draft – 2009-08-03

Towards a Toolbox for Relational Probabilistic Knowledge Representation, Reasoning, and Learning

Marc Finthammer[1], Sebastian Loh[2], and Matthias Thimm[2]

[1] Department of Computer Science, FernUniversität in Hagen, Germany
[2] Department of Computer Science, Technische Universität Dortmund, Germany

Abstract. This paper presents KReator, a versatile and easy-to-use toolbox for statistical relational learning currently under development. The research on combining probabilistic models and first-order theory put forth a lot of different approaches in the past few years. While every approach has advantages and disadvantages the variety of prototypical implementations make thorough comparisons of different approaches difficult. KReator aims at providing a common and simple interface for representing, reasoning, and learning with different relational probabilistic approaches. We give an overview on the system architecture of KReator and illustrate its usage.

1 Introduction

Probabilistic inductive logic programming (or *statistical relational learning*) is a very active field in research at the intersection of logic, probability theory, and machine learning, see [1, 2] for some excellent overviews. This area investigates methods for representing probabilistic information in a relational context for both reasoning and learning. Many researchers developed liftings of propositional probabilistic models to the first-order case in order to take advantage of methods and algorithms already developed. Among these are the well-known Bayesian logic programs [3] and Markov logic networks [4] which extend respectively Bayes nets and Markov nets [5] and are based on knowledge-based model construction [6]. Other approaches also employ Bayes nets for their theoretical foundation like logical Bayesian networks [7] and relational Bayesian networks [8]; or they are influenced by other fields of research like probabilistic relational models [9] by database theory and P-log [10] by answer set programming. There are also some few approaches to apply maximum entropy methods to the relational case [11, 12]. But because of this variety of approaches and the absence of a common interface there are only few comparisons of different approaches, see for example [13, 14].

In this paper we describe the KReator toolbox, a versatile integrated development environment for knowledge engineering in the field of statistical relational learning. KReator is currently under development and part of the ongoing KReate project[3] which aims at developing a common methodology for learning, modelling and inference in a relational probabilistic framework. As statistical relational learning is a (relatively) young research area there are many different proposals for integrating probability theory in first-order logic, some of them mentioned above. Although many researchers have implementations of their approaches available, most of these implementations are prototypical, and in order to compare different approaches one has to learn the usage of different tools. KReator aims at providing a common interface for different approaches to statistical relational learning and to support the researcher and knowledge engineer in developing knowledge bases and using them in a common and easy-to-use fashion. Currently, the development of KReator is still in a very early stage but already supports Bayesian logic programs, Markov logic networks, and in particular a new approach for using maximum entropy methods in a relational context [11].

The rest of this paper is organized as follows. In Sec. 2 we give an overview on the approaches of statistical relational learning that are currently supported by KReator, i. e. Bayesian logic programs, Markov logic networks, and the relational maximum entropy approach. We go on in Sec. 3 with presenting the system architecture of KReator and motivate the main design choices. In Sec. 4 we give a short manual-style overview on the usage of KReator and in Sec. 5 we give some hints on future work and conclude.

2 Relational Probabilistic Knowledge Representation

In the following we give some brief overview on frameworks for relational probabilistic reasoning that are already implemented in KReator. These are Bayesian logic programs originally due to Kersting et. al. [3], Markov logic networks originally due to Domingos et. al. [4], and a framework employing reasoning with maximum entropy methods that is currently in development [11]. We illustrate the use of these frameworks on a common example, the well-known burglary example [5, 1].

Example 1. We consider a scenario where someone—let's call him *James*—is on the road and gets a call from his neighbor saying that the alarm of James' house is ringing. James has some uncertain beliefs about the relationships between burglaries, types of neighborhoods, natural disasters, and alarms. For example, he knows that if there is a tornado warning for his home place, then the probability of a tornado triggering the alarm of his house is 0.9. A reasonable information to infer from his beliefs and the given information is "What is the probability of an actual burglary?".

2.1 Bayesian Logic Programs

Bayesian logic programming is an approach to combine logic programming and Bayesian networks [3]. Bayesian logic programs (BLPs) use a standard logic

[3] `http://www.fernuni-hagen.de/wbs/research/kreate/index.html`

programming language and attach to each logical clause a set of probabilities, which define a conditional probability distribution of the head of the clause given specific instantiations of the body of the clause.
In contrast to first-order logic, BLPs employ an extended form of predicates and atoms. In BLPs, *Bayesian predicates* are predicates that feature an arbitrary set as possible states that are not necessarily the boolean values $\{\mathsf{true}, \mathsf{false}\}$. For example, the Bayesian predicate *bloodtype/1* may represent the blood type of a person using the possible states $S(\mathit{bloodtype}) = \{a, b, ab, 0\}$ [3]. Analogously to first-order logic, Bayesian predicates can be instantiated to *Bayesian atoms* using constants and variables and then each ground Bayesian atom represents a single random variable. If A is a Bayesian atom of the Bayesian predicate p we set $S(A) = S(p)$.

Definition 1 (Bayesian Clause, Conditional Probability Distribution). *A* Bayesian clause *c is an expression $(H \mid B_1, \ldots, B_n)$ with Bayesian atoms $H, B_1, \ldots, B_n$. With a Bayesian clause c with the form $(H \mid B_1, \ldots, B_n)$ we associate a function $\mathsf{cpd}_c : S(H) \times S(B_1) \times \ldots \times S(B_n) \to [0, 1]$ that fulfills*

$$\forall b_1 \in S(B_1), \ldots, b_n \in S(B_n) : \sum_{h \in S(H)} \mathsf{cpd}_c(h, b_1, \ldots, b_n) = 1 \quad .$$

We call cpd_c a conditional probability distribution*. Let CPD_p denote the set of all conditional probability distributions $\{\mathsf{cpd}_{H|B_1,\ldots B_n} \mid H$ is an atom of $p\}$.*

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.

