Machine Learning manuscript No.
(will be inserted by the editor)

Learning Distributional Programs for Relational Autocompletion

Nitesh Kumar · Ondřej Kuželka · Luc De Raedt

Received: date / Accepted: date

Abstract Relational autocompletion is the problem of automatically filling out some missing fields in a relational database. We tackle this problem within the probabilistic logic programming framework of *Distributional Clauses* (DC), which supports both discrete and continuous probability distributions. Within this framework, we introduce *DreaML* – an approach to learn both the structure and the parameters of DC programs from databases that may contain missing information. To realize this, *DreaML* integrates statistical modeling, distributional clauses with rule learning. The distinguishing features of *DreaML* are that it 1) tackles relational autocompletion, 2) learns distributional clauses extended with statistical models, 3) deals with both discrete and continuous distributions, 4) can exploit background knowledge, and 5) uses an expectation-maximization based algorithm to cope with missing data. The empirical results show the promise of the approach, even when there is missing data.

Keywords Autocompletion · Relational Databases · Probabilistic Logic Programming · Program Synthesis

1 Introduction

Relational databases are widely used to store real-world data. They consist of multiple tables containing information about various types of entities as well as the associated foreign keys, which capture relationships among them. Such real-world databases are often noisy and may have missing values. The relational autocompletion problem is to fill out (some of) these fields automatically. This problem setting is simple, yet challenging and is viewed as an essential component of an automatic data scientist (De Raedt et al. 2018). We tackle this problem by learning a probabilistic logic program that defines the joint probability distribution over

Nitesh Kumar · Luc De Raedt
Department of Computer Science, KU Leuven, Belgium

Ondřej Kuželka
Department of Computer Science, Czech Technical University in Prague, Czechia

the database. This program can then be used to estimate the most likely values of the fields of interest.

Probabilistic logic programming (De Raedt and Kimmig 2015) (PLP) and statistical relational learning (SRL)(Richardson and Domingos 2006; Koller et al. 2007; Neville and Jensen 2007; Kimmig et al. 2012; Raedt et al. 2016) have introduced various formalisms that integrate relational logic with graphical models. While many PLP and SRL techniques exist, only a few of them are hybrid, i.e., can deal with both discrete and continuous variables. One of these hybrid formalisms are the *Distributional Clauses* (DC) introduced by Gutmann et al. (2011). Distributional clauses form a probabilistic logic programming language that extends the programming language Prolog with continuous as well as discrete probability distributions. It is this language that we adopt in this paper.

We first integrate statistical models in distributional clauses and use these to learn intricate patterns present in the data. This extended DC framework allows us to learn a DC program specifying a complex probability distribution over the entire database. Just like graphical models, this program can then be used for various types of inference. For instance, one can infer not only the output of statistical models based on their inputs but also the input when the output is observed.

In line with the approaches to inductive logic programming (Muggleton 1991; Blockeel and De Raedt 1998; De Raedt et al. 2015), we propose an approach, named *DreaML*[1] (*D*ist*r*ibutional Claus*e*s with St*a*tistical *M*odels *L*earner), that learns such a DC program from a database and background knowledge (if available). *DreaML* jointly learns the structure of distributional clauses, the parameters of its probability distributions and the parameters of the statistical models. The learned program can subsequently be used for autocompletion.

We study the problem also in the presence of missing data. The problem of learning the structure of hybrid relational models then becomes even more challenging and has, to the best our knowledge, never been attempted before. To tackle this problem, *DreaML* performs structure learning inside the stochastic EM procedure (Diebolt and Ip 1995).

Related Work There are several works in SRL for learning probabilistic models for relational databases, such as probabilistic relational models (PRMs, Friedman et al. 1999), relational Markov networks (RMNs, Taskar et al. 2002), and relational dependency networks (RDNs, Neville and Jensen 2007). PRMs extend Bayesian networks with concepts of objects, their properties, and relations between them. RDNs extend dependency networks, and RMNs extend Markov networks in the same relational setting. However, these models are generally restricted to discrete data. To address this shortcoming, several hybrid SRL formalisms were proposed such as continuous Bayesian logic programs (CBLPs, Kersting and De Raedt 2007), hybrid Markov logic networks (HMLNs, Wang and Domingos 2008), hybrid probabilistic logic programming (HProbLog, Gutmann et al. 2010), hybrid probabilistic relational models (HPRMs, Narman et al. 2010), and relational continuous models (RCMs, Choi et al. 2010). The work on hybrid SRL has mainly been focused on developing theory to represent continuous variables within the various SRL formalisms and on adapting inference procedures for hybrid domains. However,

[1] The code is publicly available: `https://github.com/niteshroyal/DreaML`, publication date: 15/09/19

little attention has been given to the design of algorithms for structure learning of hybrid SRL models. The closest to our work is the work on hybrid relational dependency networks (HRDNs, Ravkic et al. 2015), for which structure learning was also studied, but this approach assumes that the data is fully observed. There are few approaches for structure learning in the presence of missing data such as Kersting and Raiko (2005); Khot et al. (2012, 2015). However, these approaches are restricted to discrete data. Furthermore, existing hybrid models, except for HMLNs and HProbLog, are associated with local probability distributions such as conditional probability tables. As a result, it is difficult to read certain independencies such as context-specific independencies (CSIs, Boutilier et al. 1996). On the contrary, DC can represent CSIs leading to interpretable DC programs.

Learning meaningful and interpretable symbolic representations from data, in the form of rules, has been studied in many forms by the inductive logic programming(ILP) community (Quinlan 1990; Blockeel and De Raedt 1998; De Raedt and Dehaspe 1997). Standard ILP setting requires the input to be deterministic and usually the rules as well. Although some rule learners (Neville et al. 2003; Vens et al. 2006) output the confidence of their predictions, the rules learned for different targets have not been used jointly for probabilistic inference. To alleviate these limitations, De Raedt et al. (2015) proposed ProbFoil+ that can learn probabilistic rules from probabilistic data and background knowledge. In this approach, rules learned for different targets can jointly be used for inference. However, this approach does not deal with continuous random variables. A handful of approaches can learn rules with continuous probability distributions, and the learned rules can also be jointly used for inference. One such approach was proposed by Speichert and Belle (2018) that uses piecewise polynomials to learn intricate patterns from data. This approach differs from our approach as we use statistical models to learn these patterns. Moreover, this approach is restricted to deterministic input. Another approach for structure learning of dynamic distributional clauses, an extended DC framework that deals with time, has also been proposed by Nitti et al. (2016b). However, this approach cannot learn distributional clauses from background knowledge, which itself can be a set of distributional clauses. Furthermore, it learns the dynamic distributional clauses from fully observed data and does not deal with the missing data problem in relational databases. To the best of our knowledge, this paper makes the first attempt to learn interpretable and hybrid probabilistic logic programs from probabilistic data as well as background knowledge.

Contributions We summarise our contributions in this paper as follows:

– We integrate distributional clauses with statistical models in order and use the resulting framework to represent a hybrid probabilistic relational model.
– We introduce *DreaML*, an approach for relational autocompletion that learns distributional clauses with statistical models from relational databases and background knowledge.
– We extend *DreaML* to learn DC programs from relational databases with missing fields using the stochastic EM algorithm.
– We empirically evaluate *DreaML* on synthetic as well as real-world databases, which shows the promise of our approach.

client

cliId	*age*	*creditScore*
ann	33	–
bob	40	500
carl	–	450
john	55	700

hasAcc

cliId	*accId*
ann	a_11
bob	a_11
ann	a_20
john	a_10

hasLoan

accId	*loanId*
a_11	l_20
a_10	l_20
a_20	l_31
a_20	l_41

loan

loanId	*loanAmt*	*status*
l_20	20050	appr
l_21	–	pend
l_31	25000	decl
l_41	10000	–

account

accId	*savings*	*freq*
a_10	3050	high
a_11	2043	low
a_19	3010	high
a_20	–	?

Table 1 An example of a relational database consisting of entity tables (client, loan and account), and associative tables (hasLoan and hasAcc). Missing fields are denoted by "–" and the user's query is denoted by "?".

Organization The paper is organized as follows. We start by sketching the problem setting in Section 2. Section 3 reviews logic programming concepts and distributional clauses. In Section 4, we discuss the integration of distributional clauses with statistical models. In Section 5, we describe the specification of the DC program that we shall learn. Section 6 explains the learning algorithm, which is then evaluated in Section 7.

2 Problem Setting

Consider the example database shown in Table 1. It consists of *entity tables* and *associative tables*. Each entity table (e.g., client, loan, and account) contains information about instances of the same type. An associative table (e.g., hasAcc and hasLoan) encodes a relationship among entities. This toy example showcases two important properties of real-world applications, namely i) the attributes of entities may be numeric or categorical, and ii) there may be missing values in entity tables. These are denoted by "–".

The problem that we tackle in this paper is to autocomplete the user's queries denoted by "?" in the database. This problem will be solved by automatically learning a DC program from such a database. This program can then be used to fill out the queries with the most likely values.

3 Probabilistic Logic Programming

In this section, we first briefly review logic programming concepts and then introduce DC which extend logic programs with probability distributions.

3.1 Logic Programming

An *atom* $p(t_1, ..., t_n)$ consists of a predicate p/n of arity n and terms $t_1, ..., t_n$. A *term* is either a constant (written in lowercase), a variable (in uppercase), or a function symbol. For example, `hasLoan(a_1,L)`, `hasLoan(a_1,l_1)` and `hasLoan(a_1,func(L))` are atoms and `a_1`, `L`, `l_1` and `func(L)` are terms. A *literal* is an atom or its negation. Atoms which are negated are called *negative atoms* and atoms which are not negated are called *positive atoms.* A *clause* is a universally quantified disjunction of literals. A *definite clause* is a clause which contains exactly one positive atom and zero or more negative atoms. In logic programming, one usually writes definite clauses in the implication form $h \leftarrow b_1, ..., b_n$ (where we usually omit writing the universal quantifiers explicitly). Here, the atom h is called *head* of the clause; and the set of atoms $\{b_1, ..., b_n\}$ is called *body* of the clause. A clause with an empty body is called a *fact.* A *logic program* consists of a set of definite clauses.

