Learning Distributional Programs for Relational Autocompletion[☆]

Nitesh Kumar[a], Ondřej Kuželka[b], Luc De Raedt[a]

[a]*Department of Computer Science, KU Leuven, Belgium*
[b]*Department of Computer Science, Czech Technical University in Prague, Czech Republic*

Abstract

Relational autocompletion is the problem of automatically predicting missing values in a relational database. To address this problem, we introduce *LyRic*, a novel approach based on probabilistic programming, and in particular on the distributional clauses framework, which extends logic programming with discrete and continuous probability distributions. *LyRic* learns both the structure and parameters of a distributional program that specifies a probability distribution over the entire database, in the presence of missing values. In contrast with standard approaches to missing value imputation, *LyRic* is designed to deal with multiple related tables. In addition, it natively handles discrete and continuous values and can make use of additional background knowledge, if available. Computing the best autocompletion, in the relational setting, can often become infeasible in practice. To address this issue, we also propose an approximate autocompletion approach. Our empirical results show that our approach achieves good performance, even when a large portion of the database is missing.

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. Real-world databases are often noisy and may have missing values. The relational autocompletion problem is to fill out these missing values automatically, and that is viewed as a central problem in automated data science[1]. The problem can be solved by automatically learning a probabilistic relational model specifying a joint probability distribution over the entire database. The probabilistic inference in the learned model can then be used to estimate the most likely missing values.

Statistical relational learning (SRL)[2, 3, 4, 5] aims at developing formalisms and learning probabilistic relational models. While many SRL techniques exist, only a few of them are hybrid, i.e., can deal with both discrete and continuous variables. The structure learning of these models in the presence of missing data becomes even more challenging, and to the best our knowledge has never been attempted before. We propose an algorithm, named *LyRiC* (*L*earning H*y*brid *R*elat*i*onal *C*lauses), that learns the structure of a hybrid probabilistic relational model from the database with missing values.

We use the *distributional clauses* (DC) introduced by Gutmann et al. [6] to represent our model as a DC program. DC is a hybrid probabilistic logic programming (PLP) which extends logic programming. In particular, it extends the logic programming language Prolog with continuous as well as discrete probability distributions. *LyRic* learns the structure and parameters of the DC program automatically from the database with missing values. The learned program is subsequently used for autocompletion.

To learn models from databases with missing data, one can use inference in an iterative procedure such as Expectation-Maximization (EM). A major challenge faced while using EM is that the inference

[☆]Please do not distribute

is hard since naive grounding in relational settings results in vast and complex probabilistic networks [7]. The inference mechanism for DC that combines backward reasoning with likelihood weighting [8] performs requisite grounding leading to faster inference. This allows *LyRic* to use the EM algorithm for learning the program. However, the algorithm can still become infeasible in databases where a large number of entities relate to a single entity, for instance, a database in which many clients belong to the same district. It is well known that estimating parameters of large and complex networks from large databases is beyond the scope of EM algorithm [9]. That is why, we also propose an approximate EM algorithm for these databases that works well in practice.

Related Work. There are several works in SRL for learning probabilistic relational models for relational databases, such as probabilistic relational models (PRMs) [10], relational Markov networks (RMNs) [11], and relational dependency networks (RDNs) [4]. PRMs extend Bayesian networks with concepts of objects, their properties, and relations between them. On the other hand, RDNs extend dependency networks and RMNs extend Markov network in the same relational setting. These models are generally restricted to discrete data. To address this shortcoming, several hybrid SRL formalisms were proposed such as hybrid Markov logic networks (HMLNs) [12], hybrid ProbLog (HProbLog) [13], continuous Bayesian logic programs (CBLPs) [14] and hybrid probabilistic relational models (HPRMs) [15]. 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, little attention has been given for designing algorithms for structure learning of hybrid SRL models. The closest to our work is the work on hybrid relational dependency networks (HRDNs) [16] for which structure learning was also studied; however, the approach assumes that the data is fully observed. Two recent probabilistic modeling languages BayesDB [17] and Tabular [18] that is designed for dealing with relational data in the tabular form, automatically fill out missing values in the table. However, they require the model to be defined by the user and do not learn the structure of the model. Furthermore, existing models except for HMLNs and HProbLog are associated with local probability distributions such as CPTs; as a result, it is difficult to read certain independencies such as context-specific independencies (CSIs)[19]. On the contrary, DC can represent such independencies leading to more interpretable DC program. This makes it more explainable and easier to read the regularities present in the database. Moreover, CSIs present in the program are exploited by the DC inference mechanism for efficient inference [20]. An approach for structure learning of DC has also been proposed in [20], but it learns clauses from deterministic data with a single target that cannot deal with the missing data problem.

Contribution. Our contributions in this paper are as follows:

- Representation of a hybrid probabilistic relational model in the DC formalism as a joint model program (JMP).
- *LyRic*, an approach to relational autocompletion based on learning the DC program from the relational database with missing values.
- An approximate EM algorithm for the relational autocompletion problem.
- An extensive empirical evaluation on real-world databases.

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 (DC). In Section 4 we introduce a joint model program (JMP) which represents the hybrid probabilistic relational model in the form of a DC program. Section 5 describes the learning algorithm, which is then evaluated in Section 6.

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.

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 a user's query is denoted by "?".

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 to be either numeric or categorical, and ii) the missing values in entity tables. These are denoted by "–".

The problem that we tackle in this paper is to autocomplete user's queries denoted by "?" in the database [1]. 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 distributional clauses 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 client `C` if `C` has an account `A` and `A` is associated to 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 $\theta = \{\texttt{C/c\_1}\}$ to the clause c from Example 1 yields $c\theta$ which is `clientLoan(c_1,L) ← 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 $\theta = \{\texttt{C/c\_1}, \texttt{M/L}\}$ 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 ($T_\mathrm{P}(I) = I$), 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

Distributional clauses (DCs) are a natural extension of logic programs for representing probability distributions. A DC 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_1) ~ val(55).
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 $\mathcal{D}$. Here, $h \sim val(v)$ means that h has value v with probability 1. 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.

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. [6] 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 $h\theta = v$ is added to the world I. This is also performed for deterministic clauses, adding ground atoms to I whenever the body is true. 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:
$\big[$`hasAccount(c_1,a_1), hasLoan(a_1,l_1), age(c_1)=55`$\big] \rightarrow$
$\big[$`hasAccount(c_1,a_1), hasLoan(a_1,l_1), age(c_1)=55, age(c_2)=40.2`$\big] \rightarrow$
$\big[$`hasAccount(c_1,a_1), hasLoan(a_1,l_1), age(c_1)=55, age(c_2)=40.2, status(l_1)=appr`$\big]$
$\rightarrow$
$\big[$`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)`$\big] \rightarrow$
$\big[$`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`$\big]$.

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

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

In case status of a loan is not defined in the possible world, 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. [6] 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 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 clause program satisfies the *validity conditions*.

