box. Thus, `box(x,y,u,v)` evaluates to true if the square at `x,y` is in the box at `u,v`. E.g., `box(7,8,3,3)` is true since at row 7 and column 8 s in the lower-right box. Finally the predicate `initial` tells us how squares are filled in the initial grid state. E.g. `initial(1,1,5)` is true in the grid of the Figure.

With this at hand, we can specify the constraints:

```
# each cell receives exactly one number
model += ForAll([I,J], num(I) & num(J),
            RlpSum([X, ], num(X), fill(I, J, X)) |eq| 1)
# each number is encountered exactly once per row
model += ForAll([I,X], num(I) & num(X),
            RlpSum([J, ], num(J), fill(I, J, X)) |eq| 1)
# each number is encountered exactly once per column
model += ForAll([J,X], num(J) & num(X),
            RlpSum([I, ], num(I), fill(I, J, X)) |eq| 1)
# each number is encountered exactly once per box
model += ForAll([X,U,V], num(X) & boxind(U) & boxind(V),
         RlpSum([I, J], box(I,J,U,V), fill(I,J,X)) |eq| 1)
model += ForAll([I,J,X], num(X) & num(I) & num(J),
             fill(I,J,X) |ge| 0) # nonnegativity
model += ForAll([I,J,X], initial(I, J, X),
             fill(I, J, X) |eq| 1) # initial assignment
```

The default way to add constraints to a model is by the overloaded addition operator. Constraints can be defined through the `ForAll` function, which takes as arguments a list of query symbols `X`, a logical query `L(X,Y)`, and a parametrized arithmetic expression `R(X)` (for RLPs, an expression is a linear equality or inequality), where the query symbols appear as parameters. The semantics of `ForAll` are as follows: the answer set of the logical query `L(X,Y)` is computed and projected onto `X` (i.e., we ask the knowledge base for the tuples of `answer(X) :- L(X,Y)` with duplicate elimination). For every tuple t in answer(X), we instantiate a ground constraint with the arithmetic expression R(t). E.g., the constraint `ForAll([X],num(X),fill(1,1,X)|ge|0)` is equivalent to the ground constraints `fill(1,1,1) >= 0`,..., `fill(1,1,9) >= 0`. Constraints can also be added directly by `model += R` without `ForAll`, however, no free variables should occur in `R`. E.g. `model += fill(1,1,1) |ge| 0` is acceptable. Finally, parametrized arithmetic expressions are of the form `A rel B`, where `A` and `B` are arithmetic terms and `rel` is one of `|eq|`, `|ge|` resp. `>=`, and `|le|` resp. `<=`. A linear (in terms of the RLP decision variables) expression may contain addition of linear terms, multiplication of linear terms with a non-variable numberic predicate, or an `RlpSum`. An `RlpSum` is a first-order expression that generates a sum based on the result of a query. The syntax is similar to `ForAll`. E.g. `RlpSum([X],num(X),fill(1,1,X))` is equivalent to `fill(1,1,1) + ... + fill(1,1,9)`.

To encode the LogKB we are using pyDatalog in this example. Using loops in Python it is easy to assert the facts:

```
for u in range(1, 10): pyDatalog.assert_fact('num', u)
for u in range(1, 4): pyDatalog.assert_fact('boxind', u)
pyDatalog.assert_fact('initial', 1, 1, 5)
pyDatalog.assert_fact('initial', 2, 1, 6)
...
```

Second, we add the intensional Datalog rules:

```
pyDatalog.load(""" box(I, J, U, V) <= boxind(U) & boxind
     (V) & num(I) & num(J) & (I > (U-1)*3) & (I <= U*3) &
      (J > (V-1)*3) & (J <= V*3) """)
```

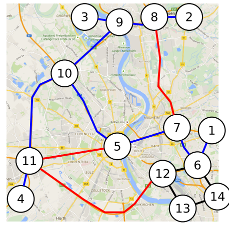
This rule defines the box predicate, which tells us if a square belongs to a box by checking if its coordinates belong to the range of the box.

