Automated Reasoning for Relational Probabilistic Knowledge Representation*

Christoph Beierle[1], Marc Finthammer[1],
Gabriele Kern-Isberner[2], Matthias Thimm[2]

[1]Dept. of Computer Science, FernUniversität in Hagen, 58084 Hagen, Germany
[2]Dept. of Computer Science, TU Dortmund, 44221 Dortmund, Germany

Abstract. KReator is a toolbox for representing, learning, and automated reasoning with various approaches combining relational first-order logic with probabilities. We give a brief overview of the KReator system and its automated reasoning facilities.

1 Introduction

Approaches combining logic with probabilities for representing uncertain information are typically based upon propositional logic (see, e.g., [8]). Various extensions to a first-order setting like Bayesian logic programs [3, Ch. 10] or Markov logic networks [3, Ch. 12] have been proposed. In order to promote the use as well as the evaluation and comparison of such proposals, KReator provides a common and easy-to-use interface for representing, learning, and automated reasoning with different relational probabilistic approaches. In this paper, we give a brief description of the KReator system and its background and usage. In Sec. 2, we start with recalling the concept of Bayesian logic programs and Markov logic networks and sketch the relational maximum entropy framework RME. Section 3 gives an overview on the KReator system and presents a system walk-through with several examples, while Sec. 4 outlines its system architecture and implementation and points out further work.

2 Background

Bayesian logic programming combines logic programming and Bayesian networks [3, Ch. 10]. The basic structure for knowledge representation in Bayesian logic programs are *Bayesian clauses* like $(alarm(\mathsf{X}) \mid lives\_in(\mathsf{X},\mathsf{Y}), tornado(\mathsf{Y}))$ which model probabilistic dependencies between Bayesian atoms. A function cpd_c for a Bayesian clause c expresses the conditional probability distribution $P(\mathsf{head}(c) \mid \mathsf{body}(c))$ and thus partially describes an underlying probability distribution P. In order to aggregate probabilities that arise from applications of different Bayesian clauses with the same head, BLPs make use of *combining*

* The research reported here was partially supported by the Deutsche Forschungsgemeinschaft (grants BE 1700/7-1 and KE 1413/2-1).

rules. Semantics are given to Bayesian logic programs via transformation into propositional forms, i. e. into Bayesian networks [8] (see [3, Ch. 10] for details).

Markov logic [3, Ch. 12] establishes a framework which combines Markov networks [8] with first-order logic to handle a broad area of statistical relational learning tasks. The Markov logic syntax complies with first-order logic where each formula is quantified by an additional weight value, e.g., $(\mathit{lives\_in}(x, y) \wedge \mathit{tornado}(y) \Rightarrow \mathit{alarm}(x), 2.2)$. Semantics are given to sets of Markov logic formulas by a probability distribution over propositional possible worlds that is calculated as a log-linear model over weighted ground formulas. The fundamental idea in Markov logic is that first-order formulas are not handled as hard constraints but each formula is more or less softened depending on its weight. A *Markov logic network (MLN)* L is a set of weighted first-order logic formulas F_i together with a set of constants C. The semantics of L is given by a ground Markov network $M_{L,C}$ constructed from F_i and C [3, Ch. 12]. The standard semantics of Markov networks [8] is used for reasoning, e.g. to determine the consequences of L (see [3, Ch. 12] for details).