Example 2. We represent Ex. 1 as a set $\{c_1, c_2, c_3\}$ of Bayesian clauses with

$$\begin{aligned}
c_1 : &\quad (\mathit{alarm}(\mathsf{X}) \mid \mathit{burglary}(\mathsf{X}))\\
c_2 : &\quad (\mathit{alarm}(\mathsf{X}) \mid \mathit{lives\_in}(\mathsf{X}, \mathsf{Y}), \mathit{tornado}(\mathsf{Y}))\\
c_3 : &\quad (\mathit{burglary}(\mathsf{X}) \mid \mathit{neighborhood}(\mathsf{X}))
\end{aligned}$$

where $S(\mathit{tornado/1}) = S(\mathit{lives\_in/2}) = S(\mathit{alarm}) = S(\mathit{burglary}) = \{\mathsf{true}, \mathsf{false}\}$ and $S(\mathit{neighborhood}) = \{\mathsf{good}, \mathsf{average}, \mathsf{bad}\}$. For each Bayesian clause c_i, we define a function cpd_{c_i} which expresses our subjective beliefs, e. g., for clause c_2 we define

$$\begin{aligned}
\mathsf{cpd}_{c_2}(\mathsf{true}, \mathsf{true}, \mathsf{true}) &= 0.9 & \mathsf{cpd}_{c_2}(\mathsf{true}, \mathsf{true}, \mathsf{false}) &= 0.01\\
\mathsf{cpd}_{c_2}(\mathsf{true}, \mathsf{false}, \mathsf{true}) &= 0 & \mathsf{cpd}_{c_2}(\mathsf{true}, \mathsf{false}, \mathsf{false}) &= 0\\
\mathsf{cpd}_{c_2}(\mathsf{false}, \mathsf{true}, \mathsf{true}) &= 0.1 & \mathsf{cpd}_{c_2}(\mathsf{false}, \mathsf{true}, \mathsf{false}) &= 0.99\\
\mathsf{cpd}_{c_2}(\mathsf{false}, \mathsf{false}, \mathsf{true}) &= 1 & \mathsf{cpd}_{c_2}(\mathsf{false}, \mathsf{false}, \mathsf{false}) &= 1
\end{aligned}$$

Considering clauses c_1 and c_2 in Ex. 2 one can see that it is possible to have multiple clauses with the same head. BLPs facilitate *combining rules* in order to aggregate probabilities that arise from applications of different Bayesian clauses. A combining rule cr_p for a Bayesian predicate p/n is a function cr_p :

$\mathfrak{P}(\mathsf{CPD}_p) \rightarrow \mathsf{CPD}_p$ that assigns to the conditional probability distributions of a set of Bayesian clauses a new conditional probability distribution that represents the *joint* probability distribution obtained from aggregating the given clauses[4]. For example, given clauses $c_1 = (b(X) \mid a_1(X))$ and $c_2 = (b(X) \mid a_2(X))$ the result $f = \mathsf{cr}_b(\{\mathsf{cpd}_{c_1}, \mathsf{cpd}_{c_2}\})$ of the combining rule cr_b is a function $f : S(b) \times S(a_1) \times S(a_2) \rightarrow [0,1]$. Appropriate choices for such functions are *average* or *noisy-or*, cf. [3].

Example 3. We continue Ex. 2. Suppose *noisy-or* to be the combining rule for *alarm*. Then the joint conditional probability distribution $\mathsf{cpd}_{c'}$ for $c' = (\mathit{alarm}(\mathsf{X}) \mid \mathit{burglary}(\mathsf{X}), \mathit{lives\_in}(\mathsf{X}, \mathsf{Y}), \mathit{tornado}(\mathsf{Y}))$ can be computed via

$$\mathsf{cpd}_{c'}(t_1, t_2, t_3, t_4) = Z * (1 - (1 - \mathsf{cpd}_{c_1}(t_1, t_2)) * (1 - \mathsf{cpd}_{c_2}(t_1, t_3, t_4)))$$

for any $t_1, t_2, t_3, t_4 \in \{\mathsf{true}, \mathsf{false}\}$. Here, Z is a normalizing constant for maintaining the property of conditional probability distributions to sum up to one for any specific head value.

Now we are able to define Bayesian logic programs as follows.

Definition 2 (Bayesian Logic Program). *A* Bayesian logic program B *is a tuple* $B = (C, D, R)$ *with a (finite) set of Bayesian clauses* $C = \{c_1, \ldots, c_n\}$*, a set of conditional probability distributions (one for each clause in* C*)* $D = \{\mathsf{cpd}_{c_1}, \ldots, \mathsf{cpd}_{c_n}\}$*, and a set of combining functions (one for each Bayesian predicate appearing in* C*)* $R = \{\mathsf{cr}_{p_1}, \ldots, \mathsf{cr}_{p_m}\}$*.*

Semantics are given to Bayesian logic programs via transformation into the propositional case, i. e. into Bayesian networks [5]. Given a specific (finite) universe U a Bayesian network BN can be constructed by introducing a node for every grounded Bayesian atom in B. Using the conditional probability distributions of the grounded clauses and the combining rules of B a (joint) conditional probability distribution can be specified for any node in BN. If BN is acyclic this transformation uniquely determines a probability distribution P on the grounded Bayesian atoms of B which can be used to answer queries.

A detailed description of the above (declarative) semantics and an equivalent procedural semantics which is based on SLD resolution are given in [3].

2.2 Markov Logic Networks

Markov logic [4] establishes a framework which combines Markov networks [5] with first-order logic to handle a broad area of statistical relational learning tasks. The Markov logic syntax complies with first-order logic[5], however each formula is quantified by an additional weight value. The semantics of a set of Markov logic formulas is explained by a probability distribution over possible worlds. A possible world assigns a truth value to every possible ground atom

[4] $\mathfrak{P}(S)$ denotes the power set of a set S.

[5] Although Markov logic also covers functions, we will omit this fact, and only consider constants.