Having created a logKB, we are ready to solve the relational LP and to print the solution:

```
model.solve()
sol = model.get_solution()
print "The solutions for the fill variables are:\n"
for key, value in sol.iteritems():
  if round(value,2)>=0.99: print key,"=",round(value,2)
```

This produces the following output:

```
GLPK Simplex Optimizer, v4.45
1082 rows, 729 columns, 3674 non-zeros
  0: obj =   0.000000000e+00  infeas =  3.530e+02 (353)
*446: obj =  -1.000000000e+00  infeas =  0.000e+00 (108)
OPTIMAL SOLUTION FOUND
The solutions for the fill variables are:
fill(1,4,6) = 1.0     fill(7,1,9) = 1.0    ...
```



(a) Highway ring around Cologne

$$
\begin{aligned}
\text{minimize}_{\sigma,q,v} \quad & \sum_{a\in A_L} \sum_{b\in A_F} q(b)\cdot u_L(a,b)\cdot \sigma(a) \\
\text{subject to} \quad & \forall b \in A_F: \quad 0 \le \left(v - \sum_{a\in A_L} u_F(a,b)\sigma(a) \right) \le (1-q(b))\cdot M\ , \\
& \forall a \in A_L: \quad 0 \le \sigma(a) \le 1\ , \\
& \forall b \in A_F: \quad q(b) \in \{0,1\}\ , \sum_{a\in A_L} \sigma(a) = 1, \ \sum_{b\in A_F} q(b) = 1, v \in \mathbb{R}\ .
\end{aligned}
$$

(b) Mixed integer quadratic program for Stackelberg games

Figure 2: There has been significant recent research interest in game-theoretic approaches to security at airports, ports, transportation, shipping and other infrastructure. Much of the of the recent research focused on utilizing the leader-follower Stackelberg game model. In (a) the ring highway network around the German city of Cologne is shown. There are many approaches to solving Stackelberg games. The approach in (b) shows a formulation as a mixed integer quadratic program.

We now turn to compressed optimization using a novel application of relational optimization: security games.

Lifted Optimization: Can you trick the police?

As can be seen from the Sudoku example, relational LPs share many common features with probabilistic relational models. It is thus natural to expect that some techniques developed for inference in probabilistic relational models could also apply to solving relational LPs. One such technique is lifted inference using color-passing. In Markov Random Fields, color-passing can be used as a preprocessing step to Belief Propagation, where it detects nodes that would send and receive identical messages (Kersting, Ahmadi, and Natarajan 2009; Ahmadi et al. 2013). These nodes are then grouped so that redundant computations are avoided. In linear programs, color-passing can reduce the size of the LP regardless of the solver being used (Mladenov, Ahmadi, and Kersting 2012; Noessner, Niepert, and Stuckenschmidt 2013; Grohe et al. 2014; Kersting, Mladenov, and Tokmakov 2015).

In RELOOP, color-passing is implemented by means of wrapping the highly efficient graph automorphism package Saucy (implemented in C), which uses color-passing as a heuristic for orbit computation. The interface to Saucy offers both options: either terminate Saucy after color-passing, outputting the coarsest equitable partition of the LP, or run the orbit solver fully and output the orbits under the automorphism group of the LP.

RELOOP offers two ways to access Saucy. The first is a direct wrapper that allows the user to compute coarsest equitable and orbit partitions of arbitrary colored graphs (matrices). The second, which we will discuss here, is integrated in the solver interface. Calling

```
model.solve(lifted = True)
```

will trigger RELOOP to automatically generate the necessary representation to lift the LP, pass it to Saucy, and apply the result to generate a smaller but equivalent LP. Additionally, one may supply the option "lifted_orbits = True" to lift according to the orbit partition.

In the context of RELOOP, our vision is that lifting could serve as a form of "compiler optimization" for mathematical programming. We envision situations where a user could write a simple mathematical program, which due to its simplicity contains a lot of redundancy. Removing this redundancy could require significant user effort, such as extra code to avoid computing the same patterns more than once. Instead, the lifting algorithm could do that automatically, relieving the user from that responsibility.

To illustrate this, consider to compute the optimal placement of traffic checkpoints over a road network as to maximize their coverage. Consider the network shown in Figure 2a, inspired by the ring network around the German city of Cologne. Suppose that in this network, nodes $T = \{1, 2, 3, 4\}$ act as sources and sinks of traffic. I.e., a driver might start in any one of them and her objective is to reach one of the remaining nodes of this set. On the other hand, the traffic police has a small number k of patrols, which it can position on any of the edges of the graph. The objective is to minimize the number of drivers that can avoid the control.

