Towards Automatically Setting Language Bias in Relational Learning

Jose Picado
Oregon State University
picadolj@oregonstate.edu

Arash Termehchy
Oregon State University
termehca@oregonstate.edu

Alan Fern
Oregon State University
alan.fern@oregonstate.edu

Sudhanshu Pathak
Oregon State University
pathaks@oregonstate.edu

ABSTRACT

Relational databases are valuable resources for learning novel and interesting relations and concepts. Relational learning algorithms learn the definition of new relations in terms of the existing relations in the database. In order to constraint the search through the large space of candidate definitions, users must specify a language bias. Unfortunately, specifying the language bias is done via trial and error and is guided by the expert's intuitions. Hence, it normally takes a great deal of time and effort to effectively use these algorithms. We report our on-going work on building *AutoMode*, a system that leverages information in the schema and content of the database to automatically induce the language bias used by popular relational learning algorithms.

ACM Reference format:
Jose Picado, Arash Termehchy, Alan Fern, and Sudhanshu Pathak. 2017. Towards Automatically Setting Language Bias in Relational Learning. In *Proceedings of DEEM'17, Chicago, IL, USA, May 14, 2017,* 4 pages.
DOI: http://dx.doi.org/10.1145/3076246.3076249

1 INTRODUCTION

Learning novel concepts and relations over relational databases has attracted a great deal of attention due to its many applications in machine learning and data management [1, 4, 9]. As an example, consider the IMDb database (*imdb.com*) that contains information about movies and people who make them. A schema fragment is shown in Table 1. Given this database, one may want to predict the *BigOpenWeek(movieid)* relation, which indicates that the movie with id *movieid* has made at least two million dollars in its opening week. Machine learning algorithms often assume that data is or can be represented in a single table. This table contains the features that are needed to predict the target relation, e.g., *BigOpenWeek*. Normally, we would be required to hand-engineer such fixed set of features. Each feature would be the result of a query to the database. We would then compute the features for each example in the training data. Finally, we would run an algorithm to learn a model that represents the desired patterns.

movies(id,title,year)	actors(id,name,gender)
movies2genres(movieid,genreid)	genres(id,name)
movies2directors(movieid,directorid)	directors(id,name)
movies2countries(movieid,countryid)	countries(id,name)
award(id, personid, desc)	. . .

Table 1: Fragment of the schema for IMDb data.

```
// Predicate definitions
predDef: BigOpenWeek(movieid)
predDef: movies2directors(movieid,directorid)
predDef: movies(movieid,title,year)
predDef: directors(directorid,directorname)
predDef: actors(actorid,actorname,gender)
predDef: award(awardid,directorid,desc)
predDef: award(awardid,actorid,desc)
                    ...
// Mode definitions
mode: BigOpenWeek(+)
mode: movies(+,-,-)
mode: movies(-,-,+)
mode: movies2directors(+,-)
mode: genres(+,-)
mode: genres(+,#)
                    ...
```

Table 2: A subset of predicate and mode definitions for learning *BigOpenWeek* relation over IMDb data.

There are three obstacles with using such a "table-based approach". First, hand-engineering and transforming features is a tedious process and requires significant expertise. Second, by transforming data into a set of features, we may lose the relational structure in the database, which may be important and relevant to the target relation. Third, the result of the algorithm may be hard to interpret by users.

In contrast to the aforementioned approach, relational machine learning (also called relational learning) aims at learning concepts directly from a relational database [1, 4]. Given a database and training instances of a target relation, relational learning algorithms attempt to find definitions of the target relation according to the existing ones [1]. Learned definitions are usually first-order logical formulas and often restricted to Datalog programs. Relational learning algorithms are also used to learn the structure of statistical relational models, such as Markov Logic Networks, over relational data [4].

DEEM'17, Chicago, IL, USA

DOI: http://dx.doi.org/10.1145/3076246.3076249

Example 1.1. Given the IMDb database and a training data set of movies that have made at least two million dollars in their opening weeks, a relational learning algorithm may learn the following Datalog program for the missing relation *BigOpenWeek* based on the existing relations in the IMDb database:

$$BigOpenWeek(x) \leftarrow movies2genres(x, y), genres(y, \text{drama}),$$
$$movies2directors(x, z), award(t, z, w)$$

which indicates a movie is in *BigOpenWeek* if it is a drama movie made by an award-wining director.

