Learning Efficiently Over Heterogeneous Databases: Sampling and Constraints to the Rescue

Jose Picado
Oregon State University
picadolj@oregonstate.edu

Arash Termehchy
Oregon State University
termehca@oregonstate.edu

Sudhanshu Pathak
Oregon State University
pathaks@oregonstate.edu

ABSTRACT

Given a relational database and training examples for a target relation, relational learning algorithms learn a definition for the target relation in terms of the existing relations in the database. We propose a relational learning system called CastorX, which learns efficiently across multiple heterogeneous databases. The user specifies connections and relationships between different databases using a set of declarative constraints called matching dependencies (MDs). Each MD connects tuples across multiple databases that are related and can meaningfully join but the values of their join attributes may *not* be equal due to the different representations of these values in different databases. CastorX leverages these constraints during learning to find the information relevant to the training data and target definition across multiple databases. Since each tuple in a database may be connected to too many tuples in other databases according to an MD, the learning process will become very slow. Hence, CastorX uses sampling techniques to learn efficiently and output accurate definitions.

ACM Reference Format:
Jose Picado, Arash Termehchy, and Sudhanshu Pathak. 2018. Learning Efficiently Over Heterogeneous Databases: Sampling and Constraints to the Rescue. In *DEEM'18: International Workshop on Data Management for End-to-End Machine Learning, June 15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3209889.3209899

1 INTRODUCTION

Users often need to discover interesting and novel concepts over domains with multiple entities and relations. For instance, consider the IMDb database (*imdb.com*), which contains information about movies and people who make them, and whose schema fragments are shown in Table 1. Given this database, a user may want to find the definition for the new relation *highGrossing(title)*, which indicates that the movie with title *title* is high grossing. Typical statistical learning methods generally assume that the (training) data is independently and identically distributed. This assumption often holds in the domains with a single entity or relation but generally violated in datasets with multiple relations due to the correlation between different relations in the database [2].

DEEM'18, June 15, 2018, Houston, TX, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5828-6/18/06…$15.00
https://doi.org/10.1145/3209889.3209899

movies(id,title,year)	movies2releasedate(id,month,year)
movies2genres(id,name)	genres(id,name)
movies2directors(id,directorId)	director(id,name)

Table 1: Schema fragments for the IMDb database.

highBudgetMovies(title)
movies2totalGross(title,gross)

Table 2: Schema fragments for Box Office Mojo.

Given a relational database and training examples for a new target concept or relation, *(statistical) relational learning* algorithms attempt to learn an (approximate) definition of the target relation in terms of existing relations in the database [2]. Definitions are usually restricted to Datalog programs for the sake of efficiency. For instance, the user who wants to learn a definition for the new target relation *highGrossing* using the IMDb database may provide a set of high grossing movies as positive examples and a set of low grossing movies as negative examples to a relational learning algorithm. Given the IMDb database and these examples, the algorithm may learn the following definition:

$$highGrossing(x) \leftarrow movies(y, x, z), movies2genres(y, \text{`}comedy\text{'}), movies2releasedate(y, \text{`}June\text{'}, u),$$

which indicates that high grossing movies are often released in June and their genre is *comedy*. Since relational learning algorithms leverage the structure of the database directly to learn new relations and their results are interpretable, they have been widely used in database management and machine learning applications, such as query learning, entity resolution, and information extraction [2].

The information in a domain is usually spread across several databases. For example, IMDb does *not* contain the information about the budget or total grossing of the movies. Using this information may help the user to learn a more accurate definition for the *highGrossing* relation as high grossing movies may have high budgets. This information can be found from another database Box Office Mojo (BOM) (*boxofficemojo.com*), whose schema fragments are shown in Table 2. Currently, users have to first (manually) integrate the IMDb and BOM databases, and then learn over the integrated database. It is well-established that integrating databases is an extremely difficult, time-consuming, and labor-intensive process. For instance, the *titles* of the same movie in IMDb and BOM have different formats and there is no simple rule to match titles of the same movie. Thus, one has to write a relatively complex program to match movies in IMDb and BOM databases. More importantly, the user does *not* always need to integrate databases to learn a definition for a target relation. For instance, consider a user who wants to learn the definition of a target relation *collaborators(dir1, dir2)*, which indicates that directors whose names are *dir1* and *dir2* have co-directed a movie. This user can learn an effective definition by using only the data in the IMDb database.

