AutoMode: Relational Learning
With Less Black Magic

Jose Picado, Sudhanshu Pathak, Arash Termehchy, Alan Fern

School of EECS, Oregon State University

{picadolj,pathaks,termehca,alan.fern}@oregonstate.edu

Abstract

Relational databases are valuable resources for learning novel and interesting relations and concepts. Relational learning algorithms learn the Datalog 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 tune the algorithm by specifying 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. In particular, it is hard to find a user that knows computer science concepts, such as database schema, and has a reasonable intuition about the target relation in special domains, such as biology. We propose *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 systems. We show that AutoMode delivers the same accuracy as using manually-written language bias by imposing only a slight overhead on the running time of the learning algorithm.

1 Introduction

Learning novel concepts or relations over relational databases has attracted a great deal of attention due to its applications in machine learning and data management [12, 3, 4, 2]. Consider the UW-CSE database (*alchemy.cs.washington.edu/data/uw-cse*), which contains information about a computer science department and its schema fragments are shown in Table 1. One may want to predict the new relation *advisedBy(stud,prof)*, which indicates that the student *stud* is advised by professor *prof*. Given the UW-CSE database and positive and negative training examples of the *advisedBy* relation, relational learning algorithms attempt to find a definition of this relation in terms of the existing relations in the database [2, 11]. Learned definitions are usually first-order logic formulas and often restricted to Datalog programs. For example, a relational learning algorithm may learn the following Datalog program for the *advisedBy* relation:

$$advisedBy(x, y) \leftarrow publication(z, x), publication(z, y)$$

which indicates that a student is advised by a professor if they have been co-authors of a publication.

student(stud)	professor(prof)
inPhase(stud,phase)	hasPosition(prof,position)
courseLevel(course,level)	taughtBy(course,prof,term)
ta(course,stud,term)	publication(title,author)

Table 1: Schema fragments for the UW-CSE database.

Predicate definitions	Mode definitions
student(T1)	advisedBy(+,+)
inPhase(T1,T2)	student(+)
professor(T3)	inPhase(+,-)
hasPosition(T3,T4)	inPhase(+,#)
publication(T5,T1)	professor(+)
publication(T5,T3)	hasPosition(+,-)
...	

Table 2: A subset of predicate and mode definitions for learning *advisedBy* relation.

Relational learning algorithms can exploit the relational structure of the data, making them useful for domains where structure of data is important. Moreover, their learned definitions are interpretable and easy to understand. Relational learning has several applications in database management and machine learning, such as learning database queries and learning the structure of statistical relational models [2].

As the space of possible definitions (e.g. all Datalog programs) is enormous, relational learning algorithms must employ heuristics to constrain 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* [2]. 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 two relations can join. Table 2 shows a fragment of predicate and mode definitions used for learning the *advisedBy* relation over the UW-CSE database. A detailed explanation of these definitions is given in Section 2. To the best of our knowledge, all (statistical) relational learning systems require some form of syntactic bias to restrict the hypothesis space. It has been shown that predicate and mode definitions significantly reduce the running time of (statistical) relational learning algorithms [5].

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 Datalog programs [2]. A user should both know the internals of the learning algorithm and the schema of the input database 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, particularly in specific domains such as biology. Hence, learning a relation requires many lengthy

discussions between the database/machine learning expert and domain experts. Furthermore, the number of predicate and mode definitions of is generally large and hard to debug and maintain. Users normally improve the initial set of definitions via trial and error, which is a tedious and time-consuming process. Hence, it takes a lot of time and effort to write and maintain these definitions, particularly for 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.

We propose AutoMode, a system that leverages the information in the schema and content of the database to automatically generate predicate and mode definitions. We show that the predicate and mode definitions produced by AutoMode deliver the same accuracy as the manually written and tuned ones while imposing only a modest running-time overhead over large real-world databases.

2 Background

2.1 Relational Learning

An *atom* is a formula in the form of $R(e_1, \ldots, e_n)$, 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. Variable 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.

A relational learning algorithm learns a Horn definition from input relational databases and training data. The learned definition is 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 candidate Horn definitions that the algorithm can explore.

Relational learning algorithms search over the hypothesis space to find a definition that covers as many positive examples as possible, while covering the fewest possible negative examples. Relational learning algorithms generally follow either a top-down or a bottom-up approach. Top-down algorithms start with an empty definition and iteratively add literals to the definition until the definition cannot be improved [12]. Bottom-up algorithms first construct the most specific clause that covers a given positive example, and then generalize this clause to cover more positive examples, while covering the fewest possible negative examples [11].