We model this situation as a Stackelberg game, see e.g. (Conitzer and Sandholm 2006). A Stackelberg game is typically played between a leader and a follower. The game proceeds as follows: the leading player must act first and pick an action out of a finite set A_L and commit to it. In our case, the leading player is the traffic police, and their actions are to pick a subset of k edges of the graph to position patrol on. The follower (the driver in our case) may then observe the leader's action (say, via a social traffic app like Waze) and use that knowledge for picking her own action from a set A_F. In this case, A_F is the set of all simple paths between nodes of V. Finally, the game rewards both players according their utility functions $u_L, u_F : A_L \times A_F \to \mathbb{R}$. For the purposes of this example, the follower has negative utility if she chooses a road with a checkpoint, while the police's utility is positive. Due to space constraints, we omit the details on how to construct the utility functions.

Our goal is now to compute the optimal mixed strategy for the traffic police. This problem can be cast as the Mixed Integer Quadratic Program (MIQP) shown on Figure 2b. While we will not go into the details of how this program is constructed (we refer the reader to (Pita et al. 2008) instead), we note that this program has 3 sets of variables. First, we have the leader mixed strategy, σ, which is constrained to be a probability distribution over the leader's actions. We have also binary indicator variable $q \in \{0,1\}^{|A_F|}$ ranging over

the follower's actions. Note that the constraint that the components of q sum to 1 implies that in any feasible solution exactly one component is 1 and all others are 0. Observe that if q is fixed, the problem is an LP. Finally, we have the variable v, which is a slack variable.

This mathematical program can be solved via MIQP methods or linearized to an MILP. Here we will stick to the conceptually simpler approach of (Conitzer and Sandholm 2006): generate all $|A_F|$ many feasible q's and create an LP for each of them. In doing so, we end up with a large number of LPs (relative to the size of the input graph)—recall that A_F is the set of all paths from source to sink nodes. However, as we will see, these problems end up being symmetric. To compression the LPs, we combine the disjoint subproblems in one mega-LP and run color-passing on it. Note that this mega-LP now contains $|A_F|$ many copies of the decision variable vector σ, one for each sub-problem.

We now briefly illustrate how to encode this in RELOOP; details are omitted due to space constraints. To generate the entire mega-LP in one piece, we use the following code:

```
model += RlpSum([I,J], leader_act(I) & foll_act(S),
         lead_util(I,S) * lead_strat(S,I))
model += ForAll([S,], foll_act(S),
    RlpSum([I,], lead_act(I), lead_strat(S,I))|eq|1)
model += ForAll([S, I], foll_act(S) & lead_act(I),
    lead_strat(S,I) |ge| 0)
model += ForAll([S,], foll_act(S) & foll_act(J),
    RlpSum([I,], lead_act(I), foll_util(I,J)
        * lead_strat(S,I)) |le| bound(S))
model += ForAll([S,J], foll_act(S)&foll_act(J)&Eq(S, J),
    bound(S) - RlpSum([I,], lead_act(I), foll_util(I,J)
        * lead_strat(S,I)) |le|  M)
```

Here, the logical variable S essentially index the subproblems. To elaborate further, one can see that the decision predicate `leader_strategy` has arity 2 (whereas σ is $|A_L|$-dimensional as it ranges over leader actions). This happens as we want to express all $|A_F|$ copies of σ across all subproblems in one problem. Thus, the first variable of `leader_strategy` ranges over the follower actions (since we have essentially one LP per follower action) and gives us the copy of σ relevant to the subproblem.

We also give an excerpt of the LogKB that defines the follower and leader strategies (the rules have been simplified for clarity, e.g., we omit rules that prevent cycles in paths):

```
#follower actions
path(X,Y,P) <= node(X) & node(Y) & node(Z) & adjacent(X,
     Z) & path(Z,Y,P2)
stPath(X,Y,P) <= stPair(X,Y) & path(X,Y,P)
foll_act(P) <= stPath(X,Y,P)
#leader actions
edgeSet(K,E) <= findall("innerEdge(X,Y)",X) &
     combinations(K,X,E)
lead_act(E) <= resources(R) & edgeSet(R,E)
```

Now, we can solve the resulting relational model for, say, $k = 3$ checkpoints with symmetry compression enabled. We get 102 subproblems, each with 287 decision variables and 491 constraints. The mega-LP has thus 29.274 decision variables and 50.082 constraints. Its constraint matrix has $6.069.816$ nonzero elements. Color-passing reduces the size of the LP to 5.763 variables, 11.016 constraints. The lifted constraint matrix has 643.651 nonzero elements, a 9 time reduction, *even though the input graph is asymmetric*. This renders the problem well within the scope of modern off-the-shelf commercial solvers.