Inference in DC is the process of answering probability of a query q given evidence e. Sampling full 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 query. To estimate the probability, samples of the partial possible world, i.e., the set of relevant facts, are generated. The partial possible world

is generated after 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. The weight of i^{th} proof $w^{(i)}$ is given by,

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

A formal description of the inference in DC can be found in Nitti et al. [8].

3.3. 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 what we call *mathematical models* to be used in the bodies of the distributional clauses as well.

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

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` with the first most frequently occurring value in the list. Thus, the body of first clause is true if and only if the most frequently occurring value in this list is `appr`. The body of second clause is true if and only if this list is empty.

3.3.2. DC with a Mathematical Model

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 is to use standard comparison operators in the body of the distributional clauses, e.g. $\geq, \leq$, etc., 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 *mathematical model*, that is, a function which maps outcomes of the random variables in the body of a distributional clause to parameters of the distribution in the head. Formally, a DC with a mathematical 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 the parameter ψ which relates continuous variables in $\{b_1, ..., b_n\}$ with the parameter ϕ in the distribution $\mathcal{D}_\phi$. A distributional program having multiple DCs with mathematical models can now specify a complex probability distribution.

Example 7. Suppose that the credit score of clients depends on the age of the client. Clause 1 is the corresponding DC with a linear model. Additionally, loan status that can take value high or low depends on the amount of the loan. Clause 2 is the corresponding DC with a logistic model.

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

Here, the linear model atom in clause 1 with the parameter $\psi = [10.1, 200]$ implements the linear function $M = 10.1 \cdot Y + 200$, which relates the continuous variable Y and the mean M of the Gaussian distribution in the head. Likewise, the logistic model atom in clause 2 with parameter $\psi = [1.1, 2.0]$ implements the logistic sigmoid function, i.e., $P1 = \sigma(1.1 \cdot Y + 2.0)$ and $P2 = 1 - P1$. The model relates Y to the parameters $\phi = [P1, P2]$ of the discrete distribution in the head.

4. 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 DCs, and (ii) the type of probabilistic relational model that we shall learn.

4.1. 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 instance, 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)`.

4.2. 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 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 intractable.

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 mathematical 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 that specifies a probability distribution over the entire database in Table 1 is shown below:

```
client(ann). client(john). ...
hasAcc(ann,a_11). hasAcc(john,a_10). ...
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).
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),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),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),_), linear([Y],[500,0.8],Mean).
```

5. Learning the Joint Model Program

In this section, we describe our approach *LyRiC*, which learns JMP from the database with missing values. This section is divided into two parts. In the first part, we present the algorithm that learns JMP with negations to handle missing values. In the second part, we present the iterative algorithm that learns JMP by explicitly modeling the missing values, and that starts with the JMP learned in the first part.

5.1. JMP Learner

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

5.1.1. Input

LyRic requires three inputs: a relational database $\mathcal{DB}$ transformed into the set $\mathcal{A}_{\mathcal{DB}} \cup \mathcal{R}_{\mathcal{DB}}$ as introduced in Section 4.1, a declarative bias and a background knowledge $\mathcal{BK}$. 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. *LyRic* 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 $mode(a_1, aggr, (r_1(m_1, m_2), \ldots, r_k(m_k, m_{k+1}), a_2(m_a)))$, where m_i are different modes associated with variables of predicates, *aggr* is the name of aggregation function, r_i are relational predicates, and a_i are attribute predicates. 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, 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 of the algorithm is a (possibly empty) *background knowledge* which can be any set of distributional clauses that satisfy the requirements on distributional clauses in JMPs (as described in Section 4).

Example 9. An example of the transformation of the relational database in Table 1 along with the declarative bias is shown in Figure 5.1.1. We could further assume to have additional domain knowledge, for instance, we know a prior that the probability distribution of the age of clients has normal distribution with mean 40 and standard distribution 5.1, and 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. This background knowledge can be then expressed as a set of clauses shown in the bottom-right of the figure.

```
% 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(C)~gaussian(40,5.1)←client(C).
cliLoan(C,L)←hasAcc(C,A),hasLoan(A,L).
```

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

5.1.2. Learning Task

The aim is to learn a joint model program that best explains the database and the background knowledge. More formally the learning task is:

Definition 1. (Joint model program learning)
Given:

- An input DC program $\mathbb{P}_{\mathcal{IN}}$ consisting of:

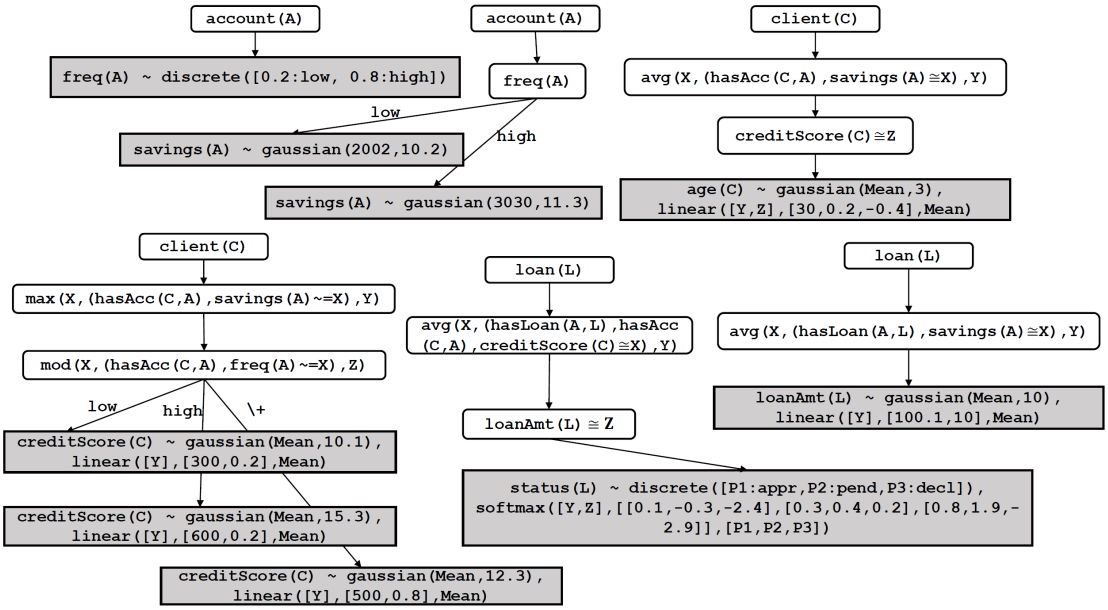


Figure 2: A collection of DLTs corresponding to the JMP in Example 8.

- the set of facts $\mathcal{A}_{\mathcal{DB}} \cup \mathcal{R}_{\mathcal{DB}}$ that represents the database $\mathcal{DB}$,
- the set of facts or/and clauses $\mathcal{BK}$ representing the background knowledge,
- the declarative bias defining the search space $\mathcal{L}_{\mathcal{DB}}$,

- a scoring function $score(\mathbb{P}_{\mathcal{DB}} \mid \mathbb{P}_{\mathcal{IN}})$, measuring the *score* of the program $\mathbb{P}_{\mathcal{DB}}$ given the program $\mathbb{P}_{\mathcal{IN}}$.