2.2 Castor Relational Learning System

We implement AutoMode on top of Castor [11], a bottom-up relational learning system. As many relational learning algorithms, Castor follows a covering approach. Algorithm 1 depicts Castor's learning algorithm. It constructs one clause at a time using the *LearnClause* function. If the clause satisfies the minimum criterion, Castor adds

---

Algorithm 1: Castor's cover-set algorithm.

   Input : Database instance I, positive examples E^+, negative examples E^-
   Output: A Horn definition H

1 $H = \{\}$
2 $U = E^+$
3 while U *is not empty* **do**
4 $C = LearnClause(I, U, E^-)$
5   **if** C *satisfies minimum criterion* **then**
6 $H = H \cup C$
7 $U = U - \{c \in U | H \wedge I \models c\}$
8 return H

---

the clause to the learned definition and discards the positive examples covered by the clause. It stops when all positive examples are covered by the learned definition.

Castor's *LearnClause* function contains two main steps. In the first step, it constructs the most specific clause that covers a positive example, relative to the database. This clause is called the *bottom-clause*. In the second step, it generalizes the bottom-clause to cover more positive examples, while covering the fewest possible negative examples. We now explain each step in more detail.

2.2.1 Bottom-clause construction

Castor uses the following algorithm to compute the bottom-clause associated with example e and relative to database I [6, 10]. The algorithm maintains a hash table that maps constants to variables. It first assigns new variables to constants in example e, and inserts the mapping from constants to variables in the hash table. It creates the head of the bottom-clause by replacing the constants in e with their assigned variables. Then, the algorithm searches for all tuples in the database that contain at least one constant in the hash table. For each tuple, it creates one or more literals with the same relation name as the tuple and adds the literals to the body of the bottom-clause. The literals may contain variables or constants, as specified by mode definitions, which are explained in Section 3. If the literal (tuple) has a new constant that should replaced by a variable, the algorithm assigns a new variable to the constant and adds the new mapping to the hash table. In the following iterations, the algorithm selects tuples in the database that contain the newly added constants to the hash table and adds their corresponding literals to the clause. The algorithm finishes after the user-specified number of iterations.

Example 2.1 *Consider the database instance I shown in Table 3 and a positive example $e = advisedBy(alice,bob)$. Then, the bottom-clause associated with e and relative to I, after one iteration, is:*

$$advisedBy(x, y) \leftarrow student(x), inPhase(x, u), professor(y),$$
$$hasPosition(y, v), publication(z, x), publication(z, y).$$

student(alice)	professor(bob)	publication(p1,alice)
student(john)	professor(mary)	publication(p1,bob)
inPhase(alice,post_quals)	hasPosition(bob,assistant_prof)	publication(p2,john)
inPhase(john,post_quals)	hasPosition(mary,associate_prof)	publication(p2,mary)

Table 3: Fragments of the UW-CSE database.

The hash table created by the algorithm contains the following mapping from constants to variables: { *alice* → x, *bob* → y, *p1* → z, *post_quals* → u, *assistant_prof* → v}.

2.2.2 Generalization

After building the bottom-clause associated with a given positive example, Castor generalizes the clause to cover more positive examples. Castor uses the same generalization strategy as ProGolem [8]. Castor uses the *asymmetric relative minimal generalization* (*armg*) operator to generalize clauses. Castor performs a beam search to select the best clause generated after multiple applications of the *armg* operator. More formally, given clause C, Castor randomly picks a subset E_S^+ of positive examples to generalize C. For each example $e' \in E_S^+$, Castor uses the *armg* operator to generate a candidate clause C', which is more general than C and covers e'. It then selects the highest scoring candidate clauses to keep in the beam and iterates until the clauses cannot be improved.

We now explain the *armg* operator in detail. Let C be the bottom-clause associated with example e, relative to I. Let e' be another example. L_i is a *blocking atom* iff i is the least value such that for all substitutions θ where $e' = T\theta$, the clause $C\theta = (T \leftarrow L_1, \cdots, L_i)\theta$ does not cover e', relative to I [8]. Given the bottom-clause C and a positive example e', *armg* drops all blocking atoms from the body of C until e' is covered. Because *armg* drops literals from the clause, it is guaranteed that the size of the clause reduces when doing generalization.