Example 1 The clause $c \equiv$ `clientLoan(C,L)` $\leftarrow$ `hasAccount(C,A), hasLoan(A,L)` is a definite clause. Intuitively, it states that `L` is a loan of a client `C` if `C` has an account `A` and `A` is associated to the loan `L`.

A term, atom or clause, is called *ground* if it does not contain any variable. A *substitution* $\theta = \{V_1/t_1, ..., V_m/t_m\}$ assigns terms t_i to variables V_i. Applying θ to a term, atom or clause e yields the term, atom or clause $e\theta$, where all occurrences of V_i in e are replaced by the corresponding terms t_i. A substitution θ is called *grounding* for c if $c\theta$ is ground, i.e., contains no variables (when there is no risk of confusion we drop "for c").

Example 2 Applying the substitution $\theta = \{$`C/c_1`$\}$ to the clause c from Example 1 yields $c\theta$ which is `clientLoan(c_1,L)` $\leftarrow$ `hasAccount(c_1,A), hasLoan(A,L).`

A substitution θ *unifies* two atoms l_1 and l_2 if $l_1\theta = l_2\theta$. Such a substitution is called a unifier. Unification is not always possible. If there exists a unifier for two atoms l_1 and l_2, we call such atoms *unifiable* or we say that l_1 *and* l_2 *unify.*

Example 3 The substitution $\theta = \{$`C/c_1,M/L`$\}$ unifies the atoms `clientLoan(c_1,L)` and `clientLoan(C,M).`

The *Herbrand base* of a logic program P, denoted HB(P), is the set of all ground atoms which can be constructed using the predicates, function symbols and constants from the program P. A *Herbrand interpretation* is an assignment of truth-values to all atoms in the Herbrand base. A Herbrand interpretation I is a model of a clause $h \leftarrow \mathcal{Q}$, if and only if, for all grounding substitutions θ such that $\mathcal{Q}\theta \subseteq I$, it also holds that $h\theta \in I$.

The *least Herbrand model* of a logic program P, denoted LH(P), is the intersection of all Herbrand models of the logic program P, i.e. it consists of all ground atoms $f \in$ HB(P) that are logically entailed by the logic program P. The least Herbrand model of a program P can be generated by repeatedly applying the so-called T_{P} operator until fixpoint. Let I be the set of all ground facts in the program P. Starting from the set I of all ground facts contained in P, the T_{P} operator is defined as follows:

$$T_{\mathrm{P}}(I) = \{h\theta \mid h \leftarrow \mathcal{Q} \in \mathrm{P}, \mathcal{Q}\theta \subseteq I, h\theta \text{ is ground}\}, \tag{1}$$

That is, if the body of a rule is true in I for a substitution θ, the ground head $h\theta$ must be in $T_{\mathrm{P}}(I)$. It is possible to derive all possible true ground atoms using the

T_P operator recursively, until a fixpoint is reached ($T_P(I) = I$), i.e., until no more ground atoms can be added to I.

Given a logic program P, an *answer substitution* to a *query* of the form $? - q_1, \ldots, q_m$, where the q_i are literals, is a substitution θ such that all $q_i\theta$ are entailed by P, i.e., they belong to LH(P).

3.2 Distributional Clauses

DC is a natural extension of logic programs for representing probability distributions. A distributional clause is a rule of the form $h \sim \mathcal{D} \leftarrow b_1, ..., b_n$, where $\sim$ is a binary predicate used in infix notation. Note that the term $\mathcal{D}$ can be non-ground. For instance the next clause is a distributional clause.

```
creditScore(C) ∼ gaussian(755.5,0.1) ← clientLoan(C,L),status(L)≅appr.
```

A distributional clause without body is called a *probabilistic fact*. For instance:

```
age(c_2) ∼ gaussian(40,0.2).
```

The idea is that such ground atoms $h \sim \mathcal{D}$ define the random variable h as being distributed according to $\mathcal{D}$. To access the values of the random variables, we use the binary predicate $\cong$, which is used in infix notation for convenience. Here, $r \cong v$ is defined to be true if v is the value of the random variable r. It is also possible to define random variables that take only one value with probability 1, i.e., deterministic facts, like this:

```
age(c_1) ∼ val(55).
```

Example 4 Let us now apply the grounding substitution $\theta = \{\texttt{C/c\_1}, \texttt{L/l\_1}\}$ to the distributional clause mentioned above. This results in defining the random variable `creditScore(c_1)` as being drawn from the distribution $\mathcal{D}\theta =$ `gaussian(755.5, 0.1)` if `clientLoan(c_1,l_1)` is true and the outcome of the random variable `status(l_1)` takes the value `appr`, i.e., `status(l_1)` $\cong$ `appr`.

A distributional program $\mathbb{P}$ consists of the set of distributional and the set of definite clauses. A *possible world* for the program is generated using the $ST_{\mathbb{P}}$ operator, a stochastic version of the T_P operator. Gutmann et al. (2011) define the $ST_{\mathbb{P}}$ operator using the following generative process. The process starts with an initial world I containing all ground facts from the program. Then for each distributional clause $h \sim \mathcal{D} \leftarrow b_1, ..., b_n$ in the program, whenever the body $b_1\theta, ..., b_n\theta$ is true in the set I for the substitution θ, a value v for the random variable $h\theta$ is sampled from the distribution $\mathcal{D}\theta$ and $h\theta \cong v$ is added to the world I. This is also performed for deterministic clauses, adding ground atoms to I whenever the body is true. A function ReadTable($\cdot$) keeps track of the sampled values of random variables and ensures that for each random variable, only one value is sampled. This process is then recursively repeated until a fixpoint is reached ($ST_{\mathbb{P}}(I) = I$), i.e., until no more variables can be sampled and added to the world. The resulting world is called a *possible world*, while the intermediate worlds are called *partial possible worlds*.

Example 5 Suppose that we are given the following DC program $\mathbb{P}$:

```
hasAccount(c_1, a_1).
hasLoan(a_1, l_1).
age(c_1) ∼ val(55).
age(c_2) ∼ gaussian(40, 0.2).
status(l_1) ∼ discrete([0.7:appr, 0.3:decl]).
clientLoan(C,L) ← hasAccount(C,A), hasLoan(A,L).
creditScore(C) ∼ gaussian(755.5,0.1) ← clientLoan(C,L),
    status(L)≅appr.
creditScore(C) ∼ gaussian(350,0.1) ← clientLoan(C,L), status(L)≅decl.
```

Applying the $ST_{\mathbb{P}}$ operator, we can sample a possible world of the program $\mathbb{P}$ as follows:

```
[hasAccount(c_1,a_1), hasLoan(a_1,l_1), age(c_1)≅55] →
[hasAccount(c_1,a_1), hasLoan(a_1,l_1), age(c_1)≅55, age(c_2)≅40.2] →
[hasAccount(c_1,a_1), hasLoan(a_1,l_1), age(c_1)≅55, age(c_2)≅40.2,
    status(l_1)≅appr] →
[hasAccount(c_1,a_1), hasLoan(a_1,l_1), age(c_1)≅55, age(c_2)≅40.2,
    status(l_1)≅appr, clientLoan(c_1,l_1)] →
[hasAccount(c_1,a_1), hasLoan(a_1,l_1), age(c_1)≅55, age(c_2)≅40.2,
    status(l_1)≅appr, clientLoan(c_1,l_1), creditScore(c_1)≅755.0]
```

Distributional clauses can also have negated literals in their body. For instance:

```
creditScore(C) ∼ gaussian(755.5,0.1) ← clientLoan(C,L), \+status(L)≅_.
```

Here, if the status of the loan `L` is not defined, comparison involving the non-defined status will fail and its negation will succeed.

A distributional program $\mathbb{P}$ is said to be *valid* if it satisfies the following conditions. First, for each random variable $h\theta$, $h\theta \sim \mathcal{D}\theta$ has to be unique in the least fixpoint, i.e., there is one distribution defined for each random variable. Second, the program $\mathbb{P}$ needs to be stratified, i.e., there exists a rank assignment $\prec$ over predicates of the program such that for each distributional clause $h \sim \mathcal{D} \leftarrow b_1, ..., b_n$: $b_i \prec h$, and for each definite clause $h \leftarrow b_1, ..., b_n : b_i \preceq h$. Third, all ground probabilistic facts are Lebesgue-measurable. Fourth, each atom in the least fixpoint can be derived from a finite number of probabilistic facts.

Gutmann et al. (2011) show that:

Proposition 1 *Let $\mathbb{P}$ be a valid program. $\mathbb{P}$ defines a probability measure $P_{\mathbb{P}}$ over the set of fixpoints of the operator $ST_{\mathbb{P}}$. Hence, $\mathbb{P}$ also defines, for an arbitrary formula q over atoms in its Herbrand base, the probability that q is true.*

This proposition states that one obtains a proper probability measure when the distributional program satisfies the *validity conditions*.

By inference in DC we understand the process of answering probability of a query q given evidence e. Sampling full possible worlds for inference is generally inefficient or may not even terminate as possible worlds can be infinitely large. Therefore, DC uses an efficient sampling algorithm based on backward reasoning and likelihood weighting to generate only those facts that are relevant to answer the given query. To estimate the probability, samples of the partial possible world, i.e., the set of *relevant* facts, are generated. A partial possible world is generated after a successful completion of a proof of the evidence and the query using backward reasoning. The proof procedure is repeated N times to estimate the probability

$p(q \mid e)$ that is given by,

$$p(q \mid e) = \frac{\sum_{i=1}^{N} w_q^{(i)} w_e^{(i)}}{\sum_{i=1}^{N} w_e^{(i)}} \tag{2}$$

where $w_e^{(i)}$ is the likelihood weight of e and $w_q^{(i)}$ is the likelihood weight of q in i^{th} proof (we refer to Nitti et al. (2016a) for details).