Classification: Linear Program SVM

We conclude this section with a more practical example from machine learning to illustrate the runtime performance of grounding LPs. We would like to classify documents from the CORA data set using an l_∞-SVM. This problem has an LP formulation, see (Kersting, Mladenov, and Tokmakov 2015) for details about the method and the data.

```
# objective
model += -r() + c * RlpSum({I}, b_label(I, Q), slack(I))
# constraints
model += ForAll({I,Z}, b_paper(I, Z),
                label(I) * (RlpSum({X,J}, b_paper(X, J),
                      weight(X) * label(X) * kernel(Z, J)
                     ) + b()) + slack(I) >= r())
model += ForAll({X}, b_paper(X, I), weight(X) <= 1)
model += ForAll({X}, b_paper(X, I), -weight(X) <= 1)
model += r() >= 0
model += ForAll({I}, b_label(I, Z), slack(I) >= 0)
```

Our training examples consist of bag-of-words representations of the abstracts of papers. We compute an RBF kernel with Numpy for every pair and insert the resulting list of (paper, paper, kernel value) tuples into the PostgreSQL table "kernel". We grounded the model for 170, 258, 428, 840 and 1.699 papers in a virtual machine on a $3,4$GHz i7 desktop with 8GB RAM. This yielded problems with 29.921, 681.13, 184.033, 710.641, and $2.896.796$ non-zero entries in the constraint matrix. The running times were (rounded) 7s, 16s, 36s, 105s, and $460s$ with more than 90% of spent in PostgreSQL (10% for the RELOOP overhead). While this may not be the most efficient way to approach this problem, it shows that our grounder (described in the next section) adds little overhead over what is necessary to retrieve the data. It also motivates further research into a more direct integration of Numpy into the RELOOP system.

RELOOP Architecture and Algorithms

After the walkthrough of the major RELOOP features, let us now illustrate the architecture and algorithms underlying RELOOP.

The core architecture of RELOOP is sketched in Fig. 3. It consists of three major components—a LogKB, a grounder and a solver. There is no RELOOP specific language component. Instead we employ the open source symbolic algebra system Sympy[2]. From a Sympy point of view, RELOOP atoms, `RlpSums` and `ForAll` expressions are essentially uninterpreted symbolic terms. This allows us to use Sympy's expression manipulation procedures to simplify RELOOP expressions and to expand brackets, among other operations. Once `solve()` is called, the Sympy expression trees

[2]http://www.sympy.org/

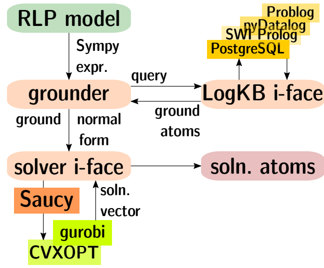


Figure 3: The RELOOP architecture.

for the model constraints and objective are passed to the grounder, where they are interpreted. The grounder takes a set of relational constraints and an objective and outputs a normal form that a solver can understand, e.g., the matrices and vectors (A, b, G, h, c) that define the linear program: $\text{minimize}_x\, c^T x \quad \text{s.t. } Ax = b, Gx \geq h$. This is where most of the work in RELOOP happens. Currently, RELOOP supports two grounding approaches: recursive and block.

Recursive grounding: This grounds every constraint in turn as follows: it first executes the query in the ForAll term, creating a new constraint for each answer; then, it grounds composite terms recursively in the same way. Consider e.g. the following RELOOP model, which could be a part of an LP computing what ingredients must be purchased as to maximize the amount of milkshake made for $5.

```
model += ForAll([X,], drink(X), RlpSum([Y,], has(X, Y),
            cost(Y)*buy(Y))) |le| 5)
model += ForAll([X, Y], drink(X) & has(X, Y),
            buy(Y) |ge| has(X,Y)*make(X))
model += ForAll([X], drink(X), make(X) |ge| 0)
```

With the LogKB (for the sake of simplicity, we assume that banana milkshake consists of only milk and bananas)

```
drink("milkshake"). has("milkshake", "bananas", 0.3).
has("milkshake", "milk", 0.7).
cost("bananas", 3). cost("milk", 1).
```

this model (after switching the first inequality) grounds to:

$$\begin{bmatrix} -3 & -1 & 0 \\ 1 & 0 & -0.3 \\ 0 & 1 & -0.7 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \texttt{buy}(\text{bananas}) \\ \texttt{buy}(\text{milk}) \\ \texttt{make}(\text{milkshake}) \end{bmatrix} \geq \begin{bmatrix} -5 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