The bottom-clause is the most specific hypothesis that belongs to the hypothesis space. Therefore, any generalization of it is also in the hypothesis space. It has been shown that bottom-up algorithms are generally more effective than top-down algorithms [2, 11].

3 Language Bias

In relational learning algorithms, language bias restricts the structure and syntax of the generated clauses. Language bias is specified through predicate and mode definitions [2].

Predicate definitions assign one or more *types* to each attribute in a database relation. In a candidate clause, two relations can be joined over two attributes (i.e., attributes are assigned the same variable) only if the attributes have the same type. For instance, in Table 2, the predicate definition `student(T1)` indicates that the attribute in relation *student* is of type `T1`, and the predicate definition `inPhase(T1,T2)` indicates that the first and second attributes of relation *inPhase* are of type `T1` and `T2`,
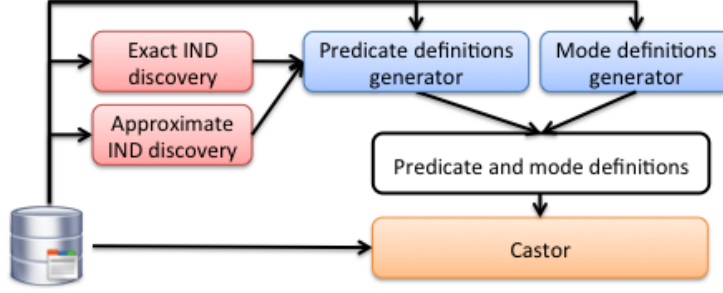
Figure 1: AutoMode is implemented on top of Castor.

respectively. Therefore, relations *student* and *inPhase* can be joined on attributes *student[stud]* and *inPhase[stud]*. It is possible to assign multiple types to an attribute. For example the predicate definitions `publication(T5,T1)` and `publication(T5,T3)` indicate that the attribute *author* in relation *publication* belongs to both types `T1` and `T3`. Predicate definitions restrict the joins that can appear in a candidate clause: two relations can be joined only if their attributes share a type.

Mode definitions indicate whether a term in an atom should be a new variable (i.e., existentially quantified variable), an existing variable (i.e., appears in a previously added atom), or a constant. They do so by assigning one or more symbols to each attribute in a relation. 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. For instance, the mode definition `inPhase(+,-)` in Table 2 indicates that the first term must be an existing variable and the second term can be either an existing or a new variable. Symbol $\#$ indicates that a term should be a constant. For instance, the mode definition `inPhase(+,#)` indicates that the second term must be a constant. Each atom in a candidate clause must satisfy at least one mode definition.

4 AutoMode System

Figure 1 shows the components of the AutoMode system. We implement AutoMode on top of Castor [11], a bottom-up relational learning system. AutoMode reads and extracts the information about the schema of the underlying database from the RDBMS. It then generates predicate and mode definitions in a pre-processing step. Castor uses these definitions to learn the definition of some target relation. The same predicate and mode definitions can be used to learn different target relations.

4.1 Generating Predicate Definitions

Let R and S be two relation symbols in the schema of the underlying database. Let $R(e_1, \cdots, e_n)$ and $S(o_1, \cdots, o_m)$ be two atoms in a clause C. Let e_i be the term in attribute $R[A]$ and o_j be the term in attribute $S[B]$, and let e_i and o_j be assigned the

6

same variable or constant. That is, clause C joins R and S on A and B. Clause C is satisfiable only if these attributes share some values in the input database. Typically, the more frequently used joins are the ones over the attributes that participate in inclusion dependencies (INDs), such as foreign-key to primary-key referential constraints. AutoMode uses INDs in the input database to find which attributes, among all relations, share the same type. Let X and Y be sets of attribute names in R and S, respectively. Let I_R and I_S be the relations of R and S in the database. Relations I_R and I_S satisfy *exact IND* (*IND* for short) $R[X] \subseteq S[Y]$ if $\pi_X(I_R) \subseteq \pi_Y(I_S)$. If X and Y each contain only a single attribute, the IND is a *unary IND*. Given IND $R[X] \subseteq S[Y]$ in a database, the database satisfies unary IND $R[A] \subseteq S[B]$, where $A \in X$ and $B \in Y$. INDs are normally stored in the schema of the database. If they are not available in the schema, one can extract them from the database content. AutoMode uses the Binder algorithm [9] to discover INDs from the database, shown by the **Exact IND discovery** box in Figure 1, and generates all unary INDs implied by them.