As relational learning algorithms do not require the intermediate step of feature extraction from relational databases, they are potentially easier to use [1]. Nevertheless, as the space of possible definitions (e.g. all Datalog programs) is enormous, relational learning algorithms must employ heuristics to constraint the search space. These heuristics are generally specified through a language bias. One form of language bias is *syntactic bias*, which restricts the structure and syntax of the learned Datalog programs. Relational learning systems usually allow users to specify the syntactic bias through statements called *predicate definitions* and *mode definitions* [1]. Predicate and mode definitions express several types of restrictions on the structure of the learned Datalog programs, such as the relations allowed to be in the Datalog program, whether an attribute can appear as a variable or constant, and whether an attribute can introduce a new variable. Table 2 shows a fragment of predicate and mode definitions used for learning the *BigOpenWeek* relation over the IMDb database. A detailed explanation of these definitions is given in Section 3.

Predicate and mode definitions are necessary to make the search efficient in (statistical) relational learning [1, 4]. Further, it has been shown that predicate and mode definitions significantly reduce the running time of (statistical) relational learning algorithms [5]. If there are no restrictions on the structure of the learned Datalog programs, a learning algorithm may take a long running time or may run out of computational resources. To the best of our knowledge, all (statistical) relational learning systems require some form of language bias to restrict the hypothesis space.

For a relational learning algorithm to be effective and efficient, predicate and mode definitions must encode a great deal of information about the structure of the learned rules [1]. Since a user has to capture various cases of assigning (repeated) variables and constants to the attributes of a relation in the Datalog programs, the number of mode definitions that are written for each relation in the database may be exponential in the number of its attributes. Clearly, it will take a lot of time and effort to write and maintain these definitions, particularly for a relatively complex schema. Furthermore, a user should both know the internals of the learning algorithm and have a relatively clear intuition on the structure of effective Datalog programs for the target relation to set a sufficient degree of restriction. However, there may not be any user that both knows the database concepts, such as schema, and has a clear intuition about the target relation in special domains, such as biology. Hence, learning a relation requires many lengthy discussions between the database/machine learning expert and domain experts. Users normally improve the initial set of definitions via trial and error, which is a tedious and time-consuming process. Also, it is very hard to debug predicate and mode definitions. A slight typo in the predicate and mode definitions may significantly reduce the effectiveness of learning. Such typos are hard to detect, particularly when learning is done over a relatively complex schema. In our conversations with (statistical) relational learning experts, they have called predicate and mode definitions the "black magic" needed to make relational learning work and believe them to be a major reason for the relative unpopularity of these algorithms among users.

Ideally, predicate and mode definitions should be induced automatically. We report the progress of an on-going effort to develop *AutoMode*, a system that leverages the information in the schema and content of the underlying database to induce predicate and mode definitions automatically.

2 BACKGROUND

An *atom* is a formula in the form of $R(u_1, \ldots, u_n)$, where R is a relation symbol. A *literal* is an atom, or the negation of an atom. Each attribute in a literal is set to either a variable or a constant, i.e., value. Variables and constants are also called *terms*. A *Horn clause* (clause for short) is a finite set of literals that contains exactly one positive literal. Horn clauses are also called conjunctive queries. A *Horn definition* is a set of Horn clauses with the same positive literal.

Relational learning algorithms learn Horn definitions from input relational databases and training data. The learned definitions are called the hypothesis, which is usually restricted to non-recursive Datalog definitions without negation, i.e., unions of conjunctive queries, for efficiency reasons. The *hypothesis space* is the set of all possible Horn definitions that the algorithm must explore.

Relational learning algorithms search over the hypothesis space to find the hypothesis that best deduces the training data. More formally, given a database instance I, positive examples E^+, negative examples E^-, and a target relation T, the task of a relational learning algorithm is to find a definition H for T such that $\forall p \in E^+, H \wedge I \models p$ (completeness) and $\forall p \in E^-, H \wedge I \not\models p$ (consistency). Training examples E are usually tuples of a single target relation T, which express positive (E^+) or negative (E^-) examples. The learned definitions are called the hypothesis H. Example 1.1 shows an example of a learned definition for the target relation *BigOpenWeek* over the IMDb database. Relational learning algorithms typically follow one of two approaches to search over the hypothesis space. Top-down algorithms [9] start from the most general hypothesis and employ specialization operators to get more specific hypotheses. On the other hand, bottom-up algorithms [7] start from specific hypotheses that are constructed based on training examples, and use generalization operators to search the hypothesis space [1].