4 Advanced Constructs in the DC Framework

In this section, we describe two advanced modeling constructs that can be represented in the DC framework. First, we allow aggregates in bodies of the distributional clauses and, second, we allow *statistical models* to be present in the bodies of the distributional clauses as well.

4.1 Aggregation

Aggregation functions are used to combine the properties of a set of instances of a specific type into a single property. Examples include the mode (most frequently occurring value); mean value (if values are numerical), maximum or minimum, cardinality, etc. They are implemented by second order *aggregation predicates* in the body of clauses. Aggregation predicates are analogous to the *findall* predicate in Prolog. They are of the form $aggr(T, Q, R)$, where $aggr$ is an aggregation function (e.g. sum), T is the target aggregation variable that occurs in the conjunctive goal query Q, and R is the result of the aggregation.

Example 6 Consider the following two clauses:

```
creditScore(C) ∼ gaussian(755.5,0.1) ← mod(T, (clientLoan(C,L),
    status(L)≅T), X), X≅appr.
creditScore(C) ∼ gaussian(500.5,0.1) ← mod(T, (clientLoan(C,L),
    status(L)≅T), X), \+X≅_.
```

The aggregation predicate `mod` in the body of this clause collects the status property of all loans that a client has into a list and unifies the constant `appr` ("approved loan") with the first most frequently occurring value in the list. Thus, the body of the first clause is true if and only if the most frequently occurring value in this list is `appr`. It may also happen that a client has no loan, or the client has loans but the statuses of these loans are not defined. In this case, this list will be empty, and the body of the second clause will be true.

4.2 Distributional Clauses with Statistical Models

Next we look at the way continuous random variables can be used in the body of a distributional clause for specifying the distributions in the head. One possibility as described in Gutmann et al. (2011) is to use standard comparison operators in the body of the distributional clauses, e.g., $\geq, \leq, >, <$, which can be used to compare values of random variables with constants or with values of other random variables.

Another possibility which we describe in this section, is to use a *statistical model*, that maps outcomes of the random variables in the body of a distributional clause to parameters of the distribution in the head. Formally, a distributional clause with a statistical model is a rule of the form $h \sim \mathcal{D}_\phi \leftarrow b_1, ..., b_n, \mathcal{H}_\psi$, where $\mathcal{H}_\psi$ is a function with the parameter ψ which relates continuous variables in $\{b_1, ..., b_n\}$ with the parameter ϕ in the distribution $\mathcal{D}_\phi$.

Example 7 Consider the following distributional clauses, which state that the credit score of a client depends on the age of the client. The loan status which can either be high or low depends on the amount of the loan. The loan amount is, in turn, distributed according to a Gaussian distribution.

```
creditScore(C) ∼ gaussian(M,0.1) ← age(C)≅Y, linear([Y],[10.1,200],M).
status(L) ∼ discrete(P1:low,P2:high) ← loan(L), loanAmt(L)≅Y,
    logistic([Y], [1.1,2.0],[P1,P2]).
loanAmt(L) ∼ gaussian(25472.3,10.2) ← loan(L).
```

Here, in the first clause, the linear model atom[2] with the parameter $\psi = [10.1, 200]$ relates the continuous variable Y and the mean M of the Gaussian distribution in the head. Likewise, in the second clause, the logistic model atom[3] with parameter $\psi = [1.1, 2.0]$ relates Y to the parameters $\phi = [P1, P2]$ of the discrete distribution in the head.

It is worth spending a moment studying the form of distributional clauses with statistical models as discussed above. Statistical models such as linear and logistic regression are fully integrated with the probabilistic logic framework in a way that exploits the full expressiveness of logic programming and the strengths of these models in learning intricate patterns. Moreover, we will see in Section 6 that these models can easily be trained along with learning the structure of the program. In this fully integrated framework, we not only infer in the forward direction, i.e., the output based on the input of these models but we can also infer in the backward direction, i.e., the input if we observe the output. For instance, in the above example, if we observe the status of the loan, then we can infer the loan amount, which is the input of the logistic model. Now, we can specify a complex probability distribution over continuous and/or discrete random variables using a distributional program having multiple clauses with statistical models.

5 Joint Model for a Relational Database

We will now use the DC formalism to define a probability distribution over the entire relational database. The next subsections describe: (i) how to map the relational database onto the set of distributional clauses, and (ii) the type of probabilistic relational model that we shall learn.

[2] `linear([Y],[10.1,200],M)` implements the linear function `M is 10.1 · Y + 200`. This signify that the credit score X is sampled from the probability distribution $\frac{1}{\sqrt{0.2\pi}} e^{-\frac{1}{0.2}(X-10.1\cdot Y-200)^2}$

[3] `logistic([Y],[1.1,2.0],[P1,P2])` implements the logistic function `P1 is` $\frac{1}{1+e^{-1.1\cdot Y-2.0}}$, `P2 is 1-P1`. This signify that the status X is sampled from the probability distribution $\left\{\frac{1}{1+e^{-1.1\cdot Y-2.0}}\right\}^{1[X=low]} \left\{1-\frac{1}{1+e^{-1.1\cdot Y-2.0}}\right\}^{1[X=high]}$, where $1[\cdot]$ is the "indicator function", so that 1 [a true statement] $= 1$, and 1 [a false statement] $= 0$

5.1 Modeling the Input Database (Sets $\mathcal{A}_{\mathcal{DB}}$ and $\mathcal{R}_{\mathcal{DB}}$)

In this paper, we assume relational databases consisting of multiple entity tables and multiple associative tables. The entity tables are assumed not to contain any foreign keys whereas the associative tables are assumed to contain only foreign keys which represent relations among entities. Although this is not a standard form, any database can be transformed into this canonical form, without loss of generality. For instance, the database in Table 1 is already in this form.

Next, we transform the given database $\mathcal{DB}$ to a set $\mathcal{A}_{\mathcal{DB}} \cup \mathcal{R}_{\mathcal{DB}}$ of probabilistic and deterministic facts. Here, $\mathcal{A}_{\mathcal{DB}}$ contains information about the values of attributes, represented using probabilistic facts, and $\mathcal{R}_{\mathcal{DB}}$ consists of information about the relational structure of the database (which entities exist and the relations among them), represented using deterministic facts.

In particular, given a database $\mathcal{DB}$, we transform it as follows:

– For every instance t in an entity table e, we add the deterministic fact $e(t)$ to $\mathcal{R}_{\mathcal{DB}}$. For example, from the client table, we add `client(ann)` for the instance `ann`.
– For each associative table r, we add deterministic facts $r(t_1, t_2)$ to $\mathcal{R}_{\mathcal{DB}}$ for all tuples (t_1, t_2) contained in the table r. For example, `hasAcc(ann,a_11)`.
– For each instance t with an attribute a of value v, we add a probabilistic fact $a(t) \sim val(v)$ to $\mathcal{A}_{\mathcal{DB}}$. For example, `age(ann) ~ val(33)`.

We call the predicate $e/1$ as entity predicate, $a/1$ as attribute predicate, and $r/2$ as relation predicate.

This representation of databases ensures that the *existence* of the individual entities is not a random variable. Likewise, the relations among entities are also not random variables. On the other hand, the values of attributes are random variables. This is exactly what we need for the relational autocompletion setting that we study in this paper in which we are only interested in predicting missing values of attributes but not in predicting missing relations or missing entities.

5.2 Modeling the Probability Distribution

Next, we describe the form of DC programs, called *joint model programs* (JMPs), that we will learn in this paper. JMPs are simply valid DC programs that satisfy certain additional restrictions. The reason why we restrict DC programs allowed as JMPs is that, owing to their expressive power, DC programs can easily represent very complex distributions in which inference may quickly become infeasible. The restrictions that JMPs must satisfy are as follows:

– Distributional clauses in JMPs cannot contain relational atoms in the heads; attribute atoms are the only atoms allowed in heads of the distributional clauses in JMPs. We will assume that the relational structure of the database is fixed and given by the set of deterministic facts $\mathcal{R}_{\mathcal{DB}}$.
– Distributional clauses in JMPs cannot contain comparison operators on outcomes of continuous random variables; continuous random variables in JMPs are only allowed to affect other (continuous or discrete) random variables via distributional clauses with the statistical model.

Apart from restricting their form, we also require JMPs to specify a probability distribution over all attributes of every instance in the database, given the relational structure of the database. So for every attribute that appears in the database, JMPs must contain at least one distributional clause with the predicate corresponding to this attribute in the head. The relational structure is fixed by adding the set of deterministic facts $\mathcal{R}_{\mathcal{DB}}$ in JMPs.