When the recursive approach grounds the first constraint, it first materializes the query $\texttt{drink}(X)$. This returns the answer set $\{X = \text{milkshake}\}$. For each answer, it creates a new (partially grounded) constraint in which X is substituted with the answer. In our case, we get:

```
RlpSum([Y,], has(milkshake, Y), cost(Y)*buy(Y) ) |le| 5
```

It will now recursively ground the summands. 5 is already ground, thus it grounds the `RlpSum`. To do so, it queries $\texttt{has}(\text{milkshake}, Y)$. This returns the answer set $\{Y = \text{milk}, Y = \text{bananas}\}$ and generates the term

```
  cost(milkshake, milk) * buy(milk)
+ cost(milkshake, bananas) * buy(bananas)
```

The instances of the `cost` predicate can now be queried in the LogKB to get the actual coefficients. While the recursive grounder may not run fast in practice, it assumes very little from the LogKB–only that it can answer logical queries.

Block grounding: This is a more efficient alternative to recursive grounder motivated by approaches "feeding" a linear program directly from relational database systems (Mitra et al. 1995; Atamtürk et al. 2000; Farrell and Maness 2005). It grounds the constraint matrix block by block, where a block is the sub-matrix in which the rows correspond to instances of one relational constraint while the columns correspond to instances of one variable predicate. For example, consider the sub-matrix consisting of the second and third row and the third column. This is the block of the second relational constraint and the predicate `make`. One can see that the sub-matrix rows correspond to values of Y and the columns correspond to values of X. The actual entries of this sub-matrix are the values of $-Z$ in $\texttt{has}(X, Y, Z)$. Triggered by this, the block grounder puts together the query from `ForAll`, `RlpSum`, and the one from the expression inside, generating the matrix in-database. E.g. the block of the first constraint and `buy` is generated by the logical query $\texttt{q}(X, Y, -Z) \leftarrow \texttt{drink}(X) \wedge \texttt{has(X,Y,Z)}$. Then, a matrix is generated by mapping the answers for X to columns, Y to rows, and $-Z$ to values. Once all blocks are grounded, they are stacked together as Scipy sparse matrices. Note that to separate the blocks, we first flatten all levels of nested `RlpSum` statements. In practice, this grounding strategy performs many orders of magnitude faster than the recursive grounder, however, it also assumes that the LogKB is able to process algebraic expressions such as $\texttt{q}(X, Y, -Z)$.

LogKB and solver interfaces: They are intended to be templates for writing interfaces to various LogKBs and solvers. Currently, RELOOP features PostgreSQL, pyDatalog, SWI Prolog, and Problog as LogKBs as well as CVXOPT and Gurobi (and others through PICOS) as solvers.

Conclusion

We have presented RELOOP, a domain-specific language for relational optimization embedded in Python. It allows the user to describe relational optimization data, variables, objectives, and constraints in a natural syntax that follows loops, logic and linear algebra as well as can automatically reduce the dimensionality of the underlying ground model. This should help building (relational) optimization approaches and applications quickly and reliably as well as bridging the gap between statistical relational AI, probabilistic programming and numerical optimization in order to facilitate an easy transfer of results between the communities. We strongly encourage others to write their own RELOOP applications and to contribute wrappers to whatever LogKB and solver backends that fit their applications the best.

Acknowledgements The authors would like to thank the anonymous reviewers for their feedback. The work was partly supported by the DFG Collaborative Research Center SFB 876, projects A6 and B4.

References

Ahmadi, B.; Kersting, K.; Mladenov, M.; and Natarajan, S. 2013. Exploiting symmetries for scaling loopy belief propagation and relational training. *Machine Learning* 92:91–132.

Atamtürk, A.; Johnson, E.; Linderoth, J.; and Savelsbergh, M. 2000. A relational modeling system for linear and integer programming. *Operations Research* 48(6):846–857.

Cheng, X., and Roth, D. 2013. Relational inference for wikification. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 1787–1796.

Clarke, J., and Lapata, M. 2008. Global inference for sentence compression: An integer linear programming approach. *Journal of Artificial Intelligence Research* 31:399–429.