In this paper, we describe our progress in building a relational learning system called CastorX, which learns a target definition over multiple heterogeneous databases. Instead of integrating databases as a preprocessing step, we follow a different approach. Users can guide CastorX on how to join and match tuples in different databases using a set of declarative constraints called matching dependencies (MDs) [3]. MDs provide information about the attributes across multiple databases that are related and can meaningfully join but their values may not match exactly. For example, there is an MD between the *highBudgetMovies[title]* attribute in *BOM* and the *movies[title]* attribute in *IMDb*. One can use this MD to find the tuples in *BOM* that contain the information about the budgets of movies in *IMDb*. CastorX leverages these constraints to learn a definition for the target relation. Generally, a tuple in one database may (approximately) match a lot of tuples in another relation in the same database or another database. It would be extremely expensive to explore all matched tuples. CastorX uses sampling methods to learn effective definitions over multiple large databases efficiently. CastorX presents the final definitions to the user as if the learning has been performed over an integrated database.

2 BACKGROUND

2.1 Relational Learning

An *atom* is a formula in the form of $R(u_1, \ldots, u_n)$, where R is a relation symbol. Each attribute in an atom is set to either a variable or a constant, i.e., value. Variable and constants are also called *terms*. A *ground atom* is an atom that only contains constants. A *literal* is an atom, or the negation of an atom. A *Horn clause* (clause for short) is a finite set of literals that contains exactly one positive literal. Horn clauses are also called Datalog rules (without negation) or conjunctive queries. A *Horn definition* is a set of Horn clauses with the same positive literal, i.e., Datalog program.

Relational learning algorithms learn first-order logic definitions from an input relational database and training examples. Training examples E are tuples of a single target relation, and express positive (E_+) or negative (E_-) examples. The learned definition is called the *hypothesis*, which is usually restricted to Horn definitions. The *hypothesis space* is the set of all possible Horn definitions that the algorithm can explore. Clause C covers an example e if $I \wedge C$ entails e, denoted by $I \wedge C \models e$, i.e., if I and C are true, then e is true. Definition H covers an example e if any of its clauses covers e.

CastorX's learning algorithm is depicted in Algorithm 1. It constructs one clause at a time using the *LearnClause* function. The *LearnClause* function follows a bottom-up approach, which consists of two steps: 1) build the most specific clause in the hypothesis space that covers a given positive example, called a *bottom-clause*, and 2) generalize the bottom-clause to cover as most positive and as fewest negative examples as possible.

Algorithm 1: CastorX's learning algorithm.

```
Input  : Database instance $I$, examples $E$
Output : Horn definition $H$
1 $H = \{\}, U = E_+$
2 while $U$ is not empty do
3     $C = \text{LearnClause}(I, U, E_-)$
4     if $C$ satisfies minimum criterion then
5         $H = H \cup C$
6         $U = U - \{e \in U | H \wedge I \models e\}$
7 return $H$
```

2.2 Matching Dependencies

A *matching dependency (MD)* [3] is a sentence of the form $R_1[A_1] \approx R_2[A_2] \rightarrow R_1[B_1] \rightleftharpoons R_2[B_2]$, where $\approx$ is a similarity operator. The exact implementation of the similarity operator depends on the underlying domains of attributes. For example, given that the domain of attributes is a set of strings, one may use string similarity functions, such as edit distance. $R_1[B_1] \rightleftharpoons R_2[B_2]$ indicate that one can exchange the values of $R_1[B_1]$ and $R_2[B_2]$ in the database. Intuitively, the aforementioned MD says that given the values of $R_1[A_1]$ and $R_2[A_2]$ are sufficiently similar, the values of $R_1[B_1]$ and $R_2[B_2]$ are essentially different representations of the same value.