Example 8 A JMP[4] that specifies a probability distribution over the entire database in Table 1 is shown below:

```
client(ann). client(john). ...
hasAcc(ann,a_11). hasAcc(ann,a_20). ...
freq(A) ∼ discrete([0.2:low,0.8:high]) ← account(A).
savings(A) ∼ gaussian(2002,10.2) ← account(A), freq(A)≅low.
savings(A) ∼ gaussian(3030,11.3) ← account(A), freq(A)≅high.
age(C) ∼ gaussian(Mean,3) ← client(C), avg(X,(hasAcc(C,A),
     savings(A)≅X), Y), creditScore(C)≅Z,
     linear([Y,Z],[30,0.2,-0.4],Mean).
loanAmt(L) ∼ gaussian(Mean,10) ← loan(L), avg(X,(hasLoan(A,L),
     savings(A)≅X),Y), linear([Y],[100.1, 10],Mean).
loanAmt(L) ∼ gaussian(25472.3,10.2) ← loan(L),
     avg(X,(hasLoan(A,L),savings(A)≅X),Y), \+Y≅_.
status(L) ∼ discrete([P1:appr, P2:pend, P3:decl]) ← loan(L), avg(X,
     (hasLoan(A,L),hasAcc(C,A),creditScore(C)≅X),Y), loanAmt(L)≅Z,
     softmax([Y,Z],
     [[0.1,-0.3,-2.4],[0.3,0.4,0.2],[0.8,1.9,-2.9]],[P1,P2,P3]).
creditScore(C) ∼ gaussian(Mean,10.1) ← client(C), max(X,(hasAcc(C,A),
     savings(A)≅X), Y), mod(X,(hasAcc(C,A),freq(A)≅X),Z),Z≅low,
     linear([Y],[300,0.2],Mean).
creditScore(C) ∼ gaussian(Mean,15.3) ← client(C), max(X,(hasAcc(C,A),
     savings(A)≅X), Y), mod(X,(hasAcc(C,A),freq(A)≅X),Z),Z≅high,
     linear([Y],[600,0.2],Mean).
creditScore(C) ∼ gaussian(Mean,12.3) ← client(C), max(X,(hasAcc(C,A),
     savings(A)≅X), Y), mod(X, (hasAcc(C,A),freq(A)≅X),Z),\+Z≅_,
     linear([Y],[500,0.8],Mean).
```

At this point, it is worth taking time to study the above program in detail as several aspects of the probability distribution specified by the program can be directly read from it. First of all, the program specifies a probability distribution over 24 random variables (fields) of the database, where 8 of them belong to `client` table, 8 to `loan` table, and 8 to `account` table. The set of clauses with the same head, when grounded, explicates random variables that directly influence the random variable defined in the head. For instance, the program explicates that random variables `freq(a_11)`, `freq(a_20)`, `savings(a_11)` and `savings(a_20)` directly influence the random variable `creditScore(ann)`, since the client `ann` has two accounts, namely `a_11` and `a_20`. The distributions in the head and the statistical models in the body of these grounded clauses quantify this direct causal

[4] The `softmax` predicate implements the softmax function `Pj is` $\frac{e^{w_{j_1} \cdot V_1 + \cdots + w_{j_i} \cdot V_i + w_{j_0}}}{\sum_{k=1}^{d} e^{w_{k_1} \cdot V_1 + \cdots + w_{k_i} \cdot V_i + w_{k_0}}}$. This signify that the status X is sampled from the probability distribution $\left\{\frac{e^{0.1 \cdot Y - 0.3 \cdot Z - 2.4}}{N}\right\}^{1[X=appr]} \left\{\frac{e^{0.3 \cdot Y + 0.4 \cdot Z + 0.2}}{N}\right\}^{1[X=pend]} \left\{\frac{e^{0.8 \cdot Y + 1.9 \cdot Z - 2.9}}{N}\right\}^{1[X=decl]}$, where $N = e^{0.1 \cdot Y - 0.3 \cdot Z - 2.4} + e^{0.3 \cdot Y + 0.4 \cdot Z + 0.2} + e^{0.8 \cdot Y + 1.9 \cdot Z - 2.9}$.

influence. The program represents this knowledge about all random variables in a concise way.

6 Learning the Joint Model Program

In this section, we describe our approach *DreaML*[5] that learns the joint model program. This section is divided into two parts. In the first part, we present an algorithm that learns JMPs from a relational database and background knowledge. This algorithm uses negated literals in the body of learned clauses to handle missing fields in the database. In the second part, we present an iterative algorithm that learns JMP by explicitly modeling the missing values, and that starts with the JMP learned in the first part.

6.1 JMP Learner

Let us first look at the input of the learning algorithm.

6.1.1 Input

DreaML requires a DC program $\mathbb{P}_{\mathcal{IN}}$ as input, which consists of three components: a relational database $\mathcal{DB}$ transformed into the set $\mathcal{A}_{\mathcal{DB}} \cup \mathcal{R}_{\mathcal{DB}}$ as introduced in Section 5.1, a declarative bias and a background knowledge $\mathcal{BK}$ (if available). We discuss the bias and $\mathcal{BK}$ in turn.

The *declarative bias* consists of four types of declarations, i.e., type, mode, rand and rank declarations, which specifies the space of possible clauses $\mathcal{L}_{\mathcal{DB}}$ that is explored by our learning algorithm. *DreaML* requires that all predicates are accompanied by *type declarations* of the form $type(pred(t_1, \cdots, t_n))$, where t_i denotes the type of the i-th argument, i.e., the domain of the variable. We also employ modes for each attribute predicate, which specify the form of literal b_i in the body of the clause $h \sim \mathcal{D}_\phi \leftarrow b_1, \ldots, b_n, \mathcal{H}_\psi$. A *mode declaration* is an expression of the form $mode(a_1, aggr, (r(m_1, \ldots, m_j), a_2(m_k)))$, where m_i are different modes associated with variables of predicates, $aggr$ is the name of aggregation function, r is the relational predicate, and a_i are attribute predicates. If the relational predicates are absent, then the aggregation function is not needed, so the mode declaration reduces to the form $mode(a_1, none, a_2(m_k)))$. The modes m_i can be either *input* (denoted by "+"), *output* (denoted by "−") or *ground* (denoted by "*c*"). Furthermore, the type of random variable (i.e., discrete or continuous) is defined by what we call *rand declarations*. As we have already seen in Section 3.2, the second validity condition of the DC program states that there exists a rank assignment $\prec$ over predicates of the program. Hence, we introduce an additional declaration, which we call *rank declaration*, to specify the rank assignment over attribute predicates.

The third component of the input is a *background knowledge*, that is, additional information about entities in the database and relations among the entities that the learning algorithm should take into consideration. Inductive logic programming (Muggleton and De Raedt 1994) specifically targets learning from structured

[5] Apart from JMPs, *DreaML* can also learn other DC programs based on the specification provided by the user in the input, following the same principle.

data and background knowledge, since first-order logic is expressive and declarative. The idea of learning from data as well as background knowledge is also adopted by probabilistic inductive logic programming literature, such as ProbFOIL+ (De Raedt et al. 2015) for the Problog programs (De Raedt et al. 2007). Motivated by the literature, we also allow background knowledge, expressed as the set of definite and/or distributional clauses $\mathcal{BK}$, as the input of the learning algorithm.

Example 9 An input DC program $\mathbb{P}_{\mathcal{IN}}$ for the database in Table 1 is shown in Figure 1. The first clause in background knowledge shown in the bottom-right of the figure states that the age of `carl` follows a Gaussian distribution, and the second clause states that if a client has an account in the bank and the account is linked to a loan account, then the client also has a loan.

```
% Type declaration
type(client(c)).
type(loan(l)).
type(account(a)).
type(hasAcc(c,a)).
type(hasLoan(c,l)).
type(age(c)).
type(creditScore(c)).
type(loanAmt(l)).
type(status(l)).
type(savings(a)).
type(freq(a)).

% Mode declaration
mode(age,none,creditScore(+)).
mode(age,sum,(hasAcc(+,-),savings(+))).
mode(age,avg,(hasAcc(+,-),savings(+))).
mode(age,mod,(hasAcc(+,-),freq(+))).
mode(age,max,(cliLoan(+,-),loanAmt(+))).
mode(age,mod,(cliLoan(+,-),status(+))).
mode(status,none,loanAmt(+)).
mode(status,mod,(hasLoan(-,+),freq(+))).
⋮

% Rank declaration
rank([age,creditScore,loanAmt,
      status,savings,freq]).

% Random variable declaration
rand(age,continuous,[]).
rand(creditScore,continuous,[]).
rand(loanAmt,continuous,[]).
rand(status,discrete,[appr,pend,decl]).
rand(savings,continuous,[]).
rand(freq,discrete,[low,high]).

% Transformed database
client(ann).
loan(l_20).
account(a_10).
age(ann) ~ val(33).
creditScore(john) ~ val(700).
savings(a_10) ~ val(3050).
freq(a_10) ~ val(high).
loanAmt(l_20) ~ val(20050).
hasAcc(ann,a_11).
hasLoan(a_11,l_20).
⋮

% Background knowledge
age(carl) ~ gaussian(40,5.1).
cliLoan(C,L)←hasAcc(C,A),hasLoan(A,L).
```

Fig. 1 An example of $\mathbb{P}_{\mathcal{IN}}$ consisting of a transformed relational database in Table 1, along with a background knowledge and a declarative bias.

6.1.2 Learning

Our goal is to learn a joint model program $\mathbb{P}_{\mathcal{DB}} \in \mathcal{L}_{\mathcal{DB}}$ that best explains the database and the background knowledge present in the input DC program $\mathbb{P}_{\mathcal{IN}}$, where $\mathcal{L}_{\mathcal{DB}}$ is the search space specified by the declarative bias in $\mathbb{P}_{\mathcal{IN}}$.