The basic idea of the *relational maximum entropy* framework (RME) [2, 6, 11]. is to make use of propositional maximum entropy techniques [7, 4, 9] after grounding the knowledge base appropriately. The *entropy* H is an information-theoretic measure on probability distributions and is defined as a weighted sum on the information encoded in every possible world $\omega \in \Omega$: $H(P) = -\sum_{\omega \in \Omega} P(\omega) \log P(\omega)$. By employing the *principle of maximum entropy* one can determine the single probability distribution that is the optimal model for a consistent knowledge base $\mathcal{R}$ in an information-theoretic sense: $P^{ME}_{\mathcal{R}} = \arg\max_{P \models \mathcal{R}} \mathcal{H}(P)$. However, this depends crucially on $\mathcal{R}$ being consistent, for otherwise no model of $\mathcal{R}$ exists, let alone models with maximum entropy. Since groundings may introduce non-trivial conflicts, this problem is all the more difficult in a first-order context with free variables where one has probabilistic first-order clauses like $(\mathit{alarm}(\mathsf{X}) \mid \mathit{lives\_in}(\mathsf{X}, \mathsf{Y}), \mathit{tornado}(\mathsf{Y}))[0.9]$, specifying that the conditional probability of $\mathit{alarm}(\mathsf{X})$ given $\mathit{lives\_in}(\mathsf{X}, \mathsf{Y})$ and $\mathit{tornado}(\mathsf{Y})$ ought to be 0.9. The RME inference process can be divided into three steps: (1) ground the knowledge base $\mathcal{R}$ with a grounding operator $\mathcal{G}$, (2) calculate the probability distribution $P^{ME}_{\mathcal{G}(\mathcal{R})}$ with maximum entropy for the grounded instance $\mathcal{G}(\mathcal{R})$, and (3) determine the probabilistic implications of $P^{ME}_{\mathcal{G}(\mathcal{R})}$ [2, 6, 11].

3 Examples and System Overview

The KReator system provides automated reasoning facilities for all three probabilistic relational frameworks sketched in Sec. 2. As an illustration, we consider the well-known burglary example given in [8] where we have some (uncertain) beliefs about the relationships between burglaries, types of neighborhoods, natural disasters, and alarms. This example could be represented by a BLP containig the three Bayesian clauses

$c_1 : (\mathit{alarm}(\mathsf{X}) \mid \mathit{burglary}(\mathsf{X}))$ $\qquad$ $c_3 : (\mathit{burglary}(\mathsf{X}) \mid \mathit{nhood}(\mathsf{X}))$

$c_2 : (\mathit{alarm}(\mathsf{X}) \mid \mathit{lives\_in}(\mathsf{X}, \mathsf{Y}), \mathit{tornado}(\mathsf{Y}))$

together with corresponding conditional probability distributions cpd_{c_i}. For instance, $\mathsf{cpd}_{c_2}(\mathsf{true},\mathsf{true},\mathsf{true}) = 0.9$ would express our subjective belief that *alarm* is true with probability 0.9 if $\mathit{lives\_in}(\mathsf{X},\mathsf{Y})$ and $\mathit{tornado}(\mathsf{Y})$ are true. In this BLP modelling, *nhood* is a multi-valued unary predicate, while in the modellings using MLNs and RME, *nhood* will be a binary-valued two-place predicate.

Using the Alchemy syntax [5] for MLN files, a MLN knowledge base for the burglary example is given by the following five weighted clauses:

$$\begin{array}{llll}
2.2\ \mathit{burglary}(x) & => \mathit{alarm}(x) & -0.8\ \mathit{nhood}(x, \mathit{Good}) & => \mathit{burglary}(x) \\
2.2\ \mathit{lives\_in}(x,y) \wedge \mathit{tornado}(y) & => \mathit{alarm}(x) & -0.4\ \mathit{nhood}(x, \mathit{Average}) & => \mathit{burglary}(x) \\
 & & 0.4\ \mathit{nhood}(x, \mathit{Bad}) & => \mathit{burglary}(x)
\end{array}$$