Find: A joint model program $\mathbb{P}^*_{\mathcal{DB}} \in \mathcal{L}_{\mathcal{DB}}$ such that $\mathbb{P}^*_{\mathcal{DB}} = \arg\max_{\mathbb{P}_{\mathcal{DB}}} score(\mathbb{P}_{\mathcal{DB}} \mid \mathbb{P}_{\mathcal{IN}})$.

5.1.3. Learning

Clauses in the DC program are *mutually exclusive*, i.e., there is only one distribution defined for each random variable in a possible world (recall the first validity condition of the DC program). In other words, a random variable $h\theta$ is sampled from a unique head $h\theta \sim \mathcal{D}\theta$ of a clause of the program in a possible world. This property of the DC program will allow us to learn the joint model program $\mathbb{P}_{\mathcal{DB}}$ by inducing a tree for each attribute predicate in the database. The program in Example 8 represented as a collection of trees is shown in Figure 2. Notice the resemblance between the program and the collection of trees. In the case of trees also, a random variable is sampled from the distribution defined at a unique leaf node in a possible world.

We call the tree induced by our algorithm, a *distributional logic tree* (DLT). The trees will have to conform to the declarative bias described in the previous section. Next, we specify the DLT learned by *LyRiC*.

Distributional Logic Tree. A 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 mathematical model $\mathcal{H}_\psi$, and each internal node is labeled with an atom b. A path beginning from the root and proceeding to a leaf node then corresponds to a distributional clause of the form $h \sim \mathcal{D}_\phi \leftarrow b_1, ..., b_n, \mathcal{H}_\psi$. A 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),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 mathematical 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.

A leaf node of the tree represents a head of the form $h \sim \mathcal{D}_\phi$ and specifies a mathematical model $\mathcal{H}_\psi$. We now specify the mathematical model that is present in the current implementation of *LyRiC*. In principle, any mathematical model that specifies a well-defined probabilistic model could be used. Depending on the type of random variable defined by h, the distribution and the model can be one of the three types:

- 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 $\mu = w_0 + w_1.V_1 + \cdots + w_i.V_i$ 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 $\mathcal{H}_\psi$ implements a logistic function $q_1 = \frac{1}{1+exp(-w_0-w_1.V_1-\cdots-w_i.V_i)}$ and $\mathcal{D}_\phi$ is a discrete distribution $discrete([q_1{:}true, (1-q_1){:}false])$.
- If h defines a d-valued discrete variable with continuous variables $V_1, \cdots, V_i$, appearing in the branch, then $\mathcal{H}_\psi$ implements a softmax function $q_j = \frac{\exp\left(w_{j_0}+w_{j_1}.V_1+\cdots+w_{j_i}.V_i\right)}{\sum_{k=1}^{d}\exp\left(w_{k_0}+w_{k_1}.V_1+\cdots+w_{k_i}.V_i\right)}$ and $\mathcal{D}_\phi$ is a discrete distribution $discrete([q_1{:}l_1, \cdots, q_d{:}l_d])$, where $l_i \in dom(h)$.

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.

Inducing a Distributional Logic Tree. Induction of the tree for the attribute predicate starts with the 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 6, which will be explained later in this section. The best refinement is selected and incorporated into the internal node and a new node is added to the tree for each branch of the internal node. This procedure is called recursively for the new nodes. For generating refinements of the node, the algorithm employs the refinement operator [21] that specializes the body $\mathcal{Q}$ (a set of atoms in the path from the root to the node) by adding an atom 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 mathematical 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 we are interested in inducing DLT for savings and a path from the root to leaf node in the DLT corresponding 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 few samples of partial possible world obtained by proving the query ?- $h\theta_1, \mathcal{Q}\theta_1$ and few samples obtained by proving the query ?- $h\theta_2, \mathcal{Q}\theta_2$. Suppose, we obtained the following partial possible worlds by proving them twice, where each world is weighted by the weight of the corresponding proof using Equation 3.

```
[savings(a_1)=3000, account(a_1), freq(a_1)=low, deposit(a_1)=30010.1], w^(1)_θ1 = 0.1.
[savings(a_1)=3000, account(a_1), freq(a_1)=high, deposit(a_1)=40410.3], w^(2)_θ1 = 0.
[savings(a_2)=4000, account(a_2), freq(a_2)=low, deposit(a_2)=30211.3], w^(1)_θ2 = 0.5.
[savings(a_2)=4000, account(a_2), freq(a_2)=low, deposit(a_2)=30410.5], w^(2)_θ2 = 0.5.
```

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

$$\begin{aligned}&\ln\mathcal{N}(3000 \mid 30010.1w_1 + w_0, \sigma) \times 0.1 + \ln\mathcal{N}(3000 \mid 40410.3w_1 + w_0, \sigma) \times 0 + \\ &\ln\mathcal{N}(4000 \mid 30211.3w_1 + w_0, \sigma) \times 0.5 + \ln\mathcal{N}(4000 \mid 30410.5w_1 + w_0, \sigma) \times 0.5\end{aligned} \tag{4}$$

Ideally infinite number of proofs are needed to get the exact estimate of parameters, since continuous random variables `deposit(a_1)` and `deposit(a_2)` can take infinite possible values. However, a good approximation of parameters can be obtained with only few proofs in this simple example. It should be clear that the same approach can be used to estimate the parameters for any distributional clauses and/or facts in $\mathbb{P}_{\mathcal{IN}}$.

From the above example, notice that substitutions of the clause are required to estimate parameters of the clause. We now formally define such substitutions. 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 atoms in the clause. In general, parameters of any distribution and/or of any mathematical model at any leaf node can be estimated by maximizing the expectation of log-likelihood $L(\boldsymbol{\varphi} \mid \Theta, \mathbb{P}_{\mathcal{IN}})$, which is given by the following expression,

$$L(\boldsymbol{\varphi} \mid \Theta, \mathbb{P}_{\mathcal{IN}}) = \sum_{\theta_i \in \Theta} \sum_{j=1}^{N} \ln p(h\theta_i \mid \boldsymbol{\varphi}, V\theta_i^{(j)}) w_{\theta_i}^{(j)} \tag{5}$$

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} proof and $V\theta_i^{(j)}$ is j^{th} sample of continuous random variables. For the three simpler mathematical models that we considered, the expression of expectation of log-likelihood is a convex function. *LyRiC* uses scikit-learn [22] to obtain the maximum likelihood estimate $\widehat{\boldsymbol{\varphi}}$ of the parameters.