(constructible from the set of predicates and the set of constants). The probability distribution 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. Instead, each formula is more or less softened depending on its weight. So a possible world *may* violate a formula without necessarily receiving a zero probability. Rather a world is more probable, the less formulas it violates. A formula's weight specifies how strong the formula is, i. e. how much the formula influences the probability of a satisfying world versus a violating world. This way, the weights of all formulas influence the determination of a possible world's probability in a complex manner. One clear advantage of this approach is that Markov logic can directly handle contradictions in a knowledge base, since the (contradictious) formulas are weighted against each other anyway. Furthermore, by assigning appropriately high weight values to certain formulas, it can be enforced that these formulas will be handled as hard constraints, i. e. any world violating such a hard formula will have a zero probability. Thus, Markov logic also allows the processing of purely logical first-order formulas.

Definition 3 (Markov logic network). *A* Markov logic network (MLN) L *is a set of first-order logic formulas F_i, where each formula F_i is quantified by a real value w_i, its* weight. *Together with a set of constants C it defines a* Markov network $M_{L,C}$ *as follows:*

- *$M_{L,C}$ contains a node for each possible grounding of each predicate appearing in L.*
- *$M_{L,C}$ contains an edge between two nodes (i. e. ground atoms) iff the ground atoms appear together in at least one grounding of one formula in L.*
- *$M_{L,C}$ contains one feature (function) for each possible grounding of each formula F_i in L. The value of the feature for a possible world x is 1, if the ground formula is true for x (and 0 otherwise). Each feature is weighted by the weight w_i of its respecting formula F_i.*

According to the above definition, a MLN (i. e. the weighted formulas) defines a template for constructing *ground Markov networks*. For a different set C' of constants, a different ground Markov network $M_{L,C'}$ emerges from L. These ground Markov networks may vary in size, but their general structure is quite similar, e. g. the groundings of a formula F_i have the weight w_i in any ground Markov network of L. The ground Markov network $M_{L,C}$ specifies

$$P(X = x) = \frac{1}{Z} \exp\left(\sum_i w_i n_i(x)\right)$$

as the probability distribution over possible worlds x (whereas Z is a normalization factor). For each formula F_i, the binary result values of its feature functions have been incorporated into the counting function $n_i(x)$, so $n_i(x)$ compactly expresses the number of true groundings of F_i in the possible world x.

Example 4. In the following example, we model the relations described in Ex. 1 as a MLN (using the Alchemy syntax [15] for MLN files). The "!" operator used in the predicate declarations of *lives_in* and *neighborhood* enforces that the respective variables will have mutually exclusive and exhaustive values, i. e. that every person lives in exactly one town and one neighborhood (in terms of ground atoms). The weights of the formulas are to be understood exemplary (since "realistic" weights cannot be estimated just like that, but requires learning from data). We declare the typed predicates $alarm(person)$, $neighborhood(person, hood\_state!)$, $lives\_in(person, town!)$, $burglary(person)$, the types and constants $person = \{James, Carl\}$, $town = \{Freiburg, Yorkshire, Austin\}$, $hood\_state = \{Bad, Average, Good\}$, and add the following weighted formulas:

$$\begin{array}{rll} 2.2 & burglary(x) & => alarm(x) \\ 2.2 & lives\_in(x, y) \land tornado(y) & => alarm(x) \\ -0.8 & neighborhood(x, Good) & => burglary(x) \\ -0.4 & neighborhood(x, Average) & => burglary(x) \\ 0.4 & neighborhood(x, Bad) & => burglary(x) \end{array}$$

2.3 Relational Maximum Entropy

In this paper we also consider a specific approach for reasoning under maximum entropy on first-order probabilistic conditional logic [11]. The approach of relational maximum entropy (RME) employed in this paper is inspired by the work of Lukasiewicz and Kern-Isberner [16] but combines relational representation and reasoning under maximum entropy in a new manner, see below for more details.

Knowledge is captured in RME using probabilistic conditionals as in probabilistic conditional logic, cf. [17].

Definition 4 (RME conditional). *A RME conditional $r = (\phi \mid \psi)[\alpha][c]$ consists of a head literal ϕ, a list of n body literals $\psi = \psi_1, \ldots, \psi_n$, a real value $\alpha \in [0, 1]$, and a list of* meta-constraints *$c = c_1, \ldots, c_m$, which allows the restriction of the substitution for certain variables. A meta-constraint is either an expression of the form $X \neq Y$ or $X \notin \{k_1, \ldots, k_l\}$, with variables X, Y and $\{k_1, \ldots, k_l\} \subseteq U$. A conditional r is called* ground *iff r contains no variables. The set of all RME conditionals is determined as the language $(\mathcal{L} \mid \mathcal{L})^{rel}$ and the set of all ground conditionals is referred to by $(\mathcal{L} \mid \mathcal{L})^{rel}_U$.*

Definition 5 (RME knowledge base). *A RME knowledge base KB is a quadruple $KB = (S, U, P, R)$ with a finite set of* sorts *S, a finite set of constants U, a finite set of predicates P, and a finite set of RME conditionals R. Any constant of the* universe *U is associated with one sort in S and U_σ determines the constants with sort σ. Each argument of a predicate $p(\sigma_1, \ldots, \sigma_k) \in P$ is also associated with a sort in S, $\sigma_i \in S, 1 \leq i \leq k$. Furthermore, all variables $Var(R)$ occurring in R are associated with a sort in S, too. Constants and variables are referred to as terms t.*

Example 5. We represent Ex. 1 as a RME knowledge base KB which consists of sorts $S = \{Person, Town, Status\}$, constants $U_{Person} = \{carl, stefan\}$ of sort $Person$, $U_{Town} = \{freiburg, yorkshire, austin\}$ of sort $Town$, $U_{Status} =$

$\{bad, average, good\}$ of sort *Status*, predicates $P = \{alarm(Person),\ burglary\ (Person),\ lives\_in(Person, Town),\ neighbourhood(Person, Status)\}$, and conditionals $R = \{c_1, \ldots, c_7\}$.