3 PROBLEM DEFINITION

We extend relational learning to learn over multiple databases. Given a set of databases $I_1, \ldots, I_m$ with schemas $S_1, \ldots, S_m$, a set of MDs, and training examples E for the target relation T, we wish to learn a Datalog definition for T in terms of the relations in schemas $S_1, \ldots, S_m$. The input set of MDs are defined over relations in $S_1, \ldots, S_m$ and express the relationships between these schemas. In this paper, we focus on simple matching dependencies in the form of $R_i[A] \approx R_j[B] \rightarrow R_i[A] \rightleftharpoons R_j[B]$, where R_i and R_j are relations in S_i and S_j, $1 \leq i, j \leq m$, respectively, and A and B are attributes in R_i and R_j, respectively.

4 FINDING RELEVANT INFORMATION

4.1 Bottom-clause Construction

A *bottom-clause* C_e associated with an example e is the most specific clause in the hypothesis space that covers e. Let $I_1, \ldots, I_m$ be the input databases. The bottom-clause construction algorithm consists of two phases. First, CastorX finds all the information in $I_1, \ldots, I_m$ relevant to e. The information relevant to example e are sets of tuples $I_{e1} \subseteq I_1, \ldots, I_{em} \subseteq I_m$ that are connected to e. A tuple t is connected to e if we can reach t using a sequence of (similarity) search operations, starting from e. Next, given the information relevant to e, CastorX creates the bottom-clause C_e.

To find the information relevant to e, CastorX uses the following algorithm. Without loss of generality, we assume that all attributes have the same domain. CastorX maintains a set M that contains all seen constants. Let $e = T(a_1, \ldots, a_n)$ be a training example. First, CastorX adds $a_1, \ldots, a_n$ to M. These constants are values that appear in tuples in $I_1, \ldots I_m$. Then, for each $I_j \in \{I_1, \ldots, I_m\}$, CastorX searches all tuples in I_j that contain at least one constant in M and adds them to I_{ej}. To search across multiple databases, CastorX uses MDs that provide the relationship between databases. If M contains constants in some relation $R_i \in I_i$ and given an MD $R_i[A] \approx R_j[B] \rightarrow R_i[A] \rightleftharpoons R_j[B]$, where $R_j \in I_j$, CastorX performs a similarity search over $R_j[B]$ to find relevant tuples in R_j. For each new tuple in I_{ej}, the algorithm extracts new constants and adds them to M. The algorithm repeats this process for a fixed number of iterations d.

movies(m1,Superbad (2007),2007)	movies2genres(m1,comedy)
movies(m2,Zoolander (2001),2001)	movies2genres(m2,comedy)
movies(m3,Orphanage (2007),2007)	movies2genres(m3,drama)
movies2countries(m1,c1)	countries(c1,USA)
movies2countries(m2,c1)	countries(c2,Spain)
movies2countries(m3,c2)	englishMovies(m1)
movies2releasedate(m1,August,2007)	englishMovies(m2)
movies2releasedate(m2,September,2001)	spanishMovies(m3)

Table 3: Example database.

To create the bottom-clause C_e from $I_{e1}, \dots I_{em}$, CastorX first maps each constant in M to a new variable. It creates the head of the clause by creating a literal for e and replacing the constants in e with their assigned variables. Then, for each tuple $t \in I_{ej}$, $I_{ej} \in \{I_{e1}, \dots, I_{em}\}$, CastorX creates a literal and adds it to the body of the clause, replacing each constant in t with its assigned variable. If there is an input MD $R_i[A] \approx R_j[B] \rightarrow R_i[A] \rightleftharpoons R_j[B]$, where $R_j \in I_j$, and there is a tuple $t' \in R_i$, then I_{ej} may contain a tuple t obtained through similarity search. In this case, we add a similarity literal $sim_{R_i[A],R_j[B]}(v_1, v_2)$, where v_1 and v_2 are the variables assigned to $t'[A]$ and $t[B]$, respectively.