The Scoring Function. Clauses are scored by the Bayesian Information Criterion (BIC)[23] for selecting among the set of candidate clauses while inducing DLTs. The score of a clause $\mathcal{C}$ is given by,

$$s(\mathcal{C} \mid \mathbb{P}_{\mathcal{IN}}) = 2L(\widehat{\boldsymbol{\varphi}} \mid \Theta, \mathbb{P}_{\mathcal{IN}}) - k \ln |\Theta| \tag{6}$$

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. The scoring function $score(\mathbb{P}_{\mathcal{DB}} \mid \mathbb{P}_{\mathcal{IN}})$ for the program $\mathbb{P}_{\mathcal{DB}}$ is decomposable, i.e., it can be written as,

$$score(\mathbb{P}_{\mathcal{DB}} \mid \mathbb{P}_{\mathcal{IN}}) = \sum_{\mathcal{C}_i \in \mathbb{P}_{\mathcal{DB}}} s(\mathcal{C}_i \mid \mathbb{P}_{\mathcal{IN}}) \tag{7}$$

where $s(\mathcal{C}_i \mid \mathbb{P}_{\mathcal{IN}})$ is a local score measuring how well addition of the clause $\mathcal{C}_i$ in $\mathbb{P}_{\mathcal{DB}}$ explains $\mathbb{P}_{\mathcal{IN}}$. Score decomposablility is important while deciding whether to split a node or not. The score of the clause formed at the node can now be 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 a root node to a leaf node in each DLT corresponds to a clause in $\mathbb{P}_{\mathcal{DB}}$.

5.2. Learning JMP using EM

In the previous section, we allowed negated literals in the body of clauses to handle missing values in the database. The learned joint model program $\mathbb{P}_{\mathcal{DB}}$ can be used to infer the user's query for the autocompletion task. In this section, we discuss a principled algorithm, that is, Expectation-Maximization (EM) to learn JMP in the presence of missing values. EM provides better estimates for parameters of clauses in learned JMP, subsequently, leading to a better estimate of the most likely value for the user's query.

The algorithm starts by first learning a JMP from the input $\mathbb{P}_{\mathcal{IN}}$, by allowing negated literals in the body of clauses. The learned JMP is then used to infer all conditional probability distributions for missing fields in the database. The inferred distributions are added as probabilistic facts to $\mathbb{P}_{\mathcal{IN}}$. Next, a new JMP is learned from the updated $\mathbb{P}_{\mathcal{IN}}$. This process is repeated until convergence, i.e., the clauses and their parameters do not vary by a specific threshold τ. Subsequent JMPs after the first iteration do not contain negated literals in their body. The algorithm is summarised in Algorithm 1.

Algorithm 1: Learning JMP using EM

Input: $\mathbb{P}_{\mathcal{IN}}$: input program, $\mathcal{M}$: a set of missing data in the database $\mathcal{DB}$;
Output: $\mathbb{P}^t_{\mathcal{DB}}$: a learned JMP;
while *t=0,1 ..., until convergence* **do**
  if *t==0* **then**
    learn $\mathbb{P}^0_{\mathcal{DB}}$ from $\mathbb{P}_{\mathcal{IN}}$;
    continue;
  end
  for $q \in \mathcal{M}$ **do**
    infer $q \sim \mathcal{D}$; ▷ `infer the probability distribution` $\mathcal{D}$ `for` q
    assert $q \sim \mathcal{D}$ in $\mathbb{P}_{\mathcal{IN}}$; ▷ `add the probabilistic fact` $q \sim \mathcal{D}$ `in` $\mathbb{P}_{\mathcal{IN}}$
  end
  learn $\mathbb{P}^t_{\mathcal{DB}}$ from $\mathbb{P}_{\mathcal{IN}}$;
  for $q \in \mathcal{M}$ **do**
    retract $q \sim \mathcal{D}$ from $\mathbb{P}_{\mathcal{IN}}$; ▷ `remove the probabilistic fact` $q \sim \mathcal{D}$ `from` $\mathbb{P}_{\mathcal{IN}}$
  end
end

The inference of conditional probability distributions for missing fields is the bottleneck of the algorithm, which requires conditioning over the entire observed database while inferring a conditional probability distribution $\mathcal{D}$ for a single missing field q. The observed fields in the database are evidence $\mathbf{E} \cong \mathbf{e}$ while inferring the distribution $\mathcal{D}$ for the field q, where $\mathbf{e}$ are values of the observed fields $\mathbf{E}$. Fortunately, only some relevant observed fields are required to infer the distribution $\mathcal{D}$. Moreover, from those relevant observed fields, some fields $\mathbf{O} \cong \mathbf{o}$ are needed to be proved; however, only observed values of some fields $\mathbf{Z} \cong \mathbf{z}$ are needed to be known. So, the distribution $\mathcal{D}$ is inferred by first asserting $\mathbf{Z} \sim val(\mathbf{z})$ as facts in the learned $\mathbb{P}_{\mathcal{DB}}$, then proving $\mathbf{O} \cong \mathbf{o}$, and after that proving q. This process allows the backward reasoning based DC inference mechanism to perform requisite grounding. Details of the inference of the distribution $\mathcal{D}$ can be found in Appendix A.

Often real-world database is not highly relational, i.e., instances of an entity type are not related to a large number of instances of other entity types. For instance, a client can have a few accounts, and an account can have a few loan accounts. For such a database, the algorithm is practically feasible, as is also evident by experiments. The requisite grounding and exploitation of context-specific independencies present in the program $\mathbb{P}_{\mathcal{DB}}$ by DC inference [20], make the EM algorithm feasible in the database.

However, when the database becomes highly relational, for example, when many clients belong to the same district, the inference becomes intractable. It is well known that learning parameters of large networks from large datasets are beyond the scope of EM algorithms [9]. To address this problem, we introduce an *approximate EM* algorithm. The difference between the exact and the approximate being, the latter asserts $\mathbf{E} \sim val(\mathbf{e})$ as facts in $\mathbb{P}_{\mathcal{DB}}$ while inferring the distribution $\mathcal{D}$ for the missing field q. It is worth mentioning that asserting $\mathbf{E} \sim val(\mathbf{e})$ as facts is equivalent to conditioning over only $\mathbf{Z} \cong \mathbf{z}$ while inferring $\mathcal{D}$ for q. This makes the inference of conditional probability distributions for missing data tractable since only q is needed to be proved and not evidence $\mathbf{O} \cong \mathbf{o}$. Indeed, the distributions obtained by this process are approximate. However, in experiments, we observe that the approximate EM can predict missing fields with reasonable accuracy.

6. Experiments

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

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

We compared the JMP learned by *LyRiC* with the state-of-the-art algorithm *Learner of Local Models - Hybrid (LLM-H)* introduced by Ravkic et al. [16]. 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 [16].

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 [16].

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: *clients*, *accounts*, *loans* and *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. When performing cross-validation, 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 [16].

We used the same evaluation metrics as used in [16] to evaluate the quality of predictions of JMP.

Evaluation metric. To measure the predictive performance for discrete attributes, multi-class area under ROC curve (AUC_{total}) [24] 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 attributes. To measure the quality of the probability estimates, weighted pseudo-log-likelihood (WPLL) [25] 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 single models for each attribute. We used the same database with the same settings as in [16] 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, a prediction for an attribute was the mode of the probability distribution for the attribute obtained by conditioning over the rest of the test data. Table 3 shows the comparison on University data set divided into train and test set. Numbers for LLM-H are taken directly from [16].