Conitzer, V., and Sandholm, T. 2006. Computing the optimal strategy to commit to. In *Proceedings 7th ACM Conference on Electronic Commerce (EC)*, 82–90.

De Raedt, L., and Kersting, K. 2010. Statistical relational learning. In C. Sammut, G. W., ed., *Encyclopedia of Machine Learning*. Heidelberg: Springer. 916–924.

De Raedt, L.; Frasconi, P.; Kersting, K.; and Muggleton, S. H., eds. 2008. *Probabilistic Inductive Logic Programming*. Springer.

De Raedt, L. 2008. *Logical and Relational Learning*. Springer.

Diamond, S., and Boyd, S. 2015. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research, MLOSS Track*. To appear.

Farrell, R., and Maness, T. 2005. A relational database approach to a linear programming-based decision support system for production planning in secondary wood product manufacturing. *Decision Support Systems* 40(2):183–196.

Fourer, R.; Gay, D.; and Kernighan, B. 2002. *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press / Brooks/Cole Publishing Company. Second Edition.

Geffner, H. 2014. Artificial intelligence: From programs to solvers. *AI Commun.* 27(1):45–51.

Getoor, L., and Taskar, B., eds. 2007. *Introduction to Statistical Relational Learning*. Cambridge, MA: MIT Press.

Grohe, M.; Kersting, K.; Mladenov, M.; and Selman, E. 2014. Dimension reduction via colour refinement. In *Proceedings of the 22th Annual European Symposium on Algorithms (ESA)*, 505–516.

Guns, T.; Nijssen, S.; and De Raedt, L. 2011. Itemset mining: A constraint programming perspective. *Artificial Intelligence* 175(12-13):1951–1983.

Kersting, K.; Ahmadi, B.; and Natarajan, S. 2009. Counting Belief Propagation. In *Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence (UAI)*.

Kersting, K.; Mladenov, M.; and Tokmakov, P. 2015. Relational linear programming. *Artificial Intelligence Journal*. In Press, Available Online.

Kordjamshidi, P.; Roth, D.; and Wu, H. 2015. Saul: Towards declarative learning based programming. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015*, 1844–1851.

Martins, A.; Smith, N.; and Xing, E. 2009. Concise integer linear programming formulations for dependency parsing. In *Proceedings of the 47th Annual Meeting of the Association for Computational Linguistics (ACL)*, 342–350.

Mitra, G.; Luca, C.; Moody, S.; and Kristjanssonl, B. 1995. Sets and indices in linear programming modelling and their integration with relational data models. *Computational Optimization and Applications* 4:263–283.

Mladenov, M.; Ahmadi, B.; and Kersting, K. 2012. Lifted linear programming. In *Proceedings of the 15th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 788–797. JMLR: W&CP volume 22.

Noessner, J.; Niepert, M.; and Stuckenschmidt, H. 2013. Rockit: Exploiting parallelism and symmetry for map inference in statistical relational models. In *Proceedings of the 27th AAAI Conference on Artificial Intelligence (AAAI)*.

Pita, J.; Jain, M.; Marecki, J.; Ordóñez, F.; Portway, C.; Tambe, M.; Western, C.; Paruchuri, P.; and Kraus, S. 2008. Deployed ARMOR protection: the application of a game theoretic model for security at the los angeles international airport. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008*, 125–132.

Riedel, S., and Clarke, J. 2006. Incremental integer linear programming for non-projective dependency parsing. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 129–137.

Riedel, S.; Smith, D.; and McCallum, A. 2012. Parse, Price and Cut–Delayed Column and Row Generation for Graph Based Parsers. In *Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, 732–743.

Rush, A., and Collins, M. 2012. A tutorial on dual decomposition and lagrangian relaxation for inference in natural language processing. *Journal of Artificial Intelligence Research* 45:305–362.

Russell, S. 2015. Unifying logic and probability. *Commun. ACM* 58(7):88–97.

Singh, S.; Rocktäschel, T.; Hewitt, L.; Naradowsky, J.; and Riedel, S. 2015. WOLFE: an nlp-friendly declarative machine learning stack. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies NAACL HLT 2015,*, 61–65.

Sra, S.; Nowozin, S.; and Wright, S., eds. 2011. *Optimization for Machine Learning*. MIT Press.

Yih, W., and Roth, D. 2007. Global inference for entity and relation identification via a linear programming formulation. In Getoor, L., and Taskar, B., eds., *An Introduction to Statistical Relational Learning*. MIT Press.