$$
\begin{aligned}
c_1 &= (alarm(\mathsf{X}) \mid burglary(\mathsf{X}))\ [0.9] \\
c_2 &= (alarm(\mathsf{X}) \mid lives\_in(\mathsf{X}, \mathsf{Y}), tornado(\mathsf{Y}))\ [0.9]\ \} \\
c_3 &= (burglary(\mathsf{X}) \mid neighborhood(\mathsf{X}, bad))\ [0.6] \\
c_4 &= (burglary(\mathsf{X}) \mid neighborhood(\mathsf{X}, average))\ [0.4] \\
c_5 &= (burglary(\mathsf{X}) \mid neighborhood(\mathsf{X}, good))\ [0.3] \\
c_6 &= (neighborhood(\mathsf{X}, \mathsf{Z}) \mid neighborhood(\mathsf{X}, \mathsf{Y}))\ [0.0]\ [\mathsf{Y} \neq \mathsf{Z}] \\
c_7 &= (lives\_in(\mathsf{X}, \mathsf{Z}) \mid lives\_in(\mathsf{X}, \mathsf{Y}))\ [0.0]\ [\mathsf{Y} \neq \mathsf{Z}]
\end{aligned}
$$

Notice, that conditionals c_6 and c_7 ensure mutual exclusion of the states for literals of *neighborhood* and *lives_in*.

Semantics are given to RME knowledge bases by grounding R with a *grounding operator* (GOP)[6] to a propositional probabilistic knowledge base and calculating the probability distribution with maximum entropy $P^{ME}_{\mathcal{G}_\chi(R)}$. A GOP (see Fig. 1) is a type of substitution pattern that facilitates the modeling of exceptional knowledge. For example, we could add the exceptional rule

$$c_8 = (alarm(\mathsf{X}) \mid lives\_in(\mathsf{X}, freiburg), tornado(freiburg))\ [0.1]$$

to KB in order to model that in Freiburg tornados are usually not strong enough to cause an alarm. Then the GOP $\mathcal{G}_{priority}$ would prefer all instances of c_8 above all instances of c_2, in which the constant *freiburg* occurs (for details see [11]).

After R is grounded, $\mathcal{G}_\chi(R)$ is treated as a propositional knowledge base and $P^{ME}_{\mathcal{G}_\chi(R)}$ can be calculated as in the propositional case [17]. A RME conditional $Q \in (\mathcal{L} \mid \mathcal{L})^{rel}$ is *fulfilled under the grounding* $\mathcal{G}_\chi(R)$ by the RME knowledge base R if the following hold:

$$R \models^{ME}_{\mathcal{G}_\chi} Q \iff P^{ME}_{\mathcal{G}_\chi(R)} \models \mathcal{G}_\chi(Q) \iff \forall q \in \mathcal{G}_\chi(Q) : P^{ME}_{\mathcal{G}_\chi(R)} \models q.$$

The RME inference process can be divided into three steps: 1.) ground the KB with a certain GOP $\mathcal{G}_\chi$, 2.) calculate the probability distribution $P^{ME}_{\mathcal{G}_\chi(R)}$ with maximum entropy for the grounded instance $\mathcal{G}_\chi(R)$, and 3.) calculate all probabilistic implications of $P^{ME}_{\mathcal{G}_\chi(R)}$.

A more elaborated overview on the framework of relational maximum entropy as introduced here is given in [11].

[6] In [11] the *naive-*, *cautious-*, *conservative-*, and *priority*-grounding strategies are presented and analyzed.

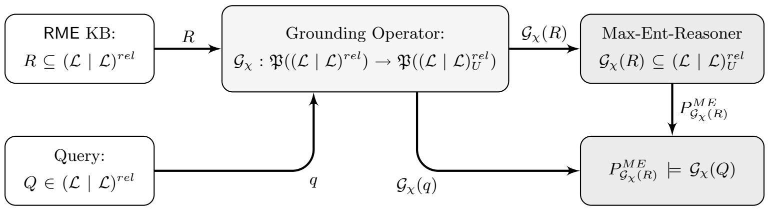

RME KB:
$R \subseteq (\mathcal{L} \mid \mathcal{L})^{rel}$
R
Grounding Operator:
$\mathcal{G}_\chi : \mathfrak{P}((\mathcal{L} \mid \mathcal{L})^{rel}) \to \mathfrak{P}((\mathcal{L} \mid \mathcal{L})^{rel}_U)$
$\mathcal{G}_\chi(R)$
Max-Ent-Reasoner
$\mathcal{G}_\chi(R) \subseteq (\mathcal{L} \mid \mathcal{L})^{rel}_U$
$P^{ME}_{\mathcal{G}_\chi(R)}$
Query:
$Q \in (\mathcal{L} \mid \mathcal{L})^{rel}$
q
$\mathcal{G}_\chi(q)$
$P^{ME}_{\mathcal{G}_\chi(R)} \models \mathcal{G}_\chi(Q)$

Fig. 1. RME semantics overview. A RME KB R is transformed into a propositional representation by the GOP $\mathcal{G}_\chi$. The result $\mathcal{G}_\chi(R)$ is used to calculate the Max-Ent-distribution $P^{ME}_{\mathcal{G}_\chi(R)}$ in order to answer the ground query $\mathcal{G}_\chi(q)$.

3 System Architecture

KReator[7] is an integrated development environment for representing, reasoning, and learning with relational probabilistic knowledge. Still being in development KReator aims to become a versatile toolbox for researchers and knowledge engineers in the field of statistical relational learning. KReator is written in Java and thus is designed using the object-oriented programming paradigm. It facilitates several architectural and design patterns such as model-view control, abstract factories, and command patterns. Central aspects of the design of KReator are *modularity*, *extensibility*, *usability*, *reproducibility*, and its intended application in scientific research.