Evaluation	Predicate	LLM-H	DLT	JMP
AUC_{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 *LyRiC* 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 *LyRiC* achieves comparable results as the state-of-the-art algorithm for fully observed data.

6.2. ***Can LyRiC 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 complete training data. The $\mathcal{BK}$ provides additional information about attributes, so learning DLTs in the presence of $\mathcal{BK}$ requires probabilistic inference. In the logic programming literature, rule learners can induce clauses from training facts and background knowledge expressed as the set of clauses. One such example is ProbFOIL+ [26] for the Problog [27]. With similar motivation, we perform this experiment to examine whether *LyRiC* 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.

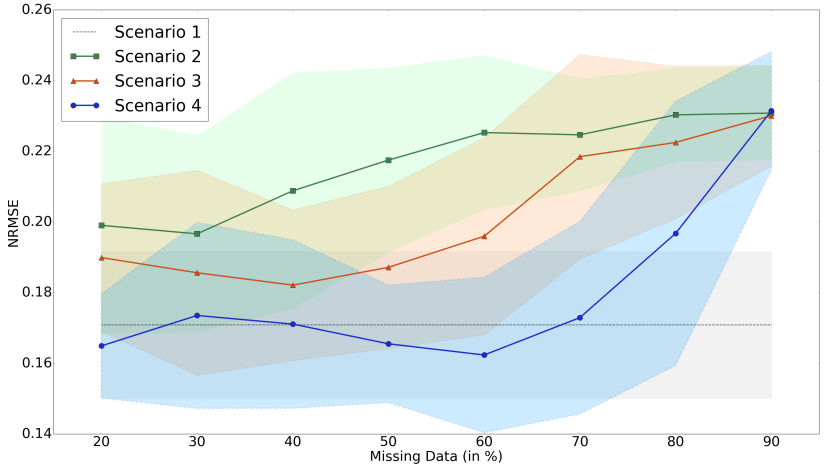


Figure 3: NRMSE of predictions for *monthlyPayment*/1 in the test set versus the percentage of removed fields for the four scenarios.

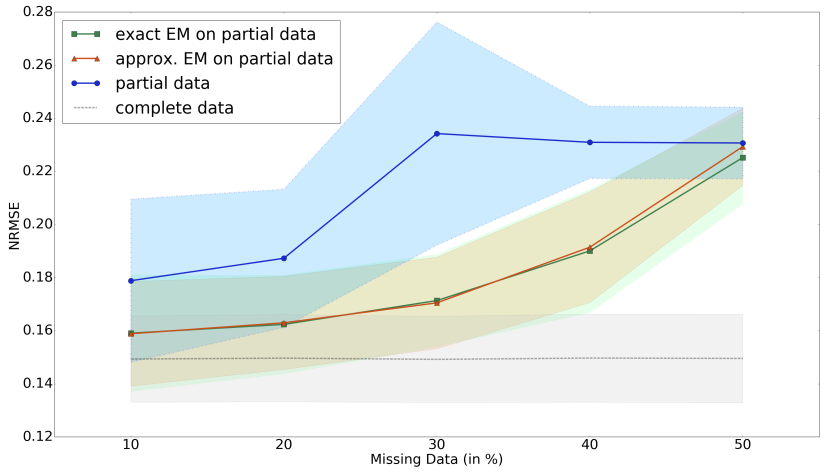
We used the PKDD data set divided into ten folds. Seven folds were used for training the DLT for an attribute; one fold was used for testing that DLT; and two folds were used for generating the background knowledge, which was a set of distributional clauses for all attributes, i.e., a JMP. We considered four 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) The missing fields in the partial set were filled with the mode of the probability distributions for the fields generated from the background knowledge; subsequently, a DLT was induced for the same attribute. 4) Rather than using the mode of the distributions, a DLT for the same attribute was induced from the partial set and the background knowledge itself.

The predictive performance for *monthlyPayment*/1 in the test set in all four scenarios, varying the percentage of removed fields is shown in Figure 3. Much lower NRMSE is observed in the fourth scenario.

We can conclude that *LyRiC* can learn DLT from the training database using additional probabilistic information from $\mathcal{BK}$.

6.3. ***Can LyRic learn the JMP from the relational database when a large portion of the database is missing?***

The joint relational model is advantageous compared to individual models for attributes when the task is to autocomplete a 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 *LyRiC* from one such database. To the best of our knowledge, no system in the literature can learn the joint model from the partially observed database containing continuous as well as discrete attributes. We used the PKDD database and performed two experiments in an attempt to answer the question.



S

Figure 4: NRMSE of predictions for $monthlyPayment/1$ in the test database for the four scenarios versus the percentage of removed fields from the client, loan, and account table. Fields from district table were not removed.

In the first experiment, we removed some percentage of fields from client, loan, and account table of the training database and then learned three probabilistic models to predict attributes in the test database. The first model was just an individual DLT for each attribute. It is worth reiterating that DLT can be learned even when some fields are missing since we allow negation in the body of DC clauses. The second model was a JMP obtained by performing the exact EM. The third model was a JMP obtained by performing the approximate EM. Another JMP was learned from the complete train database. The predictive performance of all models for the $monthlyPayment/1$ attribute is shown in Figure 4. We observe that both exact and approximate EM have similar performance when fewer fields are missing. Exact EM performs slightly better than approximate when more fields are missing. However, both versions of EM perform far better than the individual DLT. As expected, the JMP trained on the complete database achieves the lowest NRMSE.

In the PKDD database, many clients belong to the same district; as a result, if a district attribute is removed, then clients belonging to the same district become dependent in the JMP. This leads to a complex inference of conditional probability distributions for missing fields; consequently, exact EM becomes infeasible. Therefore, district attributes were not removed in the first experiment. In the second experiment, we removed district attributes to evaluate the performance of approximate EM, which is still feasible. The result is shown in Figure 5. With no surprise, much lower NRMSE is obtained with approximate EM than the individual DLT. These two experiments demonstrate that *LyRiC* can learn the JMP even when a large portion of fields in the training database is missing.

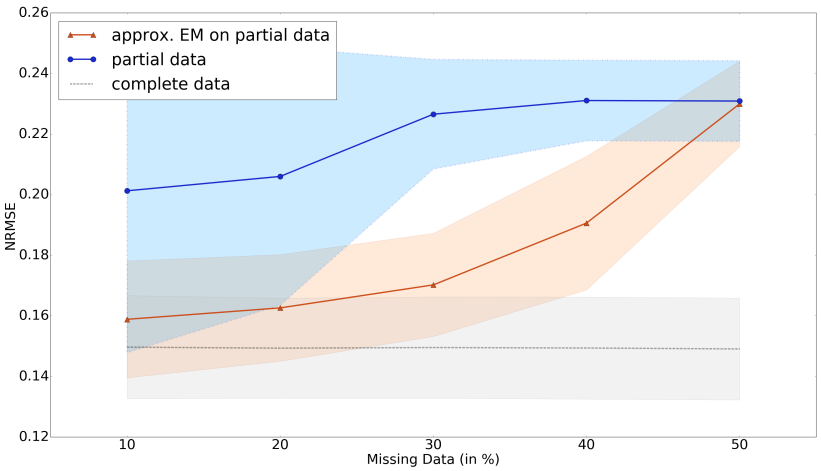


Figure 5: NRMSE of predictions for $monthlyPayment/1$ in the test database for the different scenarios versus the percentage of removed fields from the client, loan, account, and district table.