Note that, in contrast to BLPs and RMEs, MLNs do not support conditional probabilities, so the rule-like knowledge has to be modeled as material implications. Modeling our running example in RME can be done by

$$\begin{array}{ll}
c_1\colon (\mathit{alarm}(\mathsf{X}) \,|\, \mathit{burglary}(\mathsf{X}))[0.9] & c_4\colon (\mathit{burglary}(\mathsf{X}) \,|\, \mathit{nhood}(\mathsf{X}, \mathit{average}))[0.4] \\
c_2\colon (\mathit{alarm}(\mathsf{X}) \,|\, \mathit{lives\_in}(\mathsf{X},\mathsf{Y}), & c_5\colon (\mathit{burglary}(\mathsf{X}) \,|\, \mathit{nhood}(\mathsf{X}, \mathit{good}))[0.3] \\
\qquad\qquad \mathit{tornado}(\mathsf{Y}))[0.9] & c_6\colon (\mathit{nhood}(\mathsf{X},\mathsf{Z}) \,|\, \mathit{nhood}(\mathsf{X},\mathsf{Y}))[0.0]\ [\mathsf{Y} \neq \mathsf{Z}] \\
c_3\colon (\mathit{burglary}(\mathsf{X}) \,|\, \mathit{nhood}(\mathsf{X}, \mathit{bad}))[0.6] & c_7\colon (\mathit{lives\_in}(\mathsf{X},\mathsf{Z}) \,|\, \mathit{lives\_in}(\mathsf{X},\mathsf{Y}))[0.0]\ [\mathsf{Y} \neq \mathsf{Z}]
\end{array}$$

where in this knowledge base, the conditionals c_6 and c_7 ensure mutual exclusion of the states for literals of "*nhood*" and "*lives_in*".

A query in so-called *unified syntax* can be answered by KREATOR with respect to a BLP, MLN, or RME knowledge base. This query syntax abstracts from the respective syntax which is necessary to address a "native" query to a BLP, MLN, or RME knowledge base. The idea behind this functionality is that while some knowledge can be modeled in different knowledge representation approaches, the user is able to compare the reasoning facilities of these approaches in a direct way by formulating appropriate queries in unified syntax, passing them to the

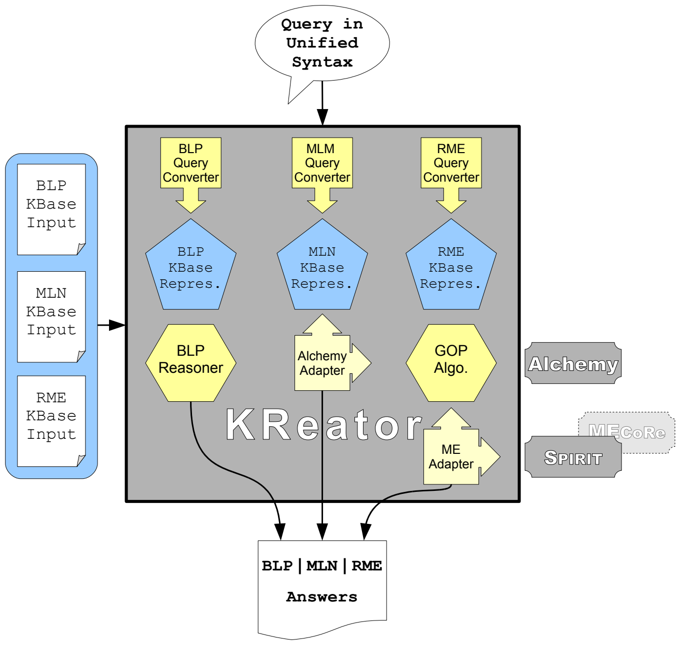


Fig. 1: Processing query in unified syntax

different knowledge bases, and analyzing the different answers. A KReator query in unified syntax consists of two parts: In the *head* of the query there are one or more ground atoms whose probabilities shall be determined. The *body* of the query is composed of several evidence atoms. For each supported knowledge representation formalism, KReator must convert a query in unified syntax in the exact syntax required by the respective inference engine. Among other things, this includes e. g. the conversion from lower case constants to upper case ones (and variables, vice versa), as required by the Alchemy tool for processing MLNs. KReator also converts the respective output results to present them in a standardized format to the user (cf. Fig. 1).

