Towards Automatically Setting Language Bias in Relational Learning

Jose Picado
Oregon State University
picadolj@oregonstate.edu

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

Arash Termehchy
Oregon State University
termehca@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:
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.

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].

1.3 SOME DEFINITIONS

Table 1: Fragment of the schema for IMDb data.

<table>
<thead>
<tr>
<th>Table 1: Fragment of the schema for IMDb data.</th>
</tr>
</thead>
<tbody>
<tr>
<td>movies(id,title,year)</td>
</tr>
<tr>
<td>actors(id,name,gender)</td>
</tr>
<tr>
<td>movies2countries(movieid,countryid)</td>
</tr>
<tr>
<td>countries(id,name)</td>
</tr>
<tr>
<td>movies2directors(movieid,directorid)</td>
</tr>
<tr>
<td>directors(id,name)</td>
</tr>
<tr>
<td>movies2genres(movieid,genreid)</td>
</tr>
<tr>
<td>genres(id,name)</td>
</tr>
<tr>
<td>award(id, personid, desc)</td>
</tr>
</tbody>
</table>
|                                            ...
|                                            |
| // 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: genres(+,-)                             |
| mode: genres(+,#)                             |

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

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 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 contain atoms genres(g1,a), genres(g1,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,marritu) | award(a2,marritu,bafka) |
| movies2genres(gravity,g1) | genres(g1,drama) |
| movies2genres(revenant,g1) | award(a1,cuaron,oscar) |

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 C1 and C2 are the bottom clauses associated with positive examples e1 = BigOpenWeek(revenant) and e2 = BigOpenWeek(gravity), respectively, relative to the database instance shown in Table 3.

\[
C_1 = \text{BigOpenWeek}(\text{revenant}) \leftarrow \text{movies2genres}(\text{revenant},g1), \text{genres}(g1,\text{drama}), \text{movies2directors}(\text{revenant},\text{marritu}), \text{award}(a2,\text{marritu},\text{bafka}).
\]

\[
C_2 = \text{BigOpenWeek}(\text{gravity}) \leftarrow \text{movies2genres}(\text{gravity},g1), \text{genres}(g1,\text{drama}), \text{movies2genres}(\text{gravity},\text{cuaron}), \text{award}(a1,\text{cuaron},\text{oscar}).
\]

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 C1 and C2, and generates the clause C that is more general than C1 and C2, but the least general such clause [1]. While doing this, it automatically generates new variables to generalize constants in C1 and C2.

The lgg operator is defined as follows. The lgg of two clauses C1 and C2 is the set of pairwise lgg operations of compatible atoms in C1 and C2. Two atoms are compatible if they have the same relation name and same polarity (either positive or negative). Let R(t1, · · · , t\text{n1}) and R(s1, · · · , s\text{n1}) be two atoms. The lgg of two atoms is lgg(R(t1, · · · , t\text{n1}), R(s1, · · · , s\text{n1})) = R(lgg(t1, s1), · · · , lgg(t\text{n1}, s\text{n1})). 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 \( \text{lgg}(t, t) = t \). The \( \text{lgg} \) of two distinct terms (either constants or variables) is \( \text{lgg}(t, s) = \nu_{ts} \), where \( \nu_{ts} \) 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, \( \text{lgg}(C_1, C_2) \) is

\[
\text{BigOpenWeek}(v_{rg}) \leftarrow \\
\text{movies2genres}(v_{rg}, g1), \text{genres}(g1, \text{drama}), \\
\text{movies2directors}(v_{rg}, v_{ic}), \text{award}(v_{ai1}, v_{ic}, v_{bo}).
\]

The current implementations of \( \text{lgg} \) only run over small databases that satisfy certain restrictions, which do not generally hold for real-world databases [1]. The reason is that the \( \text{lgg} \) operator generates very large clauses whose evaluations take prohibitively long time. This is because the size of a clause generated by \( \text{lgg}(C_1, C_2) \), 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 \( \text{lgg} \) operations can grow exponentially with the number of positive examples.

Our implementation of the \( \text{lgg} \) 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 (alchemycs washington.edu/data/uw-cse). This is a benchmark database used in relational learning literature. We learn the target relation \( \text{advisedBy(stud, prof)} \), which indicates that student \( \text{stud} \) is advised by professor \( \text{prof} \).

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 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.

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