7. Conclusions

We presented the *LyRic* approach based on probabilistic programming, to tackle the autocompletion task in relational databases with missing fields. Our approach uses the DC formalism to represent an interpretable hybrid relational model in the form of a DC program called joint model program. *LyRic* learns the program automatically from such a database and makes use of additional background knowledge, if available. The program learned from a full observed database achieves state-of-the-art performance. The DC inference mechanism that exploits context-specific independencies present in the program and performs requisite grounding allows *LyRic* to learn the program using EM from the database with missing fields. We introduced an approximate EM strategy that makes the learning of the program for a highly relational database feasible when exact EM becomes infeasible. We demonstrated that the program learned by *LyRic* works well, even when a large portion of the database is missing.

Acknowledgement

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)

References

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

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

[3] D. Koller, N. Friedman, S. Džeroski, C. Sutton, A. McCallum, A. Pfeffer, P. Abbeel, M.-F. Wong, D. Heckerman, C. Meek, et al., Introduction to statistical relational learning, MIT press, 2007.

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

[5] A. Kimmig, S. H. Bach, M. Broecheler, B. Huang, L. Getoor, A short introduction to probabilistic soft logic, in: NIPS Workshop on probabilistic programming: Foundations and applications, Vol. 1, 2012, p. 3.

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

[7] L. Getoor, N. Friedman, D. Koller, A. Pfeffer, B. Taskar, Probabilistic relational models, Introduction to statistical relational learning 8.

[8] D. Nitti, T. De Laet, L. De Raedt, Probabilistic logic programming for hybrid relational domains, Machine Learning 103 (3) (2016) 407–449.
[9] G. Van den Broeck, K. Mohan, A. Choi, A. Darwiche, J. Pearl, Efficient algorithms for bayesian network parameter learning from incomplete data, in: Proceedings of the Thirty-First Conference on Uncertainty in Artificial Intelligence, UAI'15, AUAI Press, Arlington, Virginia, United States, 2015, pp. 161–170.
[10] N. Friedman, L. Getoor, D. Koller, A. Pfeffer, Learning probabilistic relational models, in: IJCAI, Vol. 99, 1999, pp. 1300–1309.
[11] B. Taskar, P. Abbeel, D. Koller, Discriminative probabilistic models for relational data, in: Proceedings of the Eighteenth conference on Uncertainty in artificial intelligence, Morgan Kaufmann Publishers Inc., 2002, pp. 485–492.
[12] J. Wang, P. M. Domingos, Hybrid markov logic networks., in: AAAI, Vol. 8, 2008, pp. 1106–1111.
[13] B. Gutmann, M. Jaeger, L. De Raedt, Extending problog with continuous distributions, in: International Conference on Inductive Logic Programming, Springer, 2010, pp. 76–91.
[14] K. Kersting, L. De Raedt, Adaptive bayesian logic programs, in: International Conference on Inductive Logic Programming, Springer, 2001, pp. 104–117.
[15] P. Narman, M. Buschle, J. Konig, P. Johnson, Hybrid probabilistic relational models for system quality analysis, in: 2010 14th IEEE International Enterprise Distributed Object Computing Conference, IEEE, 2010, pp. 57–66.
[16] I. Ravkic, J. Ramon, J. Davis, Learning relational dependency networks in hybrid domains, Machine Learning 100 (2-3) (2015) 217–254.
[17] V. Mansinghka, R. Tibbetts, J. Baxter, P. Shafto, B. Eaves, Bayesdb: A probabilistic programming system for querying the probable implications of data, arXiv preprint arXiv:1512.05006.
[18] A. D. Gordon, T. Graepel, N. Rolland, C. Russo, J. Borgstrom, J. Guiver, Tabular: a schema-driven probabilistic programming language, in: ACM SIGPLAN Notices, Vol. 49, ACM, 2014, pp. 321–334.
[19] C. Boutilier, N. Friedman, M. Goldszmidt, D. Koller, Context-specific independence in bayesian networks, in: Proceedings of the Twelfth international conference on Uncertainty in artificial intelligence, Morgan Kaufmann Publishers Inc., 1996, pp. 115–123.
[20] D. Nitti, I. Ravkic, J. Davis, L. De Raedt, Learning the structure of dynamic hybrid relational models, in: 22nd European Conference on Artificial Intelligence (ECAI) 2016, Vol. 285, 2016, pp. 1283–1290.
[21] L. De Raedt, Logical and relational learning, Springer Science & Business Media, 2008.
[22] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine Learning in Python , Journal of Machine Learning Research 12 (2011) 2825–2830.
[23] G. Schwarz, et al., Estimating the dimension of a model, The annals of statistics 6 (2) (1978) 461–464.
[24] F. Provost, P. Domingos, Well-trained pets: Improving probability estimation trees.
[25] S. Kok, P. Domingos, Learning the structure of markov logic networks, in: Proceedings of the 22nd international conference on Machine learning, ACM, 2005, pp. 441–448.
[26] L. De Raedt, A. Dries, I. Thon, G. Van den Broeck, M. Verbeke, Inducing probabilistic relational rules from probabilistic examples, in: Twenty-Fourth International Joint Conference on Artificial Intelligence, 2015.
[27] D. Fierens, G. Van den Broeck, J. Renkens, D. Shterionov, B. Gutmann, I. Thon, G. Janssens, L. De Raedt, Inference and learning in probabilistic logic programs using weighted boolean formulas, Theory and Practice of Logic Programming 15 (3) (2015) 358–401.
[28] R. D. Shachter, Bayes-ball: The rational pastime (for determining irrelevance and requisite information in belief networks and influence diagrams), arXiv preprint arXiv:1301.7412.
[29] J. Pearl, Causality, Cambridge university press, 2009.

Appendix A. Inferring Conditional Probability Distributions for Missing Values

The joint model program $\mathbb{P}_{\mathcal{DB}}$ can be used to infer the conditional probability distribution for missing fields in the database. The distribution is inferred by asking a *conditional probability query* to $\mathbb{P}_{\mathcal{DB}}$, which is an expression of the form

$$\text{?- } a_q(t_q) \mid a_{e_1}(t_{e_1}) \cong v_1, \ldots, a_{e_m}(t_{e_m}) \cong v_m, \tag{A.1}$$

abbreviated ?- $q \mid \mathbf{E} \cong \mathbf{e}$, where $m \geq 0$. It asks for the probability distribution for random variable q, called query, given the observed random variables $\mathbf{E}$, called evidence. In the relational database, computing the distribution for a missing field requires conditioning over the entire observed database that forms the evidence set. The query can be trivially answered by proving all evidence [8]. However, this process can easily become intractable if the size of the observed database is huge. Fortunately, to answer the query, only states of some of the evidence are required to be known and some evidence are required to be proved. In this section, we discuss an efficient approach to identify these requisite evidence.

