UNIVERSITEIT TWENTE.

Data Science [201400174]

Course year 2022/2023, Quarter 1B

DATE
November 11, 2022

EXCERPT

Probabilistic Databases and Data Quality [PDBDQ]

TEACHERS
Faizan Ahmed
Ellen-Wien Augustijn
Nacir Bouali
Faiza Bukhsh
Rolf de By
Karin Groothuis-Oudshoorn
Maurice van Keulen
Mahdi Khodadadzadeh
Elena Mocanu
Estefania Talavera
Brenda Voorthuis
Shenghui Wang

COURSE COORDINATOR
Karin Groothuis-Oudshoorn (quartile 1A)
Maurice van Keulen (quartile 1B)
Faizan Ahmed (quartile 2A)

PROJECT OWNERS
Faiza Bukhsh
Karin Groothuis-Oudshoorn
Maurice van Keulen
Elena Mocanu
Mannes Poel
Michel van Putten
Mohsen Jafari Songhori
Luc Wismans

Topic 6

Probabilistic Databases and Data Quality [PDBDQ]

6.1 Introduction

Topic teacher: Maurice van Keulen

Jim Gray coined the term "The Fourth Paradigm" to signify a revolution in science where combining and analyzing data in novel ways has become a main method for research, tackling research questions that could not be answered before. New scientific disciplines have arisen that are primarily based on conducting data-intensive research such as bioinformatics. Extraction, transformation, cleaning, and integration of existing data sources have become primary activities of an e-scientist, often proving harder than originally thought and often consuming most of an e-scientist's time reducing his productivity. A significant reduction in an e-scientist's 'data fiddling' effort has the potential of significantly improving their productivity and as a consequence scientific progress in their disciplines.

It is our conviction that a significant reason for this can be found in one problem: "semantic uncertainty". Sometimes data is subjective, incomplete, not current, or incorrect, sometimes it can be interpreted in different ways, etc. In our opinion, clean correct data is only a special case, hence data management technology should treat data quality problems as a fact of life, not as something to be repaired afterwards. Recent approaches treat uncertainty as an additional source of information which should be preserved to reduce its impact. In a sense, they teach our data processing tools and systems about all forms of doubt and how to live with them. Probabilistic database technology is perfectly suitable for storing, querying, and processing uncertain data.

The topic teaches the most important skills for (a) using probabilistic database technology, and (b) how to represent several kinds of data quality problems as uncertainty in the data.

6.1.1 Global description of the practicum and project

The practicum guides you through a few steps towards being able to model data quality problems as uncertainty in data. And consequently being able to represent the data quality problems *in* the data stored and managed in a probabilistic database enabling querying and analysis. In this way, one can analyse the data while properly paying attention to the consequences of the data quality problems in the analytical results.

The probabilistic database technology we will use is *Probabilistic DataLog*. Hence the practicum will first introduce this technology to you.

- Crash course DataLog
- Querying a probabilistic database
- Creating probabilistic data
- Representing data quality problems as probabilistic data
- Probabilistic data integration

The project is about probabilistically integrating two given data sets with movie data: one extracted from a TV guide source and the other is a selection from IMDB both rather outdated (on purpose, of course). You may, however, also define your own project idea with your own data.

6.1.2 Study material and tools

Study material and additional suggested reading material for this topic can be found on the Canvas module page.

The tool we are going to use is *JudgeD*, a probabilistic DataLog reasoning engine capable of reasoning in an exact way with positive DataLog as well as in an approximate way using MonteCarlo sampling with DataLog with negation.[1]

6.1.3 Deliverables and obligatory items

The practicum assignments specify the deliverables as separate files. Please combine them in one PDF and upload that to Canvas.

6.2 Description of the practical assignments

6.2.1 Installation

Instructions Follow the following steps:

- Open a console window (Windows: see "`http://www.howtogeek.com/235101/10-ways-to-open-the-command-prompt-in-windows-10/`"; Mac: run "Terminal.app")
- Follow the instructions under "Quick start" at "`https://github.com/utdb/judged`".
- Read the rest of the "README.md" of the above URL, so that you know how to execute the variants 'exact' and 'montecarlo' from the command line as well as the options you can use to get the output you'd like. The two variants reason in a different way:
  - **exact**: reasons by sentence manipulation. For each answer, it provides the exact sentence describing the validity of the answer, but it does not calculate probabilities, nor does it handle negation.
  - **montecarlo**: reasons by monte carlo simulation. It approximates the probabilities of the answers, supports negation, but does not provide exact sentences.