We have observed that using exact INDs is not enough for generating helpful predicate definitions. For instance, to learn the accurate definition for relation *advisedBy* presented in Section 1, Castor must join relations *publication*, *student*, and *professor* on attributes *publication[author]*, *student[stud]*, and *professor[prof]*. But, the UW-CSE database does not satisfy INDs *publication[author] $\subseteq$ student[stud]* or *publication[author] $\subseteq$ professor[prof]* because *publication[author]* contains both students and professors. Hence, AutoMode also uses *approximate INDs* to assign types to attributes. In an *approximate unary IND* $(R[A] \subseteq S[B], \alpha)$, one has to remove at least α fraction of the distinct values in $R[A]$ so that the database satisfies $R[A] \subseteq S[B]$ [1]. Approximate INDs are not usually maintained in schema and are discovered from the database content. We have implemented a program to extract approximate INDs from the database, shown by the **Approximate IND discovery** box in Figure 1. We use a relatively high error rate, 50%, for the approximate INDs to enlarge Castor hypothesis space.

After discovering unary exact and approximate INDs, AutoMode runs Algorithm 2 to generate a directed graph called *type graph*, which it then uses to assign types to attributes. First, it creates a graph whose nodes are attributes in the input schema and has an edge between each pair of attributes that participate in an exact or approximate IND (lines 1-3). Figure 2 shows an example of the type graph containing a subset of the attributes in the UW-CSE schema, where edges corresponding to exact and approximate INDs are shown by solid and dashed lines, respectively. If there are both approximate INDs $(R[A] \subseteq S[B], \alpha_1)$ and $(S[B] \subseteq R[A], \alpha_2)$, AutoMode uses only the one with lower error rate. The algorithm then assigns a new type to every node in the graph without any outgoing edges (lines 4-5). For example, it assigns new types T1, T3, and T5 to *student[stud]*, *professor[prof]*, and *publication[title]*, respectively, in Figure 2. If there are cycles in the type graph, the algorithm assigns the same new type to all nodes in each cycle (lines 6-7). Next, it propagates the assigned type of each attribute to its neighbors in the reverse direction of edges in the graph until no changes are made to the graph (lines 8-11). For example, in Figure 2, the algorithm propagates type T1 to *inPhase[stud]* and *ta[stud]* and attribute *publication[author]* inherits types T1 and T3 from *student[stud]* and *professor[prof]*, respectively. Because the error rates of approximate INDs accumulate over multiple edges in the graph, AutoMode propagates

7

Algorithm 2: Algorithm to generate the type graph.

Input : Schema $\mathcal{S}$ and all unary INDs Σ.
Output: Type graph G.

1   create graph $G = (V, E)$ where V contains a node for each attribute in the schema and $E = \varnothing$
2   **foreach** *IND $R[A] \subseteq S[B] \in \Sigma$* **do**
3   add edge $v \rightarrow u$ to E, where v and u correspond to attributes $R[A]$ and $S[B]$, respectively
4   **foreach** *node $u \in V$ without outgoing edges* **do**
5   generate new type T and set $types(u) = \{T\}$
6   **foreach** *cycle $K \subseteq V$* **do**
7   generate new type T and set $types(u) = \{T\} \, \forall u \in K$
8   **repeat**
9   **foreach** *$v \rightarrow u \in E$ where $types(u) \neq \varnothing$* **do**
10   set $types(v) = types(v) \cup types(u)$
11   **until** *no changes in G*
12   return G

Figure 2: A fragment of the type graph for UW-CSE database.

types only once over edges that correspond to approximate INDs.

Given the resulting graph, for each relation, AutoMode computes the Cartesian product of the types associated with its attributes. For each tuple in this Cartesian product, it produces a predicate definition for the relation. For instance, given the type assignment in Figure 2, AutoMode generates predicate definitions `publication(T5,T1)` and `publication(T5,T3)` for the *publication* relation.

4.2 Generating Mode Definitions

4.2.1 Setting Variables and Constants Through Generalization

One approach to assigning constants or variables to terms in a clause 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 [7]. 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 $R(e_1, \cdots, e_n)$ and $R(o_1, \cdots, o_n)$ be two atoms. Then, the *lgg* of two atoms is $lgg(R(e_1, \cdots, e_n), R(o_1, \cdots, o_n)) = R(lgg(e_1, o_1), \cdots, lgg(e_n, o_n))$. 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 $lgg(e, e) = e$. The *lgg* of two distinct terms (either constants or variables) is $lgg(e, o) = v_{eo}$, where v_{eo} is a new variable associated with e and o.