Distributional clauses in a DC program are *mutually exclusive*, i.e., there is only one distribution defined for each random variable in a possible world (recall

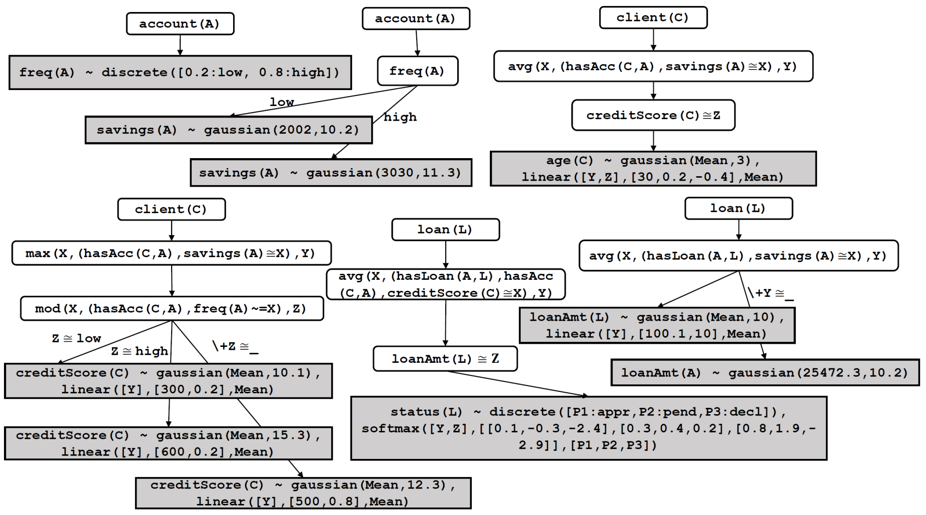
account(A)
freq(A) ~ discrete([0.2:low, 0.8:high])
account(A)
freq(A)
low
high
savings(A) ~ gaussian(2002,10.2)
savings(A) ~ gaussian(3030,11.3)
client(C)
avg(X,(hasAcc(C,A),savings(A)≅X),Y)
creditScore(C)≅Z
age(C) ~ gaussian(Mean,3), linear([Y,Z],[30,0.2,-0.4],Mean)
client(C)
max(X,(hasAcc(C,A),savings(A)≅X),Y)
mod(X,(hasAcc(C,A),freq(A)~=X),Z)
Z ≅ low
Z ≅ high
+Z ≅_
creditScore(C) ~ gaussian(Mean,10.1), linear([Y],[300,0.2],Mean)
creditScore(C) ~ gaussian(Mean,15.3), linear([Y],[600,0.2],Mean)
creditScore(C) ~ gaussian(Mean,12.3), linear([Y],[500,0.8],Mean)
loan(L)
avg(X,(hasLoan(A,L),hasAcc(C,A),creditScore(C)≅X),Y)
loanAmt(L) ≅ Z
status(L) ~ discrete([P1:appr,P2:pend,P3:decl]), softmax([Y,Z],[[0.1,-0.3,-2.4],[0.3,0.4,0.2],[0.8,1.9,-2.9]],[P1,P2,P3])
loan(L)
avg(X,(hasLoan(A,L),savings(A)≅X),Y)
+Y ≅_
loanAmt(L) ~ gaussian(Mean,10), linear([Y],[100.1,10],Mean)
loanAmt(A) ~ gaussian(25472.3,10.2)

Fig. 2 A collection of DLTs corresponding to the JMP in Example 8.

the first validity condition of the DC program). In other words, in every possible world, each random variable $h\theta$ is sampled from a unique head $h\theta \sim \mathcal{D}\theta$ of a clause of the program. This property of the DC program will allow us to learn the joint model program $\mathbb{P}_{\mathcal{DB}}$ by inducing a tree, which is also mutually exclusive, for each attribute predicate in the database. The program in Example 8 represented as a collection of trees is shown in Figure 2.

We call the tree induced by our algorithm, a *distributional logic tree* (DLT). The learning task reduces to induction of the DLT for each attribute predicate from the input program.

Definition 1 (*Learning distributional logic trees*) Given: An attribute predicate h and an input DC program $\mathbb{P}_{\mathcal{IN}}$ consisting of,

– a set of facts $\mathcal{A}_{\mathcal{DB}} \cup \mathcal{R}_{\mathcal{DB}}$ representing a database $\mathcal{DB}$;
– a set of facts or/and clauses $\mathcal{BK}$ representing the background knowledge (possibly empty);
– a declarative bias;
– a scoring function;

Find: A distributional logic tree for h, which conforms to the declarative bias and best explains instances of the attribute h and background knowledge, with respect to the scoring function.

Next, we specify the DLT learned by *DreaML*.

Distributional logic tree The DLT for the attribute predicate h is a rooted tree in which each leaf is labeled by a probability distribution $\mathcal{D}_\phi$ and a statistical model $\mathcal{H}_\psi$, and each internal node is labeled with an atom b. A path from the root to a leaf node then corresponds to a distributional clause of the form $h \sim \mathcal{D}_\phi \leftarrow b_1, ..., b_n, \mathcal{H}_\psi$. The set of all such paths in the DLT corresponds to the set of clauses for h.

Example 10 Consider the bottom-left DLT in the collection of DLTs shown in Figure 2. The leftmost path from the root proceeding to the leaf node in the DLT corresponds to the following distributional clause:

```
creditScore(C) ∼ gaussian(Mean,10.1) ← client(C), max(X,(hasAcc(C,A),
    savings(A) ≅X), Y), mod(X,(hasAcc(C,A),freq(A) ≅
    X),Z),Z≅low,linear([Y],[300,0.2],Mean).
```

Internal nodes b_i can be binary predicates $\cong$ or aggregation predicates. Based on the type of the random variable defined by the nodes, the internal nodes can be of two types:

– *Discrete nodes* that test a discrete attribute of an instance. These nodes have $n + 1$ branches where n is the number of possible values that the attribute can take. The right-most branch is reserved for "undefined", i.e., when the attribute of an instance is missing.
– *Continuous nodes* that specify a continuous attribute of an entity used to estimate the parameter of the distribution $\mathcal{D}_\phi$ and/or the statistical model $\mathcal{H}_\psi$. These nodes have two branches. Here also, the right-most branch is reserved for undefined.

The rightmost branch in all types of internal nodes corresponds to the *negated literal* in the distributional clause. Hence, instances for which the attribute is missing choose the rightmost branch of internal nodes. This allows all instances belonging to h, to be covered by at least one clause induced by the algorithm.

The leaf node of the tree contains the head of the distributional clause, which is of the form $h \sim \mathcal{D}_\phi$. The leaf node also includes the statistical model $\mathcal{H}_\psi$ present in the body of the distributional clause. Depending on the type of the random variable defined by h, the distribution $\mathcal{D}_\phi$ and the model $\mathcal{H}_\psi$ can be one of the three types in our current implementation of *DreaML*:

– If h defines a continuous variable with continuous variables $V_1, \cdots, V_i$, appearing in the branch, then $\mathcal{H}_\psi$ implements a linear function $\mu = w_1.V_1 + \cdots + w_i.V_i + w_0$ and $\mathcal{D}_\phi$ is a Gaussian distribution $gaussian(\mu, \sigma^2)$.
– If h defines a binary variable with continuous variables $V_1, \cdots, V_i$, appearing in the branch, then $\mathcal{D}_\phi$ is a discrete distribution $discrete([q_1{:}true, (1 - q_1){:}false])$ and $\mathcal{H}_\psi$ implements a logistic function $q_1 = \frac{1}{1+exp(w_1.V_1 - \cdots - w_i.V_i - w_0)}$.
– If h defines a d-valued discrete variable with continuous variables $V_1, \cdots, V_i$, appearing in the branch, then $\mathcal{D}_\phi$ is a discrete distribution $discrete([q_1{:}l_1, \cdots, q_d{:}l_d])$, where $l_i \in dom(h)$ and $\mathcal{H}_\psi$ implements a softmax function

$$q_j = \frac{\exp(w_{j_1}.V_1 + \cdots + w_{j_i}.V_i + w_{j_0})}{\sum_{k=1}^{d} \exp(w_{k_1}.V_1 + \cdots + w_{k_i}.V_i + w_{k_0})}$$

It should be clear that if no continuous variable appears in the branch then $\mathcal{H}_\psi$ is absent and $\mathcal{D}_\phi$ is a Gaussian distribution or discrete distribution depending on the type of random variable defined by h.

Induction of the distributional logic tree Induction of the tree for a target attribute predicate starts with a body $\mathcal{Q}$ containing only one atom, i.e., an entity predicate of the same type as the attribute predicate. The algorithm recursively adds nodes in the tree. To add a node, it first tests whether the score of the clause corresponding to the root proceeding to the node increases through splitting the node, by at least a threshold ϵ. If it does not, then the node is turned into a leaf node; otherwise, all possible refinements of the node are generated and scored using a scoring function in Equation 5, which will be explained later in this section. The best refinement is selected and incorporated into the internal node. This internal node is splitted by adding new nodes to each of its branch. This procedure is called recursively for the new nodes. For generating refinements of the node, the algorithm employs a refinement operator (De Raedt 2008) that specializes the body $\mathcal{Q}$ (a set of atoms in the path from the root to the node) by adding a literal l to the body yielding $(\mathcal{Q}, l)$. The operator ensures that only the refinements that are declarative bias conform are generated.

Addition of the leaf node requires estimating parameters of the statistical model $\mathcal{H}_\psi$ and/or parameters of the distribution $\mathcal{D}_\phi$. Let us look at the next example to understand the estimation of the parameters.