The requisite evidence can trivially be identified by applying Bayes-ball algorithm [28] on a *dependency graph* of $\mathbb{P}_{\mathcal{DB}}$, which is linear-time in the size of the graph.

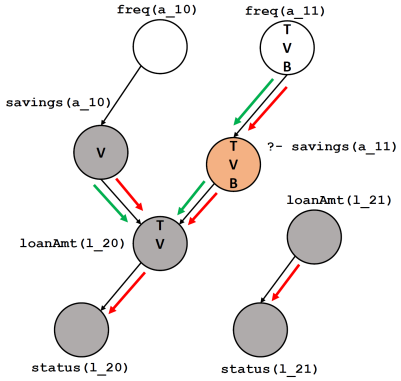


Figure A.6: The probability distribution specified by the program in Example 12 factorizes according to the dependency graph indicated by black edges in this figure. The probability distribution specified by the program after asserting `savings(a_10)` with its value as fact in the program, factories according to the mutilated graph indicated by red directed edges. Gray nodes are observed, and the orange node is the query. Nodes that are marked visited(V), top(T), and bottom (B) after applying Bayes-ball algorithm on the dependency set in Example 13, are indicated. Green directed edges indicate the requisite graph sufficient to answer the query.

Definition 2. (Dependency graph) Let $\mathbb{P}_{\mathcal{DB}}$ be a JMP for a relational database $\mathcal{DB}$. A dependency graph $\mathrm{DG}(\mathbb{P}_{\mathcal{DB}})$ is a directed acyclic graph whose nodes correspond to the ground attribute atoms and there is a directed edge from a node x to a node y if and only if there exists a clause $c \equiv h \sim \mathcal{D}_\phi \leftarrow \mathcal{Q}, \mathcal{H}_\psi \in \mathbb{P}_{\mathcal{DB}}$ and a grounding substitution θ such that $y = h\theta$, $x \cong v \in \mathcal{Q}\theta$.

Example 12. The dependency graph of the following joint model program is shown in Figure A.6.

```
loan(l_20). loan(l_21). account(a_10). account(a_11).
hasLoan(a_10,l_20). hasLoan(a_11,l_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.
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(20000,10.1) <- loan(L), \+avg(X,(hasLoan(A,L), savings(A) ≅
    X),_).
status(L) ~ discrete([P1:appr, P2:pend, P3:decl]) <- loan(L), loanAmt(L) ≅ Y,
    softmax([Y], [[-0.3,-2.4],[0.4,0.2],[1.9,-2.9]],[P1,P2,P3]).
```

The program in the above example specifies a probability distribution over nodes of the graph in Figure A.6. One can show that the probability distribution $p(\mathbf{X})$ specified by the joint model program $\mathbb{P}_{\mathcal{DB}}$ factorizes according to the dependency graph $\mathrm{DG}(\mathbb{P}_{\mathcal{DB}})$, i.e., $p(\mathbf{X})$ can be expressed as the product,

$$p(\mathbf{X}) = \prod_{x \in \mathbf{X}} p(x \mid Pa_x) \tag{A.2}$$

where Pa_x is the set of parents of x according to $\mathrm{DG}(\mathbb{P}_{\mathcal{DB}})$.

The construction of the dependency graph requires completely grounding the program which can become huge for real-world relational databases. So, rather than applying the Bayes-ball algorithm on the graph, *LyRiC* applies the algorithm on a *dependency set*.

Definition 3. (Dependency set) Let $\mathbb{P}_{\mathcal{DB}}$ be a JPM for a relational database $\mathcal{DB}$ and let $h \sim \mathcal{D}_\phi \leftarrow b_1, ..., b_n, \mathcal{H}_\psi$ be a clause in $\mathbb{P}_{\mathcal{DB}}$ and h be of the form $a(T)$. A dependency set $\mathrm{DS}(\mathbb{P}_{\mathcal{DB}})$ is the set of all facts in $\mathbb{P}_{\mathcal{DB}}$ and the set of all definite clauses that can be formed as follows:

- if b_i is the binary predicate $\cong$ of the form $a_1(T) \cong v$ or $a_1(T) \cong V$, then add a definite clause $parent(h, a_1(T)) \leftarrow e(T)$ in $DS(\mathbb{P}_{\mathcal{DB}})$.
- if b_i is the aggregation predicate of the form $aggr(X, Q, V)$, where Q is of the form $(r_1(T, T_1), \ldots, r_k(T_{k-1}, T_k), a_k(T_k) \cong X)$, then add a definite clause $parent(h, a_k(T_k)) \leftarrow e(T), r_1(T, T_1), \ldots, r_k(T_{k-1}, T_k)$ in $DS(\mathbb{P}_{\mathcal{DB}})$.

The dependency set is a first-order representation of the dependency graph. In particular, the following holds: Let $\mathrm{DG}(\mathbb{P}_{\mathcal{DB}})$ be the dependency graph and $\mathrm{DS}(\mathbb{P}_{\mathcal{DB}})$ be the dependency set of the joint model program $\mathbb{P}_{\mathcal{DB}}$. There is a directed edge from a node y to a node x if and only if $parent(x, y) \in \mathrm{LH}(\mathrm{DS}(\mathbb{P}_{\mathcal{DB}}))$, where $\mathrm{LH}(\mathrm{DS}(\mathbb{P}_{\mathcal{DB}}))$ is the least Herbrand model of $\mathrm{DS}(\mathbb{P}_{\mathcal{DB}})$. An example of a dependency set is shown in the next example.

Example 13. The dependency set for the program in Example 12 is shown below:

```
loan(l_20). loan(l_21). account(a_10). account(a_11).
hasLoan(a_10,l_20). hasLoan(a_11,l_20).
parent(savings(A), freq(A)) ← account(A).
parent(loanAmt(L), savings(A)) ← loan(L), hasLoan(A,L).
parent(status(L), loanAmt(L)) ← loan(L).
```

The set of definite clauses in Algorithm 2 implements the same Bayes-ball algorithm as discussed in Algorithm 2 in Shachter [28], but on the dependency set. For answering the query ?- $q \mid \mathbf{E} \cong \mathbf{e}$, *LyRiC* first asserts the fact `evidence(`ε`)` for each $\varepsilon \in \mathbf{E}$. The dependency set is then used to visit parents and children of q recursively. Calling the `visit` predicate with q as child, i.e, `visit(`q`,child)`, marks some evidence as visited, bottom and top. The predicate `asserta` asserts a fact, that is, marks a evidence. An evidence ε is marked visited by asserting the fact `visitMarked(`ε`)`, marked bottom by asserting `bottomMarked(`ε`)` and marked top by asserting `topMarked(`ε`)`. The result after applying Algorithm 2 on the dependency set in Example 13 is shown in Figure A.6. The requisite evidence needed to answer the query are *interventional evidence* and *observational evidence*.