In addition to a knowledge base, that typically contains only general generic knowledge, also evidential knowledge like

$$\begin{array}{l} \mathit{lives\_in}(\mathit{james}, \mathit{yorkshire}), \mathit{lives\_in}(\mathit{carl}, \mathit{austin}), \mathit{burglary}(\mathit{james}), \\ \mathit{tornado}(\mathit{austin}), \mathit{nhood}(\mathit{james}) = \mathsf{average}, \mathit{nhood}(\mathit{carl}) = \mathsf{good} \end{array}$$

can be taken into account when reasoning with KReator. The following table shows three queries and their respective probabilities inferred by KReator from each of the example knowledge bases and the evidence given above:

	BLP	MLN	RME
alarm(*james*)	0.900	0.896	0.918
alarm(*carl*)	0.550	0.900	0.880
burglary(*carl*)	0.300	0.254	0.362

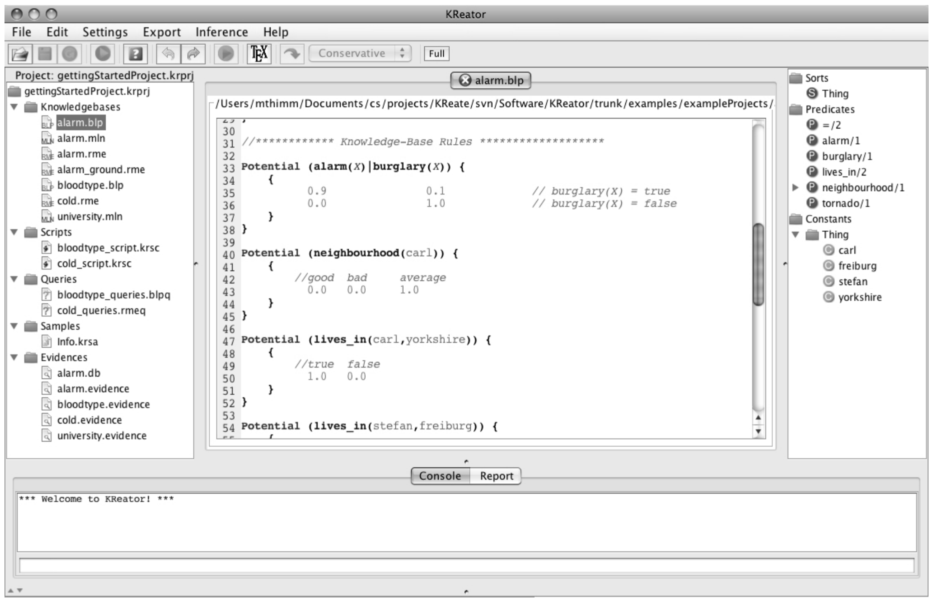
The inferred probabilities are are not identical since the same generic knowledge is modelled slightly differently in the three formalisms. For instance in BLPs, specific information resides in the combinig rules (in this example, noisy-or was used) that aggregate probabilities of different clauses with the same head.

Doing further computations in the different formalisms is conveniently supported by KReator. For example, dropping *tornado*(*austin*) from the evidence yields, as expected, the values for the query *alarm*(*james*) as given in the table above; whereas the values for *alarm*(*carl*) drop dramatically. Replacing *burglary*(*james*) by *alarm*(*james*) in the evidence and asking for *burglary*(*james*) yields 0.400 (BLP), 0.411 (MLN), and 0.507 (RME).

All specification and reasoning steps involved in these examples are conveniently supported by the KReator system. KReator comes with a graphical user interface and an integrated console-based interface. The main view of KReator (see Fig. 2) is divided into the menu, a toolbar and four main panels: the project, editor, outline, and console panel.

KReator structures its data into projects which may contain knowledge bases, scripts written in KReatorScript (see below), query collections for knowledge bases, and sample/evidence files. Although all types of files can be opened independently in KReator, projects can help the knowledge engineer to organize his work. The *project panel* (upper left in Fig. 2) gives a complete overview on the project the user is currently working on.