Modularity and Extensibility KReator is modular and extensible with respect to several components. In the following we discuss just two important aspects. First, KReator separates between the internal logic and the user interface using an abstract command structure. Each top-level functionality of KReator is internally represented and encapsulated in an abstract KReatorCommand. Consequently, the user interface can be exchanged or modified in an easy and unproblematic way, because it is separated from the internal program structure by this KReatorCommand layer. As a matter of fact, the current version of KReator features both a graphical user interface and a command line interface (the KReator *console*) which processes commands in KReatorScript syntax (see Sec. 4.1). Second, KReator was designed to support many different approaches for relational knowledge representation, cf. Sec. 2. As a consequence, KReator features very abstract notions of concepts like knowledge bases, queries and data sets that can be implemented by a specific approach. At the moment, KReator supports knowledge representation using Bayesian logic programs (BLPs), Markov logic networks (MLNs), and the relational maximum entropy (RME) approach described in Sec. 2.3. Other formalisms will be integrated in the near future.

[7] The “KR” in KReator stands for “Knowledge Representation” and the name KReator indicates its intended usage as a development environment for knowledge engineers.

Usability and Reproducibility An important design aspect of KReator and especially of the graphical user interface is usability. While prototypical implementations of specific approaches to relational probabilistic knowledge representation (and approaches for any problem in general) are essential for validating results and evaluation, these software solutions are often very hard to use and differ significantly in their usage. Especially when one wants to compare different solutions these tools do not offer an easy access for new users. KReator features a common and simple interface to different approaches of relational probabilistic knowledge representation within a single application.

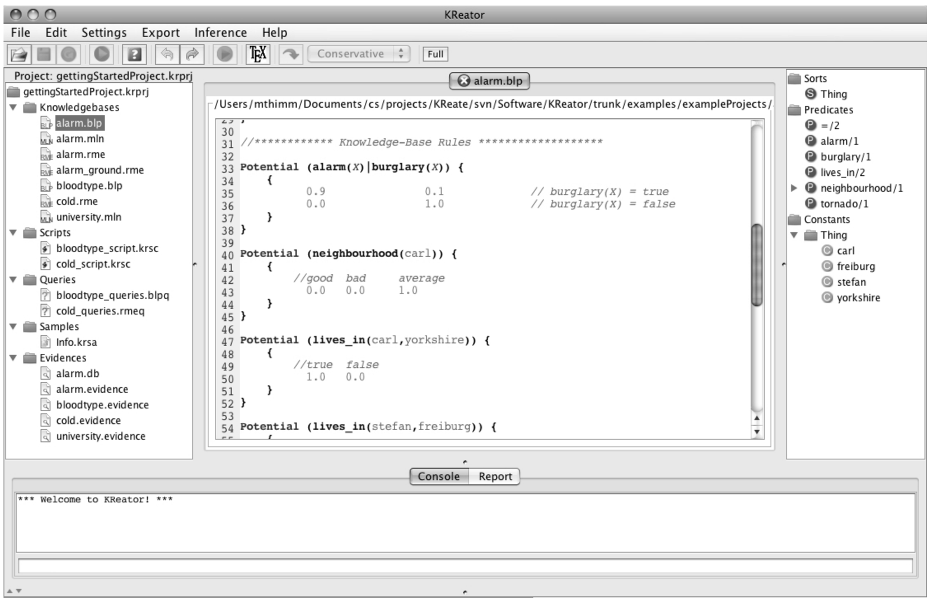
Application in Scientific Research Both usability and reproducibility are important aspects when designing a tool for conducting scientific research. Besides that, other important features are also provided within KReator. For example, KReator can export knowledge base files as formatted LaTeX output, making the seamless processing of example knowledge bases in scientific publications very convenient. KReator records every user operation (no matter whether it was caused by GUI interaction or by a console input) and its result in a *report*. Since all operations are reported in KReatorScript syntax, the report itself represents a valid script. Therefore the whole report or parts of it can be saved as a KReatorScript file which can be executed anytime to repeat the recorded operations.

Used frameworks KReator makes use of well-established software frameworks to process some of the supported knowledge representation formalisms. Performing inference on MLNs is handled entirely by the Alchemy software package [15], a console-based tool for processing Markov logic networks. Alchemy is open source software developed by the inventors of Markov logic networks and can freely be obtained on `http://alchemy.cs.washington.edu/`. To process ground RME knowledge bases, an appropriate reasoner for maximum entropy must be utilized. KReator does not directly interact with a certain reasoner. Instead, KReator uses a so-called ME-adapter to communicate with a (quite arbitrary) MaxEnt-reasoner. Currently, such an adapter is supplied for the SPIRIT reasoner [18]. SPIRIT is a tool for processing (propositional) conditional probabilistic knowledge bases using maximum entropy methods and can be obtained on `http://www.fernuni-hagen.de/BWLOR/spirit_int/`. An appropriate adapter for the MEcoRe reasoner [19] has also been developed.

4 Usage

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 and toolbars and four main panels: the project panel, the editor panel, the outline panel, and the console panel.