□

A description of the JudgeD DataLog language can be found in Section 6.3.

6.2.2 Crash course DataLog

Expected effort 1 hour.

We use the movie example from the lecture. The fully denormalized version is given in file `movie-prac.dl`. Besides giving the facts about the movie "The Namesake" having two genres and three actors, it also defines the two example rules from the lecture: actorofmovie and actororroleofmovie.

[1] it doesn't matter if this terminology doesn't make sense to you.

☞ **6.1**

(a) Add three additional actors from the slide about "The Namesake" as facts to this database.
(b) Lookup the movie on IMDB and add a few keywords and locations.
(c) Write a rule that, given a location, finds actors that have played in movies with that location.
(d) Show that the rule you just made, also works the other way around by issuing a query that, given an actor finds all locations of movies he played in.
(e) Write a rule that given a word finds out what it is: either a movie, an actor, a role, a keyword, or a location.

□

6.2.3 Querying a probabilistic database

Expected effort 2 hours.

We use an example about witnesses who may or may not have seen cars with certain colors and we're going to find out whodunnit. The example comes from the manual of the probabilistic relational database Trio developed at Stanford university.[2] The file is called `trio-crime.dl`. You need to use both the exact version and the montecarlo version; in the former case, you need to comment out the lines starting with '@' (the probability assignments).

☞ **6.2**

(a) (exact) Write a rule that finds all persons who possibly saw a 'toyota'.
(b) (exact) Write a rule that determines suspects, i.e., persons that drive a car with a color that was possibly seen by a witness.
(c) (montecarlo] Switch over to montecarlo version, to determine who is the most suspicious.
(d) (montecarlo) Who is the most suspicious if a blue or green car has been witnessed?
(e) (exact) Under which conditions (random variable assignments) is this guy a suspect? Verify that the probability is roughly correct, by calculating it exactly (calculating the probability from the sentence).
(f) (montecarlo) What is the probability that Amy saw Frank?
(g) (montecarlo) Write a rule that states that Amy did not see Frank. What is the probability that Amy did not see Frank? (the probabilities of this one plus the previous one do not add up to 1).

□

6.2.4 Creating probabilistic data

Expected effort 1 hour.

We continue with the Trio Crime example and expand it with new considerations.

☞ **6.3**

(a) (montecarlo) A new suspect enters the scene: Tom. He may also have been in the vicinity driving a red (0.8) or blue (0.2) Toyota. Who is now the most suspicious person?
(b) (montecarlo) Witnesses are not equally trustworthy. Add new facts to the database `trustworthy(...)` that represent the amount of trust we have in a witness as a probability that the witness is trustworthy (Amy: 0.2; Betty: 1; Cathy: 0.8; Diane: 0.1). Who is now the most suspicious person?

□

6.2.5 Representing data quality problems as probabilistic data

Expected effort 1 hour.

We go back to the movie database in file `movie-prac.dl`. We will model a few data quality problems are uncertainty in the data.

☞ **6.4**

[2] `http://cs.stanford.edu/people/widom/triql.html`

(a) (ambiguous name) The name of actor 2 is uncertain: it is either "Glenne Headley" or "Headly, Glenne" (probabilities 50/50). Modify the database to include this uncertainty. Verify with `genreofactor` that this uncertainty is incorporated. And verify with `genreofmovie` that the uncertainty about actor name does not influence the genre of a movie.
(b) (ambiguous role) The role of actor 1 is uncertain: it is either "Subroto Mesho" or "Subrata Mesho" (probabilities 50/50). Modify the database to include this uncertainty.
(c) (complex abiguity) We add a new actor and role to movie 1, "Irrfan Khan" / "Ashoke Ganguli", but we are unsure which name is the name of the actor and which name is the name of the role (50/50); it could be either way around. Moreover, we are unsure whether or not this actor actually played in the movie (80% confidence (s)he does). Modify the database to include this actor with its uncertainty.
(d) (possible semantic duplicate) It is uncertain whether or not "Tamal Roy Choudhoury" is the same actor as "Tamal Sengupta", i.e., whether they are different actors playing the same role, or one actor for which the name is uncertain namely the one or the other. Modify the database to include this uncertainty. Verify with `genreofactor` that this uncertainty is incorporated.
(e) (vagueness) The year of "Wall-E" is *around 2008*. Modify the database to include this uncertainty.