Definition 4. The set of evidence $\mathbf{Z} \subseteq \mathbf{E}$ only marked visited by the Bayes-ball algorithm for answering the query ?- $q \mid \mathbf{E} \cong \mathbf{e}$ in $\mathbb{P}_{\mathcal{DB}}$ is called interventional evidence and the set of evidence $\mathbf{O} \subseteq \mathbf{E}$ marked both visited and top is called observational evidence.

Algorithm 2: A set of definite clauses that implements Bayes-ball algorithm on the dependency set.

```
visit(J,child) ← visited(J), \+evidence(J), top(J), parent(J,P), visit(P,child).
visit(J,child) ← visited(J), \+evidence(J), bottom(J), parent(C,J), visit(C,parent).
visit(J,parent) ← visited(J), evidence(J), top(J), parent(J,P), visit(P,child).
visit(J,parent) ← visited(J), \+evidence(J), bottom(J), parent(C,J),visit(C,parent).
visit(_,_).
visited(J) ← \+visitMarked(J), asserta(visitMarked(J)).
visited(_).
top(J) ← \+topMarked(J), asserta(topMarked(J)).
bottom(J) ← \+bottomMarked(J), asserta(bottomMarked(J)).
```

To answer the query, it is sufficient to consider only a small subgraph of the dependency graph, called *requisite graph*. This graph can be constructed using the requisite evidence. Before discussing this graph, let us look at an important property of JMP. Asserting facts to the program corresponds to *intervention* [29] as in causal models.

Definition 5. (Mutilated graph) Let $\mathrm{DG}(\mathbb{P}_{\mathcal{DB}})$ be a dependency graph of $\mathbb{P}_{\mathcal{DB}}$, $\mathbf{X}$ be the set of all random variables, and $a_{z_1}(t_{z_1}) \cong z_1, \ldots, a_{z_n}(t_{z_n}) \cong z_n$, abbreviated $\mathbf{Z} \cong \mathbf{z}$, an instantiation of random variables $\mathbf{Z} \subseteq \mathbf{X}$. The mutilated graph $\mathrm{DG}(\mathbb{P}_{\mathcal{DB}})^{\mathbf{Z}\cong\mathbf{z}}$ is defined as follows:

- Each node $a_{z_i}(t_{z_i}) \in \mathbf{Z}$ has no parents in $\mathrm{DG}(\mathbb{P}_{\mathcal{DB}})^{\mathbf{Z}\cong\mathbf{z}}$.
- The parents of all other random variables $a_{z_i}(t_{z_i}) \in \mathbf{X} - \mathbf{Z}$ in $\mathrm{DG}(\mathbb{P}_{\mathcal{DB}})^{\mathbf{Z}\cong\mathbf{z}}$ are same as $\mathrm{DG}(\mathbb{P}_{\mathcal{DB}})$.

An example of the mutilated graph is shown in Figure A.6. One can show the following. Let $\mathbb{P}_{\mathcal{DB}}$ be a joint model program and $a_{z_1}(t_{z_1}) \cong z_1, \ldots, a_{z_n}(t_{z_n}) \cong z_n$, abbreviated $\mathbf{Z} \cong \mathbf{z}$, be an instantiation of random variables $\mathbf{Z} \subseteq \mathbf{X}$. An intervened program $\widetilde{\mathbb{P}}_{\mathcal{DB}}$ formed by asserting facts $a_{z_1}(t_{z_1}) \sim val(z_1), \ldots, a_{z_n}(t_{z_n}) \sim val(z_n)$ in the program $\mathbb{P}_{\mathcal{DB}}$, specifies a probability distribution $p(\mathbf{X})$ over $\mathbf{X}$ that factorizes according to the mutilated graph $\mathrm{DG}(\mathbb{P}_{\mathcal{DB}})^{\mathbf{Z}\cong\mathbf{z}}$, where $\mathbf{X}$ is the set of all random variables.

Now we can formally define the requisite graph that is considered by *LyRic* while answering the conditional probability query to $\mathbb{P}_{\mathcal{DB}}$.

Definition 6. (Requisite graph) Let ?- $q \mid \mathbf{E} \cong \mathbf{e}$ be a conditional query to $\mathbb{P}_{\mathcal{DB}}$, $\mathbf{Z}$ be interventional evidence, $\mathbf{O}$ be observational evidence and $\mathrm{DG}(\mathbb{P}_{\mathcal{DB}})^{\mathbf{Z}\cong\mathbf{z}}$ be a mutilated graph. The requisite graph $\mathrm{DG}(\mathbb{P}_{\mathcal{DB}})^{\mathbf{Z}\cong\mathbf{z}}_{q,\mathbf{O}\cong\mathbf{o}}$ is defined as the induced graph of $\{q\} \cup \mathbf{O} \cup \mathbf{Y}$, where $\mathbf{Y}$ is the set of all random variables y, such that, $y \in \mathbf{X}$ and there is a directed path from y to $x \in \{q\} \cup \mathbf{O}$ in $\mathrm{DG}(\mathbb{P}_{\mathcal{DB}})^{\mathbf{Z}\cong\mathbf{z}}$

An example of such a graph is shown in Figure A.6. It can be shown that the requisite graph is sufficient to answer the probabilistic query in JMP, i.e. that the following holds. Let $\mathrm{DG}(\mathbb{P}_{\mathcal{DB}})$ be a dependency graph of $\mathbb{P}_{\mathcal{DB}}$, ?- $q \mid \mathbf{E} \cong \mathbf{e}$ be a conditional query to $\mathbb{P}_{\mathcal{DB}}$, $\mathbf{Z} \cong \mathbf{z}$ be the interventional evidence and $\mathbf{O} \cong \mathbf{o}$ be the observational evidence. The computation of $p(q \mid \mathbf{E} \cong \mathbf{e})$ does not depend on any node x of $\mathrm{DG}(\mathbb{P}_{\mathcal{DB}})$ which is not a member of the requisite graph $\mathrm{DG}(\mathbb{P}_{\mathcal{DB}})^{\mathbf{Z}\cong\mathbf{z}}_{q,\mathbf{O}\cong\mathbf{o}}$.

For answering a query ?- $q \mid \mathbf{E} \cong \mathbf{e}$ to the program $\mathbb{P}_{\mathcal{DB}}$, first of all the Bayes-ball algorithm is applied on the dependency set of $\mathbb{P}_{\mathcal{DB}}$ to obtain the interventional evidence $\mathbf{Z} \cong \mathbf{z}$ and the observational evidence $\mathbf{O} \cong \mathbf{o}$. Next an intervened program $\widetilde{\mathbb{P}}_{\mathcal{DB}}$ is obtained by asserting $\mathbf{Z} \sim val(\mathbf{e})$ as facts in $\mathbb{P}_{\mathcal{DB}}$. Then the probability distribution of the query is obtained by first proving $\mathbf{O} \cong \mathbf{o}$ and then proving q in the intervened program $\widetilde{\mathbb{P}}_{\mathcal{DB}}$. The backward reasoning based DC inference mechanism as discussed in Nitti et. al [20] now performs requisite grounding to answer the conditional probability query.