The project 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



```
//************ 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

can be opened independently in KReator, projects can help the knowledge engineer to organize his work. The project panel of KReator (seen in the upper left in Fig. 2) gives a complete overview on the project the user is currently working on.

The editor panel All files supported by KReator can be viewed and edited in the editor panel (seen in the upper middle in Fig. 2). Multiple files can be opened at the same time and the editor supports editing knowledge bases and the like with syntax-highlighting, syntax check, and other features normally known from development environments for programming languages.

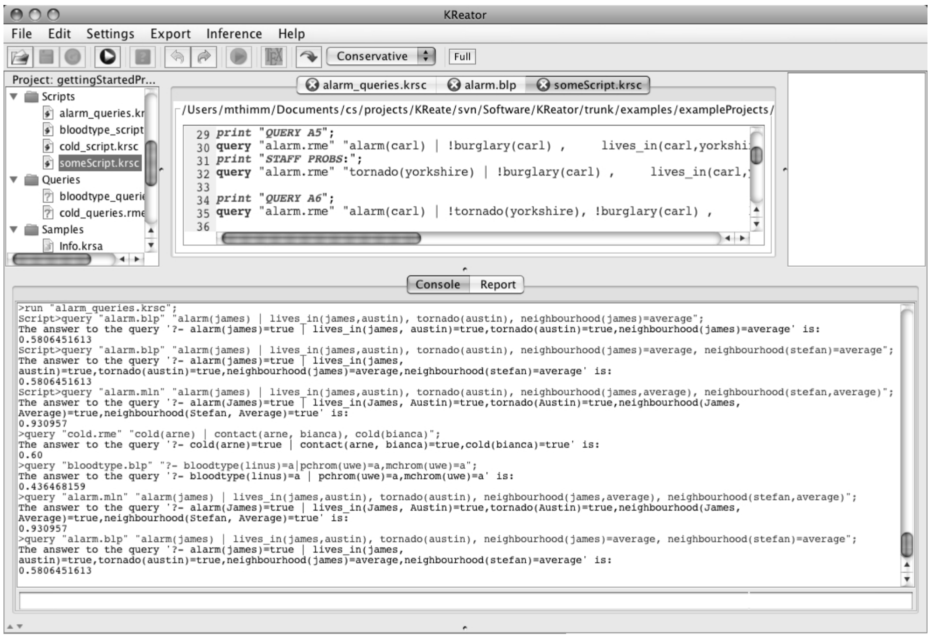
The outline panel The outline panel (seen in the upper right in Fig. 2) gives an overview on the currently viewed file in the editor panel. If the file is a knowledge base the outline shows information on the logical components of the knowledge base, such as used predicates (and, in case of BLPs, their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

The console panel The console panel (seen at the 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 nearly every 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, the scripting language also used for writing scripts (see below). The console panel also contains the report tab. 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.

4.1 The KReatorScript Language

The KReatorScript language incorporates all those commands which can be processed by the console and by script files as well (see Fig. 3 for some KReatorScript lines). Since there are commands available for all high-level KReator functionalities, every sequence of working steps can be expressed as an appropriate command sequence in a KReatorScript file. Thus, the (re-)utilization of scripts can clearly increase the efficiency and productivity when working with KReator. As mentioned above, the input console and report work hand in hand with KReator's scripting functionality, making the KReatorScript language a strong instrument in the whole working process.

Besides that, the utilization of scripts plays an important role in the active development of the KReator software, since it allows efficient and sustainable testing of high-level functionalities during the whole development process.



```
print "QUERY A5";
query "alarm.rme" "alarm(carl) | !burglary(carl) ,     lives_in(carl,yorkshi
print "STAFF PROBS:";
query "alarm.rme" "tornado(yorkshire) | !burglary(carl) ,     lives_in(carl,

print "QUERY A6";
query "alarm.rme" "alarm(carl) | !tornado(yorkshire), !burglary(carl) ,
```