□

6.2.6 Probabilistic Data Integration

Expected effort 2 hours.

We will now attempt a probabilistic integration of two data items, i.e., attemp to faithfully represent the uncertainty surrounding the integrated data.

☞ **6.5** Slide 7 from the topic lecture shows data about the movie "The Namesake" from two different sources: "TV guide" and "IMDB". Based on a matching rule we determine that it is 80% likely that it is the same movie, with 80/20 probability the integrated data should include 1 or 2 movies, respectively. In the case that it is the same movie, the various attributes need to be integrated. We assume that iff the attribute names are the same, they correspond to the same attribute.

Actors are resolved as follows: if the role of two actors is the same and the role is unique in both sources, then it has to be the same actor, regardless of the name (which obviously creates ambiguity about which actor name is the correct one). Moreover, if the name of two actors is the same, then they also have to be the same actor, regardless of the role (which obviously creates ambiguity about which role name is the correct one). Finally, if two actor names look alike, then they are possibly the same actor (just invent your own probability based on how much alike they are). In all other cases, we assume the actors are different actors.

Manually integrate both data sources each containing the one movie according to these rules and assumptions, i.e., construct the integrated result that faithfully represents the remaining uncertainty.
Hint: start with `movie-prac.dl` which contains TV guide data, complete it based on the TV guide information on the slide, and systematically merge the IMDB-data into it. □

6.3 Informal syntax of JudgeD

In this section, we explain the JudgeD language.

6.3.1 Variables and literals

As customary in logical languages, any identifier that starts with a capital or with an underscore ('_') is a *variable*. A special kind of variable is '_' which is called the *unnamed variable* or *don't care*. Otherwise, it is a *literal*. Furthermore, anything inside double quotes is also considered to be a literal.

- Examples of literals: x, val, "Value", "Green energy", 20.5, 45.
- Examples of variables: X, Var, _var, PERS, _.

A literal is just a value, either a number or a character string. Contrary to many other programming languages, a character string is not always enclosed in quotes: if it doesn't start with a capital, then it is a character string (e.g., val).

As we will see below, variables are used as placeholders for values. Important to note is that multiple occurrences of the same variable inside the same rule, need to refer to the same value. An exception is '_': this variable is used in places where one actually does not care about the value. It can be use multiple times without this behavior.

6.3.2 Terms, facts, and rules

A *term* is composed of a *predicate* and zero or more arguments between brackets. The arguments can be variables and literals.

A *fact* is composed of a *term* terminated by a period ('.'). The arguments should all be literals. A fact specifies something that is true.

For example, "`parent(john,douglas).`" specifies that john is a parent of douglas. Note that both john and douglas are literal values here.

A *rule* is composed of a term (the *head*), the symbol '`:-`', one or more or terms separated by commas (the *body*), and it is terminated by a period ('.'). The arguments in the head and body can be literals and variables. A rule specifies a derived truth: the head is true iff the terms in the body are true. In a sense, the commas in the body indicate a logical 'and'. Two or more rules with the same predicate and the same number of arguments in the head belong together. They specify a logical 'or': if one or more of those rules' body is true, their associated heads are true.

Consider the following example (`examples/ancestor.dl`):

```
% Facts
parent(john, douglas).
parent(bob, john).
parent(ebbon, bob).

% Rules
ancestor(A, B) :- parent(A, B).
ancestor(A, B) :- parent(A, C), ancestor(C, B).
```

In this example, the three facts specify parent-child relationships between 4 people. The 'ancestor' rule specifies that someone is an ancestor of someone else, if the first is a parent of the second, or if there is another person C of whom the first is a parent and who is an ancestor of the second person. Note here the recursivity.

The body of a role may also contain a *unification term*, which is of the form: variable or literal, then the '=' symbol, then another variable or literal. This is true whenever both sides of the equal sign are true. For example, the rule below is equivalent with the last rule above.

```
ancestor(A, B) :- parent(A, C), ancestor(D, B), C=D.
```

All rules must be *safe*, i.e., all variables used in the head should also be present in the body.

Finally, a term in the body may be preceded by a tilde ('~'). This indicates *negation*, i.e., that it is not required that the term is true, but rather that it is false. In case of negation, it is further required that all variables used in the negated term are also present in a positive term of the body (the *guard*).