3 LANGUAGE BIAS

Relational learning algorithms employ a language bias to restrict the hypothesis space. In this paper, we focus on syntactic bias, which restricts the structure and syntax of the candidate clauses. Syntactic bias allows the hypothesis space to contain hypotheses that an expert would deem as promising, e.g., hypotheses do not contain meaningless joins. In particular, we focus on predicate and mode definitions [1]. Most relational learning algorithms [1, 7, 9] take as input similar statements to specify a syntactic bias. We now explain the information contained in predicate and mode definitions.

Candidate relations: An atom can be placed in a candidate clause only if there is at least one predicate and mode definition with relation symbols equal to the relation symbol of the atom.

Predicate definitions: Predicate definitions assign a *semantic type* (*type* for short) to each attribute in a database relation. For instance, for the relation *movies*, the predicate definition `movies (movieid,title,year)` in Table 2 indicates that the first, second, and third attributes in the relation *movie* are of types *movieid*, *title*, *year*, respectively. It is possible to assign multiple types to an attribute. For example the predicate definitions `award(awardid, directorid,desc)` and `award(awardid,actorid,desc)` indicate that the attribute *personid* in relation *award* in Table 1 belongs to both types *directorid* and *actorid*. Two attributes can be assigned the same variable or constant in a clause only if they belong to the same type. For instance, given predicate definitions `movies(movieid,title,year)` and `movies2directors (movieid,directorid)`, relations *movies* and *movies2directors* can join only on attributes *movies.id* and *movies2directors.movieid* because they have the same type `movieid`. Two relations cannot join if they do not contain at least one attribute with the same type. If all attributes in all relations have the same type, then all relations can join with each other on every attribute. This can make the learning algorithm very inefficient. Therefore, types set restrictions so that the algorithm can run efficiently.

Mode definitions: Mode definitions set restrictions on the terms that appear in the atoms of a clause. This is done by writing some statements for each relation, and assigning a symbol to each attribute in the relation. The symbols + and − indicate that a term should be a variable. Symbol + indicates that a term must be an existing variable, except for the atom in the head of a Horn clause. Symbol − indicates that a term can be an existing variable or a new variable, i.e., an existentially quantified variable. For instance, the mode definition `movies(+,-,-)` in Table 2 indicates that the first term must be an existing variable and the next two terms can be either existing or new variables. In some domains, it is useful to have atoms that contain constants. Symbol # indicates that a term should be a constant. For instance, the mode definition `genres(+,#)` in Table 2 indicates that the first term must be an existing variable and the second term must be a constant. Each atom in every clause in the hypothesis space must satisfy at least one mode definition in the list of mode definitions. Thus, having definitions `movies(+,-,-)` and `movies(-,-,+)` in Table 2 implies that in atoms with relation *movies*, either attribute *id* or attribute *year* must be set to an existing variable in the clause. In each case, the rest of the attributes in *movies* will be set to new or existing variables. Similarly, definitions `genres(+,-)` and `genres(+,#)` in Table 2 indicate that the attribute *name* in the relation *genres* can be set to either a constant or a variable. If all attributes are allowed to be variables and constants, then the algorithm may generate very long clauses. This is because it would generate multiple atoms for a single tuple. For instance, given tuple *genres(g1,drama)* and mode definitions `genres(+,-)`, `genres(-,+)`, `genres(+,#)` and `genres(#,+)`, a clause would would contain atoms *genres(y,u)*, *genres(y,*drama*)* and *genres(*g1*,u)*. Restricting the number of attributes that are allowed to be constants may drastically reduce the number of atoms in a clause, hence making the learning algorithm more efficient.

movies2directors(gravity,cuaron)	award(a1,cuaron,oscar)
movies2directors(revenant,inarritu)	award(a2,inarritu,bafta)
movies2genres(gravity,g1)	genres(g1,drama)
movies2genres(revenant,g1)	

Table 3: Fragments of the IMDb database.

4 AUTOMODE SYSTEM

4.1 Finding Candidate Relations

AutoMode assumes that the clauses in the hypothesis space can contain any relation in the schema. Therefore, AutoMode reads the schema information, such as the list of relations and attribute names, from the RDBMS and generates at least one predicate definition and one mode definition for each relation.