Example 11 Suppose that the input program $\mathbb{P}_{\mathcal{IN}}$ contains the following distributional clauses and facts:

```
account(a_1). account(a_2).
freq(a_1) ~ discrete([0.2:low,0.8:high]).
freq(a_2) ~ val(low).
savings(a_1) ~ val(3000).
savings(a_2) ~ val(4000).
deposit(A) ~ gaussian(30000, 100.1) ← account(A), freq(A) ≅low.
deposit(A) ~ gaussian(40000, 200.2) ← account(A), freq(A) ≅high.
```

Further, suppose that a path from the root to leaf node while inducing DLT for savings corresponds to the following clause,

```
savings(A) ~ gaussian(μ,σ) ← account(A),freq(A) ≅low,deposit(A) ≅X,
    linear([X],[w1,w0],μ).
```

where $\{w_0, w_1, \mu, \sigma\}$ are the parameters that we want to estimate. There are two substitutions of the variable A, i.e., $\theta_1 = \{\texttt{A/a\_1}\}$ and $\theta_2 = \{\texttt{A/a\_2}\}$, that are possible for the clause. The parameters of the clause can be approximately estimated from samples of the partial possible world obtained by proving the query ?- $h\theta_1, \mathcal{Q}\theta_1$ and the samples obtained by proving the query ?- $h\theta_2, \mathcal{Q}\theta_2$. Following Equation 2, the weight $w^{(i)}$ of i^{th} sample is given by,

$$w^{(i)} = \frac{w_q^{(i)} w_e^{(i)}}{\sum_{i=1}^{N} w_e^{(i)}} \tag{3}$$

Suppose, we obtained the following partial possible worlds, where each world is weighted by the weight obtained using Equation 3.

```
[savings(a_1)≅3000,account(a_1),freq(a_1)≅low,deposit(a_1)≅30010.1],w_θ1^(1) = 0.1.
[savings(a_1)≅3000,account(a_1),freq(a_1)≅low,deposit(a_1)≅40410.3],w_θ1^(2) = 0.1.
[savings(a_2)≅4000,account(a_2),freq(a_2)≅low,deposit(a_2)≅30211.3],w_θ2^(1) = 0.5.
[savings(a_2)≅4000,account(a_2),freq(a_2)≅low,deposit(a_2)≅30410.5],w_θ2^(2) = 0.5.
```

The parameters can now easily be estimated by maximizing the expectation of log-likelihood (Conniffe 1987) of savings, that is given by the expression,

$$\ln(\mathcal{N}(3000 \mid 30010.1w_1 + w_0, \sigma)) \times 0.1 + \ln(\mathcal{N}(3000 \mid 40410.3w_1 + w_0, \sigma)) \times 0.1 + \\ \ln(\mathcal{N}(4000 \mid 30211.3w_1 + w_0, \sigma)) \times 0.5 + \ln(\mathcal{N}(4000 \mid 30410.5w_1 + w_0, \sigma)) \times 0.5$$

It should be clear that the same approach can be used to estimate the parameters from any distributional clauses and/or facts present in $\mathbb{P}_{\mathcal{IN}}$.

Notice from the above example that substitutions of the clause are required to estimate parameters of the clause. We now formally define such substitutions.

Definition 2 (*Substitutions at the leaf node*) Given the input program $\mathbb{P}_{\mathcal{IN}}$ and a path from the root to a leaf node L corresponding to a clause $h \sim \mathcal{D}_\phi \leftarrow \mathcal{Q}, \mathcal{H}_\psi$, we define the substitutions Θ at the leaf node L to be the set of substitutions of the clause that ground all entity, relation and attribute predicates in the clause.

In general, parameters of any distribution and/or of any statistical model at any leaf node can be estimated by maximizing the expectation of log-likelihood $E(\boldsymbol{\varphi})$, which is given by the following expression,

$$E(\boldsymbol{\varphi}) = \sum_{\theta_i \in \Theta} \sum_{j=1}^{N} \ln\Big(p(h\theta_i \mid \boldsymbol{\varphi}, V\theta_i^{(j)})\Big) w_{\theta_i}^{(j)} \tag{4}$$

where $\boldsymbol{\varphi}$ is the set of parameters, Θ is the set of substitutions at the leaf node, V is the set of continuous variables in $\mathcal{Q}$, N is the number of times the query ?- $h\theta_i, \mathcal{Q}\theta_i$ is proved, $w_{\theta_i}^{(j)}$ is the weight of the j^{th} sample, $V\theta_i^{(j)}$ is j^{th} sample of continuous random variables and $p(h\theta_i \mid \boldsymbol{\varphi}, V\theta_i^{(j)})$ is the probability distribution of the random variable $h\theta_i$ given $\boldsymbol{\varphi}$ and $V\theta_i^{(j)}$. For the three simpler statistical models that we considered, the expectation of log-likelihood is a convex function. *DreaML* uses scikit-learn (Pedregosa et al. 2011) to obtain the maximum likelihood estimate $\widehat{\boldsymbol{\varphi}}$ of the parameters.

The Scoring Function Clauses are scored by the Bayesian Information Criterion (BIC)(Schwarz et al. 1978) for selecting among the set of candidate clauses while inducing DLTs. The score of a clause $\mathcal{C}$ is given by,

$$s(\mathcal{C} \mid \mathbb{P}_{\mathcal{IN}}) = 2E(\widehat{\boldsymbol{\varphi}}) - k \ln |\Theta| \tag{5}$$

where $|\Theta|$ is the number of substitutions Θ at the leaf node corresponding to the clause, k is the number of parameters. The score avoids over-fitting and naturally takes care of the different number of substitutions for different clauses. While deciding whether to split a node or not, the score of the clause formed at the node is compared with the sum of scores of clauses formed at different split nodes.

Learning a collection of DLTs A collection of DLTs is obtained by separately inducing a DLT for each attribute predicate in the database. Each path from the root node to a leaf node in each DLT corresponds to a clause in the learned JMP $\mathbb{P}_{\mathcal{DB}}$. This program can now be used to infer the user's queries for the autocompletion task.

In the experiment, we demonstrate that such programs achieves a state-of-the-art (Ravkic et al. 2015) performance during prediction.

6.2 Learning JMPs using the stochastic EM

In the previous section, we used negated literals in the body of learned clauses to deal with missing fields in the database. In this section, we discuss a more principled approach for learning programs in the presence of missing fields. This approach uses the stochastic EM (Diebolt and Ip 1995) to deal with the missing fields. In this setting, we assume that background knowledge is not present.

Consider an input program $\mathbb{P}_{\mathcal{IN}}$ consisting of a relational database $\mathcal{DB}$ with missing fields $\mathbf{Z} = \{Z_1, \ldots, Z_m\}$ and observed fields $\{X_1 \cong x_1, \ldots, X_n \cong x_n\}$ (abbreviated as $\mathbf{X} \cong \mathbf{x}$), where x_i is the value of the observed field X_i. Given the current learned program $\mathbb{P}^i_{\mathcal{DB}}$ specifying a probability distribution $p(\mathbf{X}, \mathbf{Z})$, the $(i+1)$-th EM step is conducted in two steps:

E-step A sample $\{Z_1 \cong z_1, \ldots, Z_m \cong z_m\}$ (abbreviated as $\mathbf{Z} \cong \mathbf{z}$) of the missing fields $\mathbf{Z}$ is taken from the conditional probability distribution $p(\mathbf{Z} \mid \mathbf{X} \cong \mathbf{x})$. The missing fields $\mathbf{Z}$ are filled in the database by asserting the facts $\{Z_1 \sim val(z_1), \ldots, Z_m \sim val(z_m)\}$ (abbreviated as $\mathbf{Z} \sim val(\mathbf{z})$) in $\mathbb{P}_{\mathcal{IN}}$.

M-step A new program $\mathbb{P}^{i+1}_{\mathcal{DB}}$ is learned from the input program $\mathbb{P}_{\mathcal{IN}}$ as discussed in section 6.1 and subsequently facts $\mathbf{Z} \sim val(\mathbf{z})$ are retracted from $\mathbb{P}_{\mathcal{IN}}$. However, in this case, parameters of distribution and/or statistical models at the leaf node are estimated by maximizing the log-likelihood rather than maximizing the expectation of the log-likelihood. This is because, in this case, the input program $\mathbb{P}_{\mathcal{IN}}$ does not consist of probabilistic facts or distributional clauses. Following equation 4, the log-likelihood function $L(\boldsymbol{\varphi})$ is given by the following expression,

$$L(\boldsymbol{\varphi}) = \sum_{\theta_i \in \Theta} \ln\Big(p(h\theta_i \mid \boldsymbol{\varphi}, V\theta_i^{(j)})\Big) \tag{6}$$

The iterative procedure initiates by first learning a program $\mathbb{P}^0_{\mathcal{DB}}$ with negated literals to deal with missing fields. Subsequent programs learned after the first iteration do not contain negated literals since missing fields are filled in with their samples. It is worth noting that we learn the structure as well as parameters of the program $\mathbb{P}_{\mathcal{DB}}$, which is more challenging compared to learning only parameters of the model as in the case of standard stochastic EM. In the experiment, we demonstrate that the program learned using stochastic EM performs better compared to the learned program with negated literals.

The learning algorithm presented in this section is similar to the standard structural EM algorithm for learning Bayesian networks (Friedman et al. 1997). The main difference, apart from having different target representations (DC vs. Bayesian networks), is that structural EM uses the standard EM (Dempster et al. 1977) for structure learning. Our approach uses the stochastic EM for structure learning due to the reason of tractability (hybrid probabilistic inference in large relational databases is computationally very challenging).

7 Experiments

This section empirically evaluates JMPs learned by *DreaML*. Specifically, we want to answer the following questions:

7.1 **How does the performance of the JMP learned by *DreaML* compare with the state-of-the-art hybrid relational model when trained on a fully observed database?**

We compared the JMP learned by *DreaML* with the state-of-the-art algorithm *Learner of Local Models - Hybrid (LLM-H)* introduced by Ravkic et al. (2015). The LLM-H algorithm learns a probabilistic relational model in the form of a hybrid relational dependency network. The algorithm requires the training database to be fully observed. In addition to LLM-H, we also compared the performance of JMP with individual DLTs learned for each attribute from the training database separately. We used the same data set (one synthetic and one real-world) as used in Ravkic et al. (2015).

Synthetic University Data Set The data set contains information of 800 *students*, 125 *courses* and 125 *professors* with three attributes in the data set being continuous while the rest three attributes being discrete. For example, the attribute *intelligence/1* represents the intelligence level of students in the range $[50.0, 180.0]$ and the attribute *difficulty/1* represents the difficulty level of courses that takes three discrete values $\{easy, med, hard\}$. The data set also contains three relations: *takes/2*, denoting which course is taken by a student; *friend/2*, denoting whether two students are friends and *teaches/2*, denoting which course is taught by a professor. More details about the data set can be found in Ravkic et al. (2015).