```
>run "alarm_queries.krsc";
Script>query "alarm.blp" "alarm(james) | lives_in(james,austin), tornado(austin), neighbourhood(james)=average";
The answer to the query '?- alarm(james)=true | lives_in(james, austin)=true,tornado(austin)=true,neighbourhood(james)=average' is:
0.5806451613
Script>query "alarm.blp" "alarm(james) | lives_in(james,austin), tornado(austin), neighbourhood(james)=average, neighbourhood(stefan)=average";
The answer to the query '?- alarm(james)=true | lives_in(james,
austin)=true,tornado(austin)=true,neighbourhood(james)=average,neighbourhood(stefan)=average' is:
0.5806451613
Script>query "alarm.mln" "alarm(james) | lives_in(james,austin), tornado(austin), neighbourhood(james,average), neighbourhood(stefan,average)";
The answer to the query '?- alarm(James)=true | lives_in(James, Austin)=true,tornado(Austin)=true,neighbourhood(James,
Average)=true,neighbourhood(Stefan, Average)=true' is:
0.930957
>query "cold.rme" "cold(arne) | contact(arne, bianca), cold(bianca)";
The answer to the query '?- cold(arne)=true | contact(arne, bianca)=true,cold(bianca)=true' is:
0.60
>query "bloodtype.blp" "?- bloodtype(linus)=a|pchrom(uwe)=a,mchrom(uwe)=a";
The answer to the query '?- bloodtype(linus)=a | pchrom(uwe)=a,mchrom(uwe)=a' is:
0.436468159
>query "alarm.mln" "alarm(james) | lives_in(james,austin), tornado(austin), neighbourhood(james,average), neighbourhood(stefan,average)";
The answer to the query '?- alarm(James)=true | lives_in(James, Austin)=true,tornado(Austin)=true,neighbourhood(James,
Average)=true,neighbourhood(Stefan, Average)=true' is:
0.930957
>query "alarm.blp" "alarm(james) | lives_in(james,austin), tornado(austin), neighbourhood(james)=average, neighbourhood(stefan)=average";
The answer to the query '?- alarm(james)=true | lives_in(james,
austin)=true,tornado(austin)=true,neighbourhood(james)=average,neighbourhood(stefan)=average' is:
0.5806451613
```

Fig. 3. The console with some KReatorScript

4.2 Querying a Knowledge Base

One of the most important tasks when working with knowledge bases is to address queries to a knowledge base, i. e. to infer knowledge. For that reason, KReator provides several functionalities which simplify the dealing with queries and make it more efficient.

KReator permits the processing of queries expressed in a *unified query syntax*. This query syntax abstracts from the respective syntax which is necessary to address a "native" query to a BLP, MLN, or RME knowledge base (and which also depends on the respective inference engine). That way, a query in unified syntax can be passed to an appropriate BLP, MLN, and RME knowledge base as well. The idea behind this functionality is, that some knowledge (cf. Ex. 1) can be modeled in different knowledge representation approaches (cf. Ex. 2, Ex. 4, and Ex. 5) and the user is able to compare these approaches in a more direct way. Such a comparison can then be done by formulating appropriate queries in unified syntax, passing them to the different knowledge bases, and finally analyzing the different answers, i. e. the probabilities. 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. Figure 4 illustrates the processing of a query in unified syntax.

KReator offers the user an easy way to address a query to a knowledge base, simply by calling its query dialogue. In this dialogue (Fig. 5), the user can input the atoms to be queried and he can conveniently specify the evidence. Since evidence usually consists of several atoms and is often reused for different queries, the user has the option to specify a file which contains the evidence to be to considered. While processing a query, the output area of the dialogue informs about important steps of the inference process. The calculated answer is clearly displayed in the lower part of the dialogue.

The unified query syntax constitutes a compromise between the querying capabilities of the different knowledge representation formalisms. Therefore, some individual features of each formalism cannot be express in unified syntax. For this reason, KReator additionally offers a direct access to the querying capabilities of each inference engine. This is realized by corresponding console commands which take a query in "native" syntax as an argument. Besides the capability of passing individual (i. e. ad-hoc) queries to a knowledge base, KReator also supports so-called *query collections*. A query collections is a file which contains several queries (either in unified or native syntax). Such a query collection can be passed to a knowledge base, so that all included queries are processed one after another. That way, KReator supports a persistent handling and batch-style processing of queries.

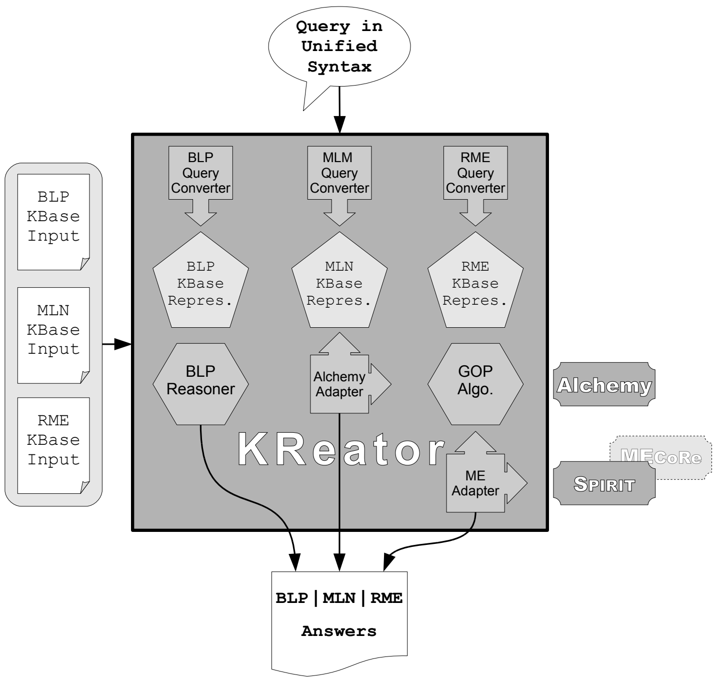
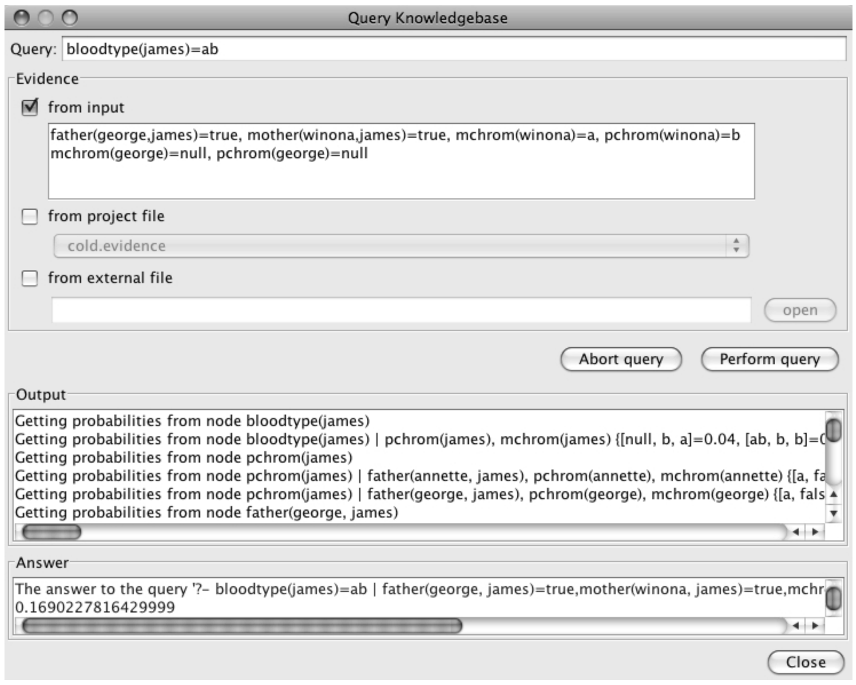


Fig. 4. Processing query in unified syntax

Example 6. Continuing our previous examples, now we consider the following evidence given: $lives\_in(james, yorkshire)$, $lives\_in(stefan, freiburg)$, $burglary(james)$, $tornado(freiburg)$, $neighborhood(james) = \mathsf{average}$, $neighborhood(stefan) = \mathsf{bad}$. The following table shows three queries and their respective probabilities inferred from each of the example knowledge bases. The nine separate calculations altogether took about three seconds. Each of the three knowledge bases represents Ex. 1 by a different knowledge representation approach. Nevertheless, the inferred probabilities are quite similar, except for the significant lower BLP probability of the query $alarm(stefan)$. This results from using noisy-or as combing rule in the BLP calculations and from the fact that the CPDs of the BLP knowledge base carry some information not incorporated in the MLN and BLP knowledge bases.

	BLP	MLN	RME
$alarm(james)$	0.900	0.896	0.918
$alarm(stefan)$	0.650	0.896	0.907
$burglary(stefan)$	0.300	0.254	0.354



Query: bloodtype(james)=ab

☑ from input

```
father(george,james)=true, mother(winona,james)=true, mchrom(winona)=a, pchrom(winona)=b
mchrom(george)=null, pchrom(george)=null
```

cold.evidence

Output

```
Getting probabilities from node bloodtype(james)
Getting probabilities from node bloodtype(james) | pchrom(james), mchrom(james) {[null, b, a]=0.04, [ab, b, b]=(
Getting probabilities from node pchrom(james)
Getting probabilities from node pchrom(james) | father(annette, james), pchrom(annette), mchrom(annette) {[a, fa
Getting probabilities from node pchrom(james) | father(george, james), pchrom(george), mchrom(george) {[a, fals
Getting probabilities from node father(george, james)
```

Answer