Example 4.1. *Given example highGrossing(Superbad), the database in Table 3, and MD ϕ : $highGrossing[title] \approx movies[title] \rightarrow highGrossing[title] \rightleftharpoons movies[title]$, CastorX finds the relevant tuples movies(m1, Superbad (2007), 2007), movies2genres(m1, comedy), movies2countries(m1, c1), englishMovies(m1), movies2releasedate(m1, August, 2007), and countries(c1, USA). Given these tuples, CastorX creates the following bottom-clause:*

$$highGrossing(x) \leftarrow movies(y,t,z), sim(x,t), movies2countries(y,v), countries(v,\text{`}USA\text{'}), movies2genres(y,\text{`}comedy\text{'}), englishMovies(y), movies2releasedate(y,\text{`}August\text{'},u).$$

4.2 Sampling in Bottom-clause Construction

The tuple sets $I_{e1}, \dots, I_{em}$ created in bottom-clause construction may be large if many tuples in $I_1 \dots, I_m$ are relevant to e. Thus, bottom-clause C_e would be very large, making the learning process prohibitively expensive. To overcome this problem, it is necessary to obtain smaller tuple sets $I^s_{e1} \subseteq I_{e1}, \dots, I^s_{em} \subseteq I_{em}$. Current algorithms [5] do not use any reliable and principled sampling operators, and they simply pick tuples to appear in I^s_{ej} arbitrarily. CastorX implements the following sampling techniques. For ease of explanation, we explain our sampling techniques for the case where we get as input a single database I.

4.2.1 *Random Sampling.* Let the function $sample(I_e)$ return a random sample of I_e. One approach to obtain a smaller tuple set I^s_e is to first compute I_e, and then compute $I^s_e = sample(I_e)$. A more efficient approach is to apply random sampling during bottom-clause construction to directly compute I^s_e. Assume that I contains only two relations $R_1(A, B)$ and $R_2(B, C)$. Let M be the set of seen constants. Let I^i_e be the set of tuples added to I^s_e in iteration i. In iteration 1, we search for constants in M in R_1 and R_2, i.e., $I^1_{R_1} = \sigma_{A,B \in M}(R_1)$ and $I^1_{R_2} = \sigma_{B,C \in M}(R_2)$, to obtain $I^1_e = I^1_{R_1} \cup I^1_{R_2}$. Because we are performing a union operation and $I^1_{R_1}$ and $I^1_{R_2}$ are disjoint, we can obtain a sample of I^1_e by sampling $I^1_{R_1}$ and $I^1_{R_2}$ separately and then performing the union between these samples [4], i.e., $sample(I^1_e) = sample(I^1_{R_1}) \cup sample(I^1_{R_2})$. In iteration i for $2 \leq i \leq d$, M contains constants found in previous iterations. Let a and a' be two constants in $I^{i-1}_{R_1}$, which were added to M. Assume that we are looking for these constants in R_2 to obtain $I^i_{R_2}$. If a appears more frequently than a' in $I^{i-1}_{R_1}$, then when computing $sample(I^i_{R_2})$, we should be more likely to pick tuples that contain constant a compared to a'. This is because we wish to obtain a random sample of I_e.

This is similar to the idea of doing random sampling over joins. Olken [4] proposed an algorithm to perform sampling over binary joins. To compute $sample(R_1 \bowtie R_2)$, Olken's algorithm first obtains a uniform sample $R^s_1 \subseteq R_1$. Then, it performs the join $J = R^s_1 \bowtie R_2$. For each tuple $t \in J$, the algorithm accepts t to be in $sample(R_1 \bowtie R_2)$ with a probability based on the frequency of the value in the join attribute in R_2, and rejects it otherwise. We adopt this technique to perform random sampling during bottom-clause construction. In iteration 1, we obtain a sample of I^1_e as above, where we obtain a random sample of each relation, and then perform the union between these samples. In iteration i for $2 \leq i \leq d$, we obtain the sample for each $I^i_{R_j}$ by using Olken's acceptance/rejection technique, and then perform the union between these samples.

A definition for a target relation may require a relation that is not necessarily in the random sample I^s_e. Therefore, we may not be able to learn the correct definition.