Real-world PKDD'99 Financial Data Set The data set is generated by processing the financial data set from the PKDD'99 Discovery Challenge. The data set is about services that a bank offers to its clients, such as loans, accounts, and credit cards. It contains information of four types of entities: $5,358$ *clients*, $4,490$ *accounts*, 680 *loans* and 77 *districts*. Ten attributes are of the continuous type, and three are of the discrete type. The data set contains four relations: *hasAccount/2* that links clients to accounts; *hasLoan/2* that links accounts to loans; *clientDistrict/2* that links clients to districts; and finally *clientLoan/2* that links clients to loans. The original data set is split into ten folds considering *account* to be the central entity. All information about clients, loans, and districts related to one account appear in the same fold. More information about the data set can be found in Ravkic et al. (2015).

We used the same evaluation metrics as used in Ravkic et al. (2015) to evaluate the quality of predictions of JMP.

Evaluation metric To measure the predictive performance for discrete attributes, multi-class area under ROC curve ($\text{AUC}_{\text{total}}$) (Provost and Domingos 2000) was used, whereas normalized root-mean-square error (NRMSE) was used for continuous attributes. The NRMSE of an attribute ranges from zero to one and is calculated by dividing the RMSE by the range of the attribute. To measure the quality of the probability estimates, weighted pseudo-log-likelihood (WPLL) (Kok and Domingos 2005) was used, which corresponds to calculating pseudo-log-likelihood of instances of an attribute in the test data set and dividing it by the number of instances in the test data set.

In our experiment, we used the aggregation function *avg* for continuous attributes, and *mode* and *cnt* (cardinality) for discrete attributes. An ordering chosen randomly among attributes was provided in the declarative bias. While training individual DLTs, ordering among attributes was not considered since those DLTs were not joint models but individual models for each attribute. We used the same database with the same settings as in Ravkic et al. (2015) to compare the performance of our algorithm. Table 2 shows the comparison on PKDD data set divided into ten folds. Nine folds were used for training and the remaining for testing. During testing, prediction of a test field was the mode of the probability distribution of the field obtained by conditioning over the rest of the test data. A Bayes-ball algorithm (Shachter 2013) that performs lazy grounding of the learned program was used to find evidence that was relevant to the test field. Table 3 shows the comparison on University data set divided into train and test set. Numbers for LLM-H are taken directly from Ravkic et al. (2015).

Evaluation	Predicate	LLM-H	DLT	JMP
$\text{AUC}_{\text{total}}$	gender/1	0.50 ± 0.01	0.50 ± 0.03	**0.52 ± 0.03**
	freq/1	0.82 ± 0.01	**0.83 ± 0.04**	0.77 ± 0.07
	loanStatus/1	0.66 ± 0.04	0.79 ± 0.04	**0.82 ± 0.05**
NRMSE	clientAge/2	0.28 ± 0.02	**0.24 ± 0.01**	0.24 ± 0.02
	avgSalary/1	**0.13 ± 0.02**	0.24 ± 0.00	0.18 ± 0.01
	ratUrbInhab/1	0.20 ± 0.00	**0.18 ± 0.00**	0.25 ± 0.01
	avgSumOfW/1	**0.02 ± 0.00**	0.03 ± 0.01	**0.02 ± 0.00**
	avgSumOfCred/1	**0.02 ± 0.00**	0.03 ± 0.01	0.02 ± 0.01
	stdOfW/1	0.05 ± 0.01	**0.05 ± 0.00**	0.05 ± 0.01
	stdOfCred/1	0.05 ± 0.01	**0.04 ± 0.00**	**0.04 ± 0.00**
	avgNrWith/1	0.15 ± 0.01	0.11 ± 0.01	**0.11 ± 0.00**
	loanAmount/1	0.16 ± 0.02	**0.11 ± 0.01**	0.12 ± 0.01
	monthlyPayments/1	0.18 ± 0.02	0.15 ± 0.02	**0.14 ± 0.01**

Table 2 The performance of JMP compared to LLM-H and single trees for each attribute (DLT) on PKDD'99 financial data set. The best results (mean and standard deviation) are in bold.

Predicate	LLM-H	DLT	JMP
nrhours/1	-4.48	**-3.20**	-3.39
difficulty/1	-0.02	-0.03	**-0.00**
ability/1	-5.34	**-3.77**	-3.83
intelligence/1	-4.66	**-3.37**	-4.08
grade/2	-1.45	**-1.00**	**-1.00**
satisfaction/2	-1.54	**-1.05**	**-1.05**
Total WPLL	-17.49	**-12.42**	-13.35

Table 3 WPLL for each attribute on the university data set consisting of 800 students, 125 courses, and 125 professors. The best results are in bold.

Unsurprisingly, we observe that the performance of JMP is similar to individual DLTs since the training data were fully observed in this first experiment, consequently, single models for attributes, i.e., DLTs learned the same patterns in the data as JMP. Also, the test data were fully observed, so JMP had no advantage over individual DLTs. The performance of JMP is similar to LLM-H since LLM-H also uses the same features as *DreaML* to learn classification or regression model for attributes; consequently, learning similar regularities from the data. However, on several occasions, JMP outperforms LLM-H. The experiment suggests that JMP learned by *DreaML* achieves comparable results as the state-of-the-art algorithm for fully observed data.

7.2 Can *DreaML* learn the DLT for a single target attribute in the presence of background knowledge?

Learning the DLT for a single target attribute from training data in the presence of background knowledge ($\mathcal{BK}$) is a more complex task compared to learning the DLT from only training data. $\mathcal{BK}$ provides additional information about attributes, so the learning task that involves probabilistic inference becomes complex. We performed this experiment to examine whether *DreaML* can also learn DLTs for a single attribute from the training database as well as the $\mathcal{BK}$ expressed as the set of distributional clauses.

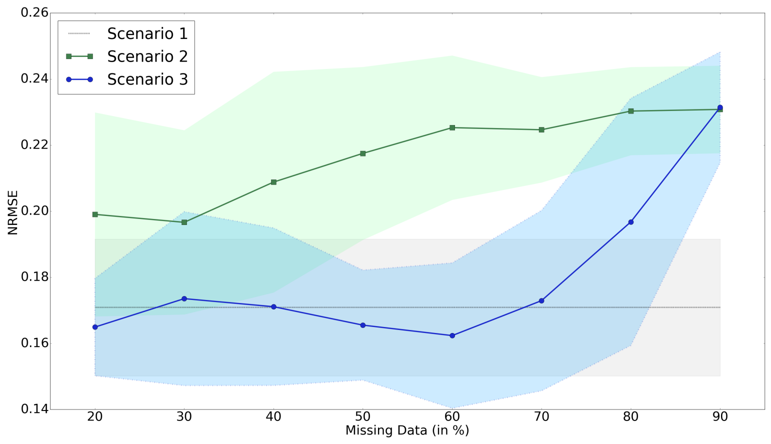

Fig. 3 NRMSE of predictions for *monthlyPayment*/1 in the test set versus the percentage of removed fields for the three scenarios.

We used the PKDD data set divided into ten folds. Seven folds were used for training the DLT for an attribute; one fold was used for testing that DLT; and two folds were used for generating $\mathcal{BK}$, which was a set of distributional clauses for all attributes, i.e., a JMP. We considered three scenarios: 1) A DLT for an attribute was induced from the full training set; subsequently, the DLT was used to predict the attribute in the test fold. 2) A partial data set was generated by removing $x\%$ of fields at random from the training set. Subsequently, a DLT for the same attribute was induced from the partial set. In this case, some clauses in the DLT had negated literals in the body. 3) A DLT for the same attribute was induced from the partial set as well as $\mathcal{BK}$.

The predictive performance for *monthlyPayment*/1 in the test set in all three scenarios, varying the percentage of removed fields is shown in Figure 3. Much lower NRMSE is observed in the third scenario. We can conclude that *DreaML* can learn DLT from the training database using additional probabilistic information from $\mathcal{BK}$.

7.3 Can *DreaML* learn the JMP from the relational database when a large portion of the database is missing?

The benefit of learning a joint probabilistic model over learning individual models for each attribute is that the joint model can be used to autocomplete the user's query in the database with missing fields. The most likely value for the query can be estimated by probabilistic inference in the joint model. Even more challenging task that requires numerous such inferences is learning the joint model from a database with a lot of missing fields. We evaluated the performance of JMP learned by *DreaML* from one such database. To the best of our knowledge, no system in the literature can learn the joint model from the partially observed relational database with continuous as well as discrete attributes. We used the PKDD financial database and performed the following experiment to answer the question.

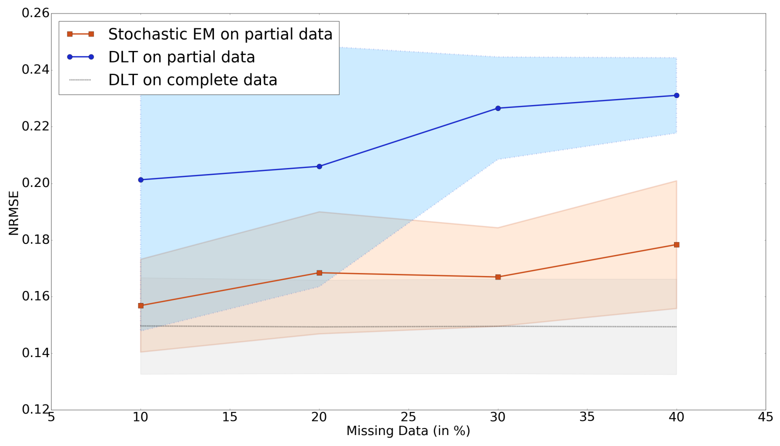

Fig. 4 NRMSE of predictions for $monthlyPayment/1$ in the test database for the three scenarios versus the percentage of removed fields from the client, loan, account, and district table.