4.2 Generating Mode Definitions

We have implemented AutoMode over the bottom-up relational learning system Castor [7]. In the first step, given a subset of positive examples, Castor constructs the most specific clause that covers each example, relative to the database. These clauses are called *bottom clauses*. In the second step, it applies a generalization operator to these clauses to create a clause that generalizes all of them.

Example 4.1. The following clauses C_1 and C_2 are the bottom clauses associated with positive examples e_1 = *BigOpenWeek(revenant)* and e_2 = *BigOpenWeek(gravity)*, respectively, relative to the database instance shown in Table 3.

C_1 = *BigOpenWeek*(revenant) ← *movies2genres*(revenant,g1), *genres*(g1,drama), *movies2directors*(revenant,inarritu), *award*(a2,inarritu,bafta).

C_2 = *BigOpenWeek*(gravity) ← *movies2genres*(gravity,g1), *genres*(g1,drama), *movies2directors*(gravity,cuaron), *award*(a1,cuaron,oscar).

When constructing a bottom clause, AutoMode forces at least one variable in an atom to be an existing variable, i.e., appears in previously added atoms, to avoid generating Cartesian products in the clause. AutoMode generates one mode definition for each attribute of each relation. In each mode definition, AutoMode assigns the + symbol to exactly one attribute and the − symbol to all remaining attributes. This means that all attributes are allowed to have new variables, except the attribute with symbol +.

Instead of indicating which attributes can be constants in mode definitions, AutoMode's approach is to postpone this decision to the generalization step in learning. The *least general generalization* (*lgg*) operator takes as input two clauses C_1 and C_2, and generates the clause C that is more general than C_1 and C_2, but the least general such clause [1]. While doing this, it automatically generates new variables to generalize constants in C_1 and C_2.

The *lgg* operator is defined as follows. The *lgg* of two clauses C_1 and C_2 is the set of pairwise *lgg* operations of compatible atoms in C_1 and C_2. Two atoms are compatible if they have the same relation name and same polarity (either positive or negative). Let $R(t_1, \cdots, t_n)$ and $R(s_1, \cdots, s_n)$ be two atoms. The *lgg* of two atoms is $lgg(R(t_1, \cdots, t_n), R(s_1, \cdots, s_n)) = R(lgg(t_1, s_1), \cdots, lgg(t_n, s_n))$. The *lgg* of two atoms with different relation symbol or opposite polarity is undefined. The *lgg* of two identical terms (either variables

or constants) is $lgg(t, t) = t$. The *lgg* of two distinct terms (either constants or variables) is $lgg(t, s) = v_{ts}$, where v_{ts} is a new variable associated with t and s.

Example 4.2. Given the bottom clauses C_1 and C_2 shown in Example 4.1, $lgg(C_1, C_2)$ is

$$BigOpenWeek(v_{rg}) \leftarrow$$
$$movies2genres(v_{rg}, \text{g1}), genres(\text{g1}, \text{drama}),$$
$$movies2directors(v_{rg}, v_{ic}), award(v_{a_2a_1}, v_{ic}, v_{bo}).$$

The current implementations of *lgg* only run over small databases that satisfy certain restrictions, which do not generally hold for real-world databases [1]. The reason is that the *lgg* operator generates very large clauses whose evaluations take prohibitively long time. This is because the size of a clause generated by $lgg(C_1, C_2)$, is bounded by $|C_1| \times |C_2|$, where $|C_i|$ is the number of literals in C_i. Then, the size of a clause resulting from multiple *lgg* operations can grow exponentially with the number positive examples.

Our implementation of the *lgg* operator over Castor [7] is able to scale to databases with about two thousand tuples and without any restrictions. It is able to do so thanks to the following reasons. Castor is implemented on top of the in-memory database VoltDB. The indexing mechanism in VoltDB allows Castor to efficiently build bottom clauses. Further, Castor uses a subsumption engine that allows it to evaluate clauses efficiently. We are in the process of scaling our system to larger databases.

5 EXPERIMENTS

We have run experiments over the UW-CSE database, which contains information about a computer science department (*alchemy.cs. washington. edu/data/uw-cse*). This is a benchmark database used in relational learning literature. We learn the target relation *advisedBy(stud, prof)*, which indicates that student *stud* is advised by professor *prof*.