Example 4.1 *Consider the following bottom-clauses C_1 and C_2 associated with examples e_1 = advisedBy(alice,bob) and e_2 = advisedBy(john,mary), respectively, relative to the database instance shown in Table 3.*

$$C_1 = advisedBy(alice, bob) \leftarrow$$
$$\quad student(alice), inPhase(alice, post\_quals), professor(bob),$$
$$\quad hasPosition(bob, assistant\_prof), publication(p1, alice),$$
$$\quad publication(p1, bob).$$
$$C_2 = advisedBy(john, mary) \leftarrow$$
$$\quad student(john), inPhase(john, post\_quals), professor(mary),$$
$$\quad hasPosition(mary, associate\_prof), publication(p2, john),$$
$$\quad publication(p2, mary).$$

Then, $lgg(C_1, C_2)$ is

$$advisedBy(v_{aj}, v_{bm}) \leftarrow$$
$$\quad student(v_{aj}), inPhase(v_{aj}, post\_quals), professor(v_{bm}),$$
$$\quad hasPosition(v_{bm}, v_{ap}), publication(v_p, v_{aj}), publication(v_p, v_{bm}).$$

A relational learning system can generate bottom-clauses for a subset of positive examples and iteratively apply the *lgg* operator over these bottom-clauses to create a clause that generalizes all of them [7]. Because *lgg* makes the decisions on setting attributes to constants or variables, there is no need to write mode definitions for them. However, the *lgg* operator generates very large clauses whose evaluations take prohibitively long time. The reason is that 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 of clauses from which it is generalized. This is opposed to the *armg* generalization operator described in Section 2.2.2, which guarantees that the size of the clause reduces after each generalization step.

The current implementations of *lgg* only run over small databases that satisfy certain restrictions, which do not generally hold for real-world databases [7]. We have implemented the *lgg* operator over Castor. Castor implements several techniques that allow running *lgg* over small databases (about two thousand tuples) without any restrictions. These techniques include efficiently evaluating clauses by using a subsumption

Name	#R	#T	#P	#N
UW-CSE	9	1.8K	102	204
HIV	80	14M	2K	4K
IMDb	46	8.4M	1.8K	3.6K

Table 4: No. of relations (#R), tuples (#T), positive examples (#P), and negative examples (#N) for each dataset.

engine, minimizing clauses by removing syntactically redundant literals, and reusing coverage tests [11]. However, it does not scale to medium or large databases. Therefore, AutoMode uses mode definitions to assign constants or variables to terms in a clause, as described in the next section.

4.2.2 Setting Variables and Constants Through Mode Definitions

AutoMode lets every attribute of every relation be a variable. However, it 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. For each attribute A in relation R, AutoMode generates a mode definition for R where attribute A is assigned the $+$ symbol and all other attributes are assigned the $-$ symbol. This means that all attributes are allowed to have new variables except the attribute with symbol $+$. For instance, AutoMode generates the mode definitions `publication(+,-)` and `publication(-,+)` for relation *publication* in Table 1.

AutoMode uses the following rule to determine whether an attribute can be a constant. If the number of distinct values in an attribute is below some given threshold, AutoMode allows the attribute to be a constant. This threshold is a hyper-parameter, but it has a relatively intuitive meaning. For each relation R in the database, AutoMode finds all attributes in R that can be constants using the aforementioned rule. Then, it computes the power set $\mathbf{M}$ of these attributes. For each non-empty set $M \in \mathbf{M}$, AutoMode generates a new set of mode definitions where it assigns $+$ and $-$ symbols as described above, except for the attributes in M, which are assigned the $\#$ symbol. For example, AutoMode finds that the number of values in attribute *phase* of relation *inPhase* in Table 1 is smaller than the input threshold. Then, this attribute can be constant and AutoMode generates the mode definition `inPhase(+,#)` for relation *inPhase*.

5 Empirical Results

5.1 Evaluating AutoMode

We have run experiments over three datasets whose information is shown in Table 4. The UW-CSE database 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*. The HIV database contains structural information about chemical compounds (*wiki.nci.nih.gov/display/NCIDTPdata*).