We randomly removed some percentage of fields from client, loan, account, and district table of the training database to obtain a partial database. Then we trained three probabilistic models to predict attributes in the test database. The first model was just an individual model, i.e., DLT for each attribute trained on the partial database. It is worth reiterating that DLT can be learned even when some fields are missing since we allow negated literals in the body of distributional clauses. The second model was a JMP obtained by performing stochastic EM on the partial database. The last model was also individual DLT for each attribute but trained on the complete training database. The predictive performance of all

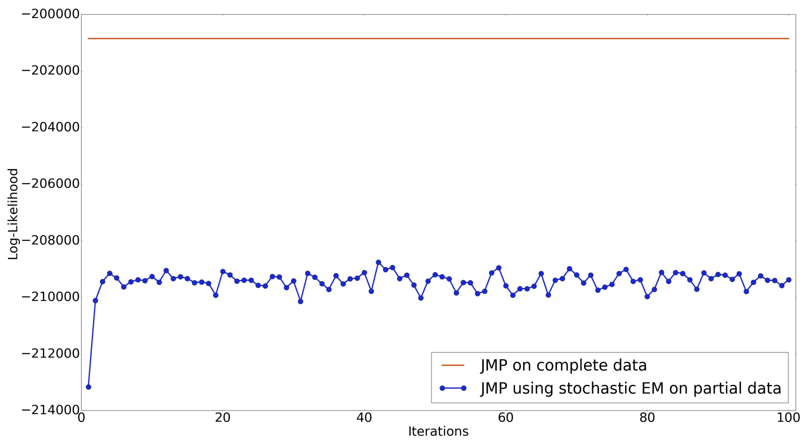

Fig. 5 The convergence of the stochastic EM on PKDD financial database

models for the *monthlyPayment/1* attribute is shown in Figure 4. Nine folds of the database were used for training and the rest for testing. The variance of NRMSE is shown by shaded region when the experiment was repeated ten times on this database. We observe that the JMP obtained using EM performs far better than the individual DLT trained on the partial database. As expected, the DLT trained on the complete database achieves the lowest NRMSE. The convergence of the stochastic EM after few iterations is shown in Figure 5. To obtain the figure, JMP was obtained from the PKDD financial database with 10% of fields removed using EM. This figure shows the database log-likelihood after each iteration of EM compared with the database log-likelihood when JMP was obtained from the complete database.

These results demonstrate that *DreaML* can learn the JMP even when a large portion of fields in the training database is missing.

8 Conclusions

We presented *DreaML*, a probabilistic logic programming based approach for tackling the problem of autocompletion in relational databases. Our approach integrates DC formalism with statistical models to represent an interpretable probabilistic relational model in the form of a DC program called joint model program. *DreaML* learns the program automatically from databases and makes use of additional background knowledge, if available. The program learned from a fully observed database achieves state-of-the-art performance. *DreaML* uses the stochastic EM to learn the program from the database with missing fields. We demonstrated that the program learned by *DreaML* performs well, even when a large portion of the database is missing.

Acknowledgements This work has received funding from the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme (grant agreement No [694980] SYNTH: Synthesising Inductive Data Models). OK's work has been supported by the OP VVV project *CZ.02.1.01/0.0/0.0/16_019/0000765* "Research Center for Informatics" and a donation from X-Order Lab. Part of this work was done while OK was with KU Leuven and was supported by Research Foundation - Flanders (project G.0428.15).

References

Blockeel H, De Raedt L (1998) Top-down induction of first-order logical decision trees. Artificial intelligence 101(1-2):285–297

Boutilier C, Friedman N, Goldszmidt M, Koller D (1996) Context-specific independence in bayesian networks. In: Proceedings of the Twelfth international conference on Uncertainty in artificial intelligence, Morgan Kaufmann Publishers Inc., pp 115–123

Choi J, Hill DJ, Amir E (2010) Lifted inference for relational continuous models. In: Workshops at the Twenty-Fourth AAAI Conference on Artificial Intelligence

Conniffe D (1987) Expected maximum log likelihood estimation. Journal of the Royal Statistical Society: Series D (The Statistician) 36(4):317–329

De Raedt L (2008) Logical and relational learning. Springer Science & Business Media

De Raedt L, Dehaspe L (1997) Clausal discovery. Machine Learning 26(2-3):99–146

De Raedt L, Kimmig A (2015) Probabilistic (logic) programming concepts. Machine Learning 100(1):5–47

De Raedt L, Kimmig A, Toivonen H (2007) Problog: A probabilistic prolog and its application in link discovery

De Raedt L, Dries A, Thon I, Van den Broeck G, Verbeke M (2015) Inducing probabilistic relational rules from probabilistic examples. In: Twenty-Fourth International Joint Conference on Artificial Intelligence

De Raedt L, Blockeel H, Kolb S, Teso S, Verbruggen G (2018) Elements of an automatic data scientist. In: International Symposium on Intelligent Data Analysis, Springer, pp 3–14

Dempster AP, Laird NM, Rubin DB (1977) Maximum likelihood from incomplete data via the em algorithm. Journal of the Royal Statistical Society: Series B (Methodological) 39(1):1–22

Diebolt J, Ip EH (1995) A stochastic em algorithm for approximating the maximum likelihood estimate. Tech. rep., Sandia National Labs., Livermore, CA (United States)

Friedman N, Getoor L, Koller D, Pfeffer A (1999) Learning probabilistic relational models. In: IJCAI, vol 99, pp 1300–1309

Friedman N, et al. (1997) Learning belief networks in the presence of missing values and hidden variables. In: ICML, vol 97, pp 125–133

Gutmann B, Jaeger M, De Raedt L (2010) Extending problog with continuous distributions. In: International Conference on Inductive Logic Programming, Springer, pp 76–91

Gutmann B, Thon I, Kimmig A, Bruynooghe M, De Raedt L (2011) The magic of logical inference in probabilistic programming. Theory and Practice of Logic Programming 11(4-5):663–680

Kersting K, De Raedt L (2007) 1 bayesian logic programming: Theory and tool. Statistical Relational Learning p 291

Kersting K, Raiko T (2005) 'say em'for selecting probabilistic models for logical sequences. In: Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence, AUAI Press, pp 300–307

Khot T, Natarajan S, Kersting K, Shavlik J (2012) Structure learning with hidden data in relational domains

Khot T, Natarajan S, Kersting K, Shavlik J (2015) Gradient-based boosting for statistical relational learning: the markov logic network and missing data cases. Machine Learning 100(1):75–100

Kimmig A, Bach SH, Broecheler M, Huang B, Getoor L (2012) A short introduction to probabilistic soft logic. In: NIPS Workshop on probabilistic programming: Foundations and applications, vol 1, p 3

Kok S, Domingos P (2005) Learning the structure of markov logic networks. In: Proceedings of the 22nd international conference on Machine learning, ACM, pp 441–448

Koller D, Friedman N, Džeroski S, Sutton C, McCallum A, Pfeffer A, Abbeel P, Wong MF, Heckerman D, Meek C, et al. (2007) Introduction to statistical relational learning. MIT press

Muggleton S (1991) Inductive logic programming. New generation computing 8(4):295–318

Muggleton S, De Raedt L (1994) Inductive logic programming: Theory and methods. The Journal of Logic Programming 19:629–679

Narman P, Buschle M, Konig J, Johnson P (2010) Hybrid probabilistic relational models for system quality analysis. In: 2010 14th IEEE International Enterprise Distributed Object Computing Conference, IEEE, pp 57–66

Neville J, Jensen D (2007) Relational dependency networks. Journal of Machine Learning Research 8(Mar):653–692

Neville J, Jensen D, Friedland L, Hay M (2003) Learning relational probability trees. In: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, pp 625–630

Nitti D, De Laet T, De Raedt L (2016a) Probabilistic logic programming for hybrid relational domains. Machine Learning 103(3):407–449

Nitti D, Ravkic I, Davis J, De Raedt L (2016b) Learning the structure of dynamic hybrid relational models. In: 22nd European Conference on Artificial Intelligence (ECAI) 2016, vol 285, pp 1283–1290

Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E (2011) Scikit-learn: Machine Learning in Python . Journal of Machine Learning Research 12:2825–2830

Provost F, Domingos P (2000) Well-trained pets: Improving probability estimation trees

Quinlan JR (1990) Learning logical definitions from relations. Machine learning 5(3):239–266

Raedt LD, Kersting K, Natarajan S, Poole D (2016) Statistical relational artificial intelligence: Logic, probability, and computation. Synthesis Lectures on Artificial Intelligence and Machine Learning 10(2):1–189

Ravkic I, Ramon J, Davis J (2015) Learning relational dependency networks in hybrid domains. Machine Learning 100(2-3):217–254

Richardson M, Domingos P (2006) Markov logic networks. Machine learning 62(1-2):107–136

Schwarz G, et al. (1978) Estimating the dimension of a model. The annals of statistics 6(2):461–464

Shachter RD (2013) Bayes-ball: The rational pastime (for determining irrelevance and requisite information in belief networks and influence diagrams). arXiv preprint arXiv:13017412

Speichert S, Belle V (2018) Learning probabilistic logic programs in continuous domains. arXiv preprint arXiv:180705527

Taskar B, Abbeel P, Koller D (2002) Discriminative probabilistic models for relational data. In: Proceedings of the Eighteenth conference on Uncertainty in artificial intelligence, Morgan Kaufmann Publishers Inc., pp 485–492

Vens C, Ramon J, Blockeel H (2006) Remauve: A relational model tree learner. In: International Conference on Inductive Logic Programming, Springer, pp 424–438

Wang J, Domingos PM (2008) Hybrid markov logic networks. In: AAAI, vol 8, pp 1106–1111