We have used four ways of setting syntactic bias in addition to AutoMode. **Baseline** generates predicate and mode definitions automatically. It assigns the same types to all attributes in all relations and allows every attribute to be a variable or a constant. **Baseline without constants** is the same as the baseline, except that it does not allow any attribute to be a constant. **Manual tuning** uses the the syntactic bias written by an expert. The expert had to learn the schema and go through several trial and error phases by running the underlying learning system and observing its results to write the predicate and mode definitions manually. **Aleph** [8] is a relational learning system able to induce predicate and mode definitions from data. Because AutoMode does not currently have the functionality to generate predicate definitions, it simply assigns the same types to all attributes in all relations. We run experiments on a 2.6GHz Intel Xeon E5-2640 processor and 50GB of main memory.

Table 4 compares AutoMode with the aforementioned methods in terms of precision, recall and learning time. The predicate and mode definitions extracted by Aleph over restrict the hypothesis space, resulting in a precision and recall of 0. AutoMode is generally more accurate than the baseline methods and manual tuning. However, it is less efficient. We note that for manual tuning, the expert has to spend additional time to understand the database and (re)write the predicate and mode definitions via trial and error.

Measure	Baseline	Baseline (w/o const.)	Manual tuning	Aleph	AutoMode
Precision	0.78	0.96	0.93	0	0.96
Recall	0.49	0.48	0.54	0	0.52
Time (s)	30	3.8	8	5	44

Table 4: Results of learning relations over the UW-CSE database.

6 RELATED WORK

There has been interest in reducing the user input in relational learning systems. Similar to AutoMode, the work in [6] and the relational learning system Aleph [8] induce predicate and mode definitions from data. The mode definitions generated by these methods may over restrict the hypothesis space because they require multiple attributes to be existing variables. The algorithm in [6] does not generate mode definitions that allow constants. Aleph allows all attributes to be constants. AutoMode assigns constants only to some attributes and based on data.

The algorithm in [3] discovers semantic types by first converting attributes of objects into unary predicates and then searching for unary predicates that semantically refer to the same attribute. The algorithm MILE [2] induces mode definitions from training examples of the target relation. Their setting assumes that examples consist of Horn clauses. This is different from our setting, where examples are ground atoms. There has been a growing interest in developing relational learning algorithms that scale to large databases [9]. These algorithms must restrict the hypothesis space through language bias or use a more restricted data model.

7 FUTURE WORK

We have presented our on-going work on building the AutoMode system. We plan to use heuristics to generate mode definitions that scale to large databases. Further, we plan to use database constraints to find attribute types and induce predicate definitions. We believe that this work is crucial to make relational learning useful to ordinary users, as they do not need to write language bias manually.

REFERENCES

[1] Luc De Raedt. 2010. *Logical and Relational Learning* (1st ed.). Springer Publishing Company, Incorporated.
[2] Nicola Di Mauro, Floriana Esposito, Stefano Ferilli, and Teresa M. A. Basile. 2004. An Algorithm for Incremental Mode Induction. In *IE/AIE*.
[3] Stefano Ferilli, Floriana Esposito, Teresa M. A. Basile, and Nicola Di Mauro. 2004. Automatic Induction of First-Order Logic Descriptors Type Domains from Observations. In *ILP*.
[4] Lise Getoor and Ben Taskar. 2007. *Introduction to Statistical Relational Learning*. MIT Press.
[5] Marcin Malec, Tushar Khot, James Nagy, Erik Blasch, and Sriraam Natarajan. 2016. Inductive Logic Programming meets Relational Databases: An Application to Statistical Relational Learning. In *ILP*.
[6] Eric Mccreath and Arun Sharma. 1995. Extraction of Meta-Knowledge to Restrict the Hypothesis Space for ILP Systems. In *Australian Joint Conference on AI*.
[7] Jose Picado, Arash Termehchy, and Alan Fern. 2017. Schema Independent Relational Learning. In *SIGMOD Conference*.
[8] Ashwin Srinivasan. 2004. *The Aleph Manual*.
[9] Qiang Zeng, Jignesh M. Patel, and David Page. 2014. QuickFOIL: Scalable Inductive Logic Programming. *PVLDB* 8 (2014), 197–208.