We learn the target relation *antiHIV(comp)*, which indicates that compound with id *comp* has anti-HIV activity. In this dataset, the positive examples are compounds known to have anti-HIV activity, while negative examples are compounds known to lack anti-HIV activity. The IMDb database (*imdb.com*) contains contains information about movies and people who make them. We learn the target relation *dramaDirector(dir)*, which indicates that person with id *dir* has directed a drama movie. Over the UW-CSE and IMDb databases, we generate the negative examples by using the closed-world assumption, and then sample to obtain twice as many negative examples as positive examples.

We use four methods of setting language bias. **Baseline** assigns the same types to all attributes and allows every attribute to be a variable or a constant. **Baseline without constants** is the same as the baseline method, except that it does not allow any attribute to be a constant. **Manual tuning** uses the language bias written by an expert who has knowledge of the relational learning system and knows how to write predicate and mode definitions. 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. The expert has written 36, 165, and 112 predicate and mode definitions for the UW-CSE, HIV, and IMDb databases, respectively. **AutoMode** uses the automatically generated predicate and mode definitions, as described in this paper. The original databases do not contain INDs. Therefore, AutoMode calls the IND discovery tools shown in Figure 1. The pre-processing step of AutoMode to extract INDs takes 2 seconds, 45 minutes, and 53 minutes over the UW-CSE, HIV, and IMDb databases, respectively. We set the hyper-parameter in AutoMode used to determine whether an attribute can be a constant (Section 4.2) to 5 for UW-CSE, 20 for HIV, and 400 for IMDb.

We compare the quality of the learned definitions using the metrics of *precision* and *recall* [2]. Let the set of true positives for a Horn definition be the set of positive examples in the testing data that are covered by the Horn definition. The precision of a Horn definition is the proportion of its true positives over all examples covered by the Horn definition. The recall of a Horn definition is the number of its true positives divided by the total number of positive examples in the testing data. Precision and recall are between 0 and 1, where an ideal definition delivers both precision and recall of 1. We also compare the running time of Castor to show the impact of predicate and mode definitions. We perform 10-fold cross validation for HIV and IMDb datasets and 5-fold cross validation for UW-CSE. We evaluate precision, recall, and running times, showing the average over the cross validation. We run experiments on a 2.3GHz Intel Xeon E5-2670 processor, running CentOS Linux 7.2 with 50GB of main memory.

The results are shown in Table 5. We analyze the results of each setting.
Baseline: Over the UW-CSE database, Castor is less accurate and efficient compared to other settings. Over the HIV database, Castor does not terminate after 36 hours. Over the IMDb database, Castor is killed by the kernel because of extreme use of resources. By allowing every attribute to be a constant, every value in the database, even if it's a non-predictive value, may appear in a literal as a constant. Therefore, the generated bottom-clause contains a large number of literals, many of which are not useful for learning a definition for the target relation. For instance, the first bottom-clause created when running over the IMDb databases contains on average 1255 literals. Further, by

11

Dataset	Measure	Baseline	Baseline (w/o const.)	Manual tuning	AutoMode
UW-CSE	Precision	0.76	0.96	0.93	0.93
	Recall	0.50	0.48	0.54	0.54
	Time	47s	6.6s	11s	10.8s
HIV	Precision	-	0.72	0.77	0.77
	Recall	-	0.91	0.89	0.89
	Time	>36h	20h	14.7m	32.2m
IMDb	Precision	-	0.68	1	1
	Recall	-	0.51	1	1
	Time	-	9.2h	2.7m	6.9m

Table 5: Results of learning relations over UW-CSE, HIV, and IMDb data (h=hours, m=minutes, s=seconds).

assigning the same type to all attributes in all relations, it allows all relations to join with each other on any attribute, resulting in a long running time.

Baseline without constants: Over the UW-CSE database, this setting is the most efficient, and obtains competitive precision and recall compared to manual tuning and AutoMode. Over the HIV database, Castor is able to learn a reasonable definition, but less accurate than manual tuning and AutoMode. However, because this setting uses the same type for all attributes, it allows all relations to join with each other over any attribute, resulting in a long running time. Over the IMDb database, the perfect definition for the target relation *dramaDirector* contains a constant. However, in this setting, constants are not allowed. Therefore, Castor learns other definitions which are significantly less accurate compared to manual tuning or AutoMode.