An example of the use of negation can be found in `examples/power.dl`. The rule below contains negation: a city is unpowered if it is not powered (obviously). The term `city(A)` is needed here as a guard: the 'A' used in the negated term, should also be present in a positive term of the body. Since 'A' is supposed to be a city, we add the term `city(A)`. In a sense, this provides all possible values for 'A'.

```
unpowered(A) :- city(A), ~powered(A).
```

6.3.3 Queries

A *query* is composed of a term terminated by a question mark ('?'). A query requests the JudgeD engine to find all possible instantiations for the variables for which it can be derived that the term is true.

For example, `ancestor(A,douglas)?` will determine all literals A for whom it can be derived that they are ancestors of "douglas". The answer will be `ancestor(bob,douglas)`, `ancestor(john,douglas)`, and `ancestor(ebbon,douglas)`.

6.3.4 Probabilistic facts and rules

A *random variable* represents a probabilistic choice between a number of *alternatives* that each may have a *probability*. A random variable has name starting with a lowercase letter and otherwise composed of alphanumberic characters. It may also contain arguments between brackets (its use is primarily meant for generator syntax explained in the next section). Examples of random variable names: x, same, p12, c(c1).

The alternatives are denoted as random variable assignments of the form random variable name, '=', alternative name. The alternative name can be any literal. Analogous to random variable names, arguments between brackets are also allowed. Examples: heads, tails, 1, 2, 3, same(p12).

A fact or rule can be annotated with a *sentence*: a propositional formula where the atoms are andom variable assignments. Logical operators 'and', 'or', and 'not' can be used as well as brackets to specify precedence. Examples of sentences: `c(c1)=heads and c(c2)=tails`, `same=1`, `same=0 or (same=1 and y=0)`. The sentence is specified a the end of the fact or rule before the terminating period.

One can optionally specify probabilities for the alternatives. The syntax is `@p`, followed by the alternative in brackets, followed by '=' with a probability, terminated by a period. For example, `@p(same=0) = 0.7.` Alternatively, one can also specify a uniform distribution for all alternatives of a specific random variable, for example, `@uniform same.`

The example below specifies facts about what cars (color and type) two witnesses saw: amy is unsure what she saw, namely a blue honda (case x1=1) or a red toyota (case x1=2) or nothing at all (case x1=3); and cathy is also unsure about what she saw, namely a red acura (case x2=1) or nothing at all (case x2=2). Observe that specifying a probability for an alternative while not defining a fact or rule with that alternative in effect reserves probability mass for the absence of the fact or rule.

```
saw(1, amy, blue, honda) [x1=1].
saw(1, amy, red, toyota) [x1=2].

saw(2, cathy, red, acura) [x2=1].

@p(x1=1) = 0.7.
@p(x1=2) = 0.2.
@p(x1=3) = 0.1.

@p(x2=1) = 0.6.
@p(x2=2) = 0.4.
```

6.3.5 Generator syntax

It is also possible to 'generate' as many random variables as needed based on a given predicate. The syntax is '{' followed by one or more facts and rules that typically use random variable names of the form `x(Var1,Var2,...)=Val`, followed by '|' with a predicate that binds these variables Var1, Var2, ..., and Val, terminated by a '}'.

In the example below, we specify 6 objects with two attributes, similarities between pairs of these objects, and then define a generic set of rules that given these produces all possible mergings (predicate `resolved`).

```
object(1,a,15).
object(2,a,16).
```

```
object(3,a,32).
object(4,b,32).
object(5,b,31).
object(6,c,10).

sim(1,2).
sim(3,4).
sim(4,5).

hasdup(I) :- object(O,_,_), sim(O,I).
hasdup(I) :- object(O,_,_), sim(I,O).

% Case for no duplicate
resolved(I,A,B) :-
    object(I,A,B), ~hasdup(I).

{
% Case for similar and duplicate
resolved(I,A,B) :-
    object(I,A,B), object(O,_,_) [s(I)=1].
resolved(I,A,B) :-
    object(I,_,_), object(O,A,B) [s(I)=1].

% Case for similar but no duplicate
resolved(I,A,B) :-
    object(I,A,B) [s(I)=2 and c(I)=1].
resolved(O,A,B) :-
    object(O,A,B) [s(I)=2 and c(I)=2].
@uniform s(I).
@uniform c(I).
| sim(I,O)
}
```