All files supported by KReator can be viewed and edited in the *editor panel* (upper middle in Fig. 2). Multiple files can be opened at the same time and the



```
//************ Knowledge-Base Rules ******************

Potential (alarm(X)|burglary(X)) {
    {
        0.9             0.1             // burglary(X) = true
        0.0             1.0             // burglary(X) = false
    }
}

Potential (neighbourhood(carl)) {
    {
        //good  bad     average
          0.0   0.0     1.0
    }
}

Potential (lives_in(carl,yorkshire)) {
    {
        //true  false
          1.0   0.0
    }
}

Potential (lives_in(stefan,freiburg)) {
```

```
*** Welcome to KReator! ***
```

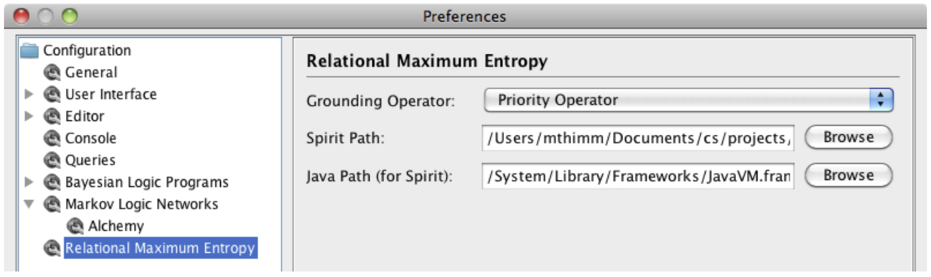
Fig. 2: KReator – Main window

editor supports editing knowledge bases and the like with syntax-highlighting, syntax check, and other features normally known from development environments for programming languages.

The *outline panel* (upper right in Fig. 2) gives an overview on the currently viewed file in the editor panel. If the file is a knowledge base the outline shows information on the logical components of the knowledge base, such as used predicates (and, in case of BLPs, their states), constants, and sorts (if the knowledge base uses a typed language).

The *console panel* (bottom in Fig. 2) contains two tabs, one with the actual console interface and one with the *report*. The console can be used to access all KReator functionality just using textual commands, e. g. querying knowledge bases, open and saving file, and so on. The console is a live interpreter for KReatorScript which can also be used for writing scripts that allows the knowledge engineer to save recurring tasks. By doing so results can be verified and sophisticated queries can be addressed to different knowledge bases with little adaptations. As a further ease, every action executed in KReator, e. g. opening a file in the graphical user interface or querying a knowledge base from the console, is recorded as a KReatorScript command in the report. The whole report or parts of it can easily be saved as script file and executed again when experiments have to be repeated and results have to be reproduced.

KReator is highly configurable and extensible. Figure 3 shows the preferences dialog which enables the configuration of nearly every property of the graphical user interface as well as special features of the different knowledge



Relational Maximum Entropy

Grounding Operator: Priority Operator

Spirit Path: /Users/mthimm/Documents/cs/projects,

Java Path (for Spirit): /System/Library/Frameworks/JavaVM.fran

Fig. 3: Preferences in KReator

representation formalism. Furthermore, every property can also be modified using KReatorScript so that different configurations can be tested easily in script files. Due to the open architecture, KReator can be extended effortlessly by further formalisms and most of the features like querying are automatically enabled.