Manual tuning: Castor obtains similar precision and recall using manual tuning and AutoMode. Castor with manual tuning is very efficient. However, an expert had to spend significant amount of time tuning the language bias. Further, a non-expert user would not be able to specify this bias.

AutoMode: AutoMode is more effective than the baselines, and as effective as manual tuning. However, AutoMode is slightly less efficient manual tuning. This is because the manually written predicate and mode definitions provide a more restricted hypothesis space than the ones generated by AutoMode. Thus, Castor has to explore a larger hypothesis space when using AutoMode. Nevertheless, the overhead in the running time is about 18 minutes for the HIV database and 4 minutes for the IMDb database, which is a reasonable overhead for saving expert's time and the enterprise's financial resources that pay the machine learning expert. There is no overhead over the UW-CSE database. Hence, we argue that automating the generation of predicate and mode definitions with the cost of a modest overhead in performance is a reasonable trade-off. Further, AutoMode enables non-experts to use relational learning systems easily.

Measure	AutoMode+LGG
Precision	0.96
Recall	0.52
Time	62.6s

Table 6: Results of learning relations over UW-CSE using AutoMode+LGG (s=seconds).

5.2 AutoMode + LGG

In Section 4.2.1, we described a method for setting variables and constants in the literals of a clause without using mode definitions. This method uses the *lgg* generalization operator. We have implemented the *lgg* operator over Castor. In this section, we evaluate the AutoMode system over the implementation of Castor that uses the *lgg* operator. We refer to this system as AutoMode+LGG. In this case, AutoMode is only used to generate predicate definitions. We run AutoMode+LGG over the UW-CSE database. The results are shown in Table 6. AutoMode+LGG obtains a better precision but worse recall compared to AutoMode. However, it is significantly less efficient. This is because, as explained in Section 4.2.1, the size of clauses grows as they are generalized. Therefore, it takes a longer time to evaluate and search through these clauses. AutoMode+LGG does not scale to larger databases, such as the HIV and IMDb databases.

6 Conclusion

We have proposed AutoMode, a system that automatically induces the language bias used by relational learning algorithms. On one hand, empirical results showed the importance of specifying predicate and mode definitions. Without specifying a language bias, a learning algorithm may take hours to run on moderate-sized databases or may crash because of reaching the limit of computational resources. On the other hand, compared to language bias manually written by an expert, AutoMode imposes a slight overhead in running time while still being accurate. Hence, AutoMode saves time and financial resources over manual tuning considering the total learning pipeline. We believe that AutoMode paves the way for non-expert users to use relational learning systems.

7 Acknowledgements

References

[1] Z. Abedjan, L. Golab, and F. Naumann. Profiling relational data: A survey. *The VLDB Journal*, 24(4), 2015.

[2] L. De Raedt. *Logical and Relational Learning*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[3] T. Kraska et al. MLbase: A Distributed Machine-learning System. In *CIDR*, 2013.

[4] A. Kumar, J. Naughton, and J. M. Patel. Learning generalized linear models over normalized data. In *SIGMOD*, 2015.

[5] M. Malec, T. Khot, J. Nagy, E. Blasch, and S. Natarajan. Inductive logic programming meets relational databases: An application to statistical relational learning. In *ILP*, 2016.

[6] S. Muggleton. Inverse Entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13:245–286, 1995.

[7] S. Muggleton and C. Feng. Efficient induction of logic programs. In *New Generation Computing*. Academic Press, 1990.

[8] S. Muggleton, J. C. A. Santos, and A. Tamaddoni-Nezhad. Progolem: A system based on relative minimal generalisation. In *ILP*, volume 5989, 2009.

[9] T. Papenbrock, S. Kruse, J.-A. Quiané-Ruiz, and F. Naumann. Divide & conquer-based inclusion dependency discovery. *Proc. VLDB Endow.*, 8(7):774–785, Feb. 2015.

[10] J. Picado, P. Ataei, A. Termehchy, and A. Fern. Schema independent and scalable relational learning by Castor. *Proc. VLDB Endow.*, 9(13):1589–1592, Sept. 2016.

[11] J. Picado, A. Termehchy, A. Fern, and P. Ataei. Schema Independent Relational Learning. In *SIGMOD*, 2017.

[12] Q. Zeng, J. M. Patel, and D. Page. Quickfoil: Scalable inductive logic programming. *PVLDB*, 2014.