```
The answer to the query '?– bloodtype(james)=ab | father(george, james)=true,mother(winona, james)=true,mchr
0.1690227816429999
```

Fig. 5. Querying a knowledge base

5 Summary and Future Work

In this paper we presented KReator and illustrated its system architecture and usage. Although KReator is still in an early stage of development it already supports Bayesian logic programs, Markov logic networks, and relational maximum entropy. Thus KReator is a versatile toolbox for probabilistic relational reasoning and alleviates the researcher's and knowledge engineer's work with different approaches to statistical relational learning.

Since KReator is still in an early development stage, there are a lot of plans on future development. The integration of adequate learning algorithms will be one of our major tasks in the near future, as our main focus so far was the integration of reasoning components. We also plan to integrate an extended version of the CONDOR system [20] and other formalisms for relational probabilistic knowledge representation such as logical Bayesian networks [7] and probabilistic relational models [9], as well as to use KReator as a testbed to evaluate other approaches for relational probabilistic reasoning under maximum entropy.

The user interface undergoes a continuous development to further improve KReator's usability. KReator's reporting behavior will be made more controllable and the reusability of the report will be further enhanced, e. g. by a "group commands per knowledge base" option. The integration of external tools like Alchemy and SPIRIT will be made more seamlessly and the user will gain a better control over the exact behavior of these tools.

We plan to enhance KReator's unified query syntax to allow more complex queries. This requires more sophisticated conversion patterns to translate a unified query to the respective target syntax, e.g. to handle multi-state BLP predicates in an automated way. The enhancement of the query syntax will go along with the development of an even more challenging feature: We plan on introducing some kind of unified knowledge base (template) format. The final goal is to be able to formulate (at least) the central aspects of a knowledge base in a unified syntax and to have this knowledge base be converted to different target languages (at least semi-)automatically. Having this functionality available would dramatically improve the handling and comparison of different knowledge representation formalisms.

KReator is available under the GNU General Public License and can be obtained from `http://ls6-www.cs.uni-dortmund.de/kreator/`. The future development of KReator will also be strongly influenced be the feedback of early users. We will include such feedback in our development decisions and try to priorities such aspects which are most important to the users.

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

References

1. De Raedt, L., Kersting, K.: Probabilistic Inductive Logic Programming. In De Raedt, L., Kersting, K., Landwehr, N., Muggleton, S., Chen, J., eds.: Probabilistic Inductive Logic Programming. Springer (2008) 1–27
2. Cussens, J.: Logic-based Formalisms for Statistical Relational Learning. In Getoor, L., Taskar, B., eds.: An Introduction to Statistical Relational Learning. MIT Press (2007)
3. Kersting, K., De Raedt, L.: Bayesian Logic Programming: Theory and Tool. In Getoor, L., Taskar, B., eds.: An Introduction to Statistical Relational Learning. MIT Press (2007)
4. Domingos, P., Richardson, M.: Markov Logic: A Unifying Framework for Statistical Relational Learning. In: Proceedings of the ICML-2004 Workshop on Statistical Relational Learning and its Connections to Other Fields. (2004) 49–54
5. Pearl, J.: Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann (1998)
6. Wellman, M.P., Breese, J.S., Goldman, R.P.: From Knowledge Bases to Decision Models. The Knowledge Engineering Review **7**(1) (1992) 35–53
7. Fierens, D.: Learning Directed Probabilistic Logical Models from Relational Data. PhD thesis, Katholieke Universiteit Leuven (2008)
8. Jaeger, M.: Relational Bayesian Networks: A Survey. Electronic Transactions in Artificial Intelligence **6** (2002)
9. Getoor, L., Friedman, N., Koller, D., Tasker, B.: Learning Probabilistic Models of Relational Structure. In Brodley, C.E., Danyluk, A.P., eds.: Proc. of the 18th International Conf. on Machine Learning (ICML 2001), Morgan Kaufmann (2001)
10. Baral, C., Gelfond, M., Rushton, N.: Probabilistic Reasoning with Answer Sets. Theory and Practice of Logic Programming (2009)
11. Loh, S.: Relational Probabilistic Inference Based on Maximum Entropy. Master's thesis, Technische Universität Dortmund (to appear 2009)

12. Thimm, M.: Representing Statistical Information and Degrees of Belief in First-Order Probabilistic Conditional Logic (In preparation)
13. Ketkar, N.S., Holder, L.B., Cook, D.J.: Comparison of graph-based and logic-based multi-relational data mining. SIGKDD Explor. Newsl. **7**(2) (2005) 64–71
14. Muggleton, S., Chen, J.: A Behavioral Comparison of some Probabilistic Logic Models. In Raedt, L.D., Kersting, K., Landwehr, N., Muggleton, S., Chen, J., eds.: Probabilistic Inductive Logic Programming. Springer (2008) 305–324
15. Kok, S., Singla, P., Richardson, M., Domingos, P., Sumner, M., Poon, H., Lowd, D., Wang, J.: The Alchemy System for Statistical Relational AI: User Manual. Department of Computer Science and Engineering, University of Washington. (2008)
16. Lukasiewicz, T., Kern-Isberner, G.: Probabilistic Logic Programming under Maximum Entropy. In: Proceedings ECSQARU-99. (1999) 279–292
17. Rödder, W.: Conditional logic and the principle of entropy. Artificial Intelligence **117** (2000) 83–106
18. Rödder, W., Meyer, C.H.: Coherent Knowledge Processing at Maximum Entropy by SPIRIT. In: Proceedings UAI 1996. (1996) 470–476
19. Finthammer, M., Beierle, C., Berger, B., Kern-Isberner, G.: Probabilistic Reasoning at Optimum Entropy with the MEcore System. In: Proceedings of FLAIRS'09, AAAI Press (2009)
20. Fisseler, J., Kern-Isberner, G., Beierle, C., Koch, A., Moeller, C.: Algebraic Knowledge Discovery using Haskell. In: Practical Aspects of Declarative Languages, 9th International Symposium. Springer (2007)