4 System Architecture and Implementation

The implementation of KReator is done in Java and mirrors its objective to support different approaches to relational probabilistic knowledge representation and reasoning. It strictly separates the internal logic and the user interface, employing an abstract command structure allowing easy modifications on both sides. In order to support the implementation of other approaches, KReator features a large library on first-order logic and basic probabilistic methods. Among others this library contains classes for formulæ, rules, conditionals and various methods to operate on these. There is also a rudimentary implementation of Prolog available that can be used for specifying background knowledge as e. g. in BLPs. This integrated library is designed to support a fast implementation of specific approaches to statistical relational learning. The task of integrating a new approach into the KReator system is supported by a small set of interfaces that have to be implemented in order to be able to access the new approach from the user interface. There are interfaces for knowledge bases (which demands e. g. support for querying), file writers and parsers (for reading and writing the specific syntax of an approach), and learner. One thing to note is that both file writers and parsers have to work on strings only, all the cumbersome overhead of file operations and I/O is handled by KReator. With the help of a plugin-like architecture the developer of a new approach only has to be concerned with connecting her approach to KReator using these interfaces. Then all the benefits of an integrated development environment as provided by KReator are immediately accessible. Currently, KReator supports knowledge representation using BLPs, MLNs, and the relational maximum entropy approach RME; other formalisms will be integrated in the near future.

Performing inference on MLNs is done using the Alchemy software package [5], a console-based tool for processing Markov logic networks. For BLPs, a reasoning component was implemented within KReator. To process ground RME knowledge bases, KReator uses a so-called ME-adapter to communicate with a MaxEnt-reasoner. Currently, such adapters are supplied for the SPIRIT reasoner [10] and for MECore [1] which are tools for processing (propositional) conditional probabilistic knowledge bases using maximum entropy methods.

Ongoing work includes integration of different learning algorithms. Due to the availability of several formalisms in KReator these algorithms can be implemented in a very general manner and employed by the formalisms in an easy way. Future work consists of extending the support for other relational probabilistic formalisms, such as Probabilistic Relational Models [3, Ch. 5].

References

1. M. Finthammer, C. Beierle, B. Berger, and G. Kern-Isberner. Probabilistic reasoning at optimum entropy with the MEcore system. In H. C. Lane and H. W. Guesgen, editors, *Proceedings 22nd International FLAIRS Conference, FLAIRS'09*. AAAI Press, Menlo Park, California, 2009.
2. M. Finthammer, S. Loh, and M. Thimm. Towards a toolbox for relational probabilistic knowledge representation, reasoning, and learning. In *Relational Approaches to Knowledge Representation and Learning. Workshop at KI-2009, Paderborn, Germany*, Informatik-Bericht 354, pages 34–48. FernUniv. in Hagen, 2009.
3. L. Getoor and B. Taskar, editors. *Introduction to Statistical Relational Learning*. MIT Press, 2007.
4. G. Kern-Isberner. Characterizing the principle of minimum cross-entropy within a conditional-logical framework. *Artificial Intelligence*, 98:169–208, 1998.
5. S. Kok, P. Singla, M. Richardson, P. Domingos, M. Sumner, H. Poon, D. Lowd, and J. Wang. *The Alchemy System for Statistical Relational AI: User Manual*. Department of Computer Science and Engineering, University of Washington, 2008.
6. S. Loh, M. Thimm, and G. Kern-Isberner. On the problem of grounding a relational probabilistic conditional knowledge base. In *Proceedings of the 14th International Workshop on Non-Monotonic Reasoning (NMR'10)*, Toronto, Canada, May 2010.
7. J.B. Paris. *The uncertain reasoner's companion – A mathematical perspective*. Cambridge University Press, 1994.
8. J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1998.
9. W. Rödder and C.-H. Meyer. Coherent knowledge processing at maximum entropy by SPIRIT. In E. Horvitz and F. Jensen, editors, *Proceedings 12th Conference on Uncertainty in Artificial Intelligence*, pages 470–476, San Francisco, Ca., 1996. Morgan Kaufmann.
10. W. Rödder, E. Reucher, and F. Kulmann. Features of the expert-system-shell SPIRIT. *Logic Journal of the IGPL*, 14(3):483–500, 2006.
11. M. Thimm, M. Finthammer S. Loh, G. Kern-Isberner, and C. Beierle. A system for relational probabilistic reasoning on maximum entropy. In *Proceedings 23rd International FLAIRS Conference, FLAIRS'10*. AAAI Press, Menlo Park, California, 2010. (to appear).