Example 4.2. *Assume that we will learn a definition that requires the relation spanishMovies in Table 3. We build a bottom-clause using random sampling, and set sample size $s = 1$. Assume that M contains known constants 'c1' and 'c2' and we will obtain a random tuple from relation movies2countries. Because constant 'c1' appears more frequently than 'c2', it is more likely that we obtain movies 'm1' or 'm2', compared to 'm3'. Therefore, our random sample will not contain movie 'm3', hence will not access relation spanishMovies.*

4.2.2 *Stratified Sampling.* We introduce the notion of stratified sampling to bottom-clause construction. Let T be the target relation for example e. A join path is any path that can occur in $T \bowtie (\cup_{R \in I} R)_1 \bowtie \dots \bowtie (\cup_{R \in I})_d$. A stratified sample I^s_e of I_e must contain at least one occurrence of every possible join path in I_e. We first compute all possible join paths. This can be done by only looking at the schema. Then, for each join path, we obtain one or more random samples. Finally, we perform the union of the samples for each join path to obtain I^s_e. Stratified sampling guarantees that the relational learning algorithm will be able to access all relations, hence explore a wide variety of definitions.

5 GENERALIZATION

After creating the bottom-clause C_e for example e, CastorX generalizes C_e iteratively. First, CastorX randomly picks a subset $E^s_+ \subseteq E_+$ of positive examples. For each example e' in E^s_+, CastorX generates a candidate clause C', which is more general than C_e and covers e'. To do so, it simply drops literals in the body of C_e that do not cover e'. CastorX then selects the highest scoring candidate clauses and iterates until the clauses cannot be improved. There are multiple functions to compute the score of a candidate clause, all of which quantify the coverage of positive and negative examples [2]. CastorX calculates the score of each candidate clause by subtracting the number of negative examples from the number of positive examples covered by the clause.

Example 5.1. *Consider the bottom-clause C_e in Example 4.1 and positive example $e' = highGrossing(Zoolander)$. To generalize C_e to cover e', CastorX drops the literal $movies2releasedate(y, \text{'}August\text{'}, u)$ because the movie Zoolander was not released in August, according to the database in Table 3.*

5.1 Approximate Clause Evaluation

To select the highest scoring candidate clauses, CastorX computes the number of positive and negative examples covered by the clauses. These tests dominate the time for learning. One approach to evaluate a clause is to transform the clause into a SQL query and evaluate it over the input database I. However, the SQL query will involve long joins, making the evaluation prohibitively expensive on large clauses. Instead, CastorX uses an approach called θ-subsumption. Clause C *θ-subsumes* C', denoted by $C \subseteq_\theta C'$, iff there is some substitution θ such that $C\theta \subseteq C'$. First, a ground bottom-clause C_e^g is created for each example $e \in E$. A ground-bottom clause is a bottom-clause that only contains ground atoms. Then, $I \wedge C \models e$ if $C \subseteq_\theta C_e^g$.

To reduce the time of evaluation, CastorX also builds C_e^g from a subset $I_e^s \subseteq I_e$, as done in Section 4.2. Because I_e^s is a sample of I_e, checking whether $C \subseteq_\theta C_e^g$ is an approximation of checking whether $I \wedge C \models e$, meaning that there may be errors. A good sampling technique to obtain I_e^s reduces errors. The learning algorithm involves many (thousands) coverage tests. Because CastorX reuses ground bottom-clauses, it can run efficiently over large databases.

5.2 Applying Chase

After finding the clauses with the highest scores, the clauses learned by CastorX may contain similarity literals. In this case, we apply the Chase to the learned definition to obtain a definition that can be interpreted and holds over the integrated database, i.e., as if the definition has been learned over the integrated and clean database [1]. Generally speaking, for each similarity literal in the clause, the Chase algorithm applies an MD whose left-hand side matches the attributes used in the similarity literal and generates a new clause. The algorithm iteratively applies MDs to the created clause until no similarity literal is left in the clause. This algorithm is guaranteed to terminate after finitely many steps [1].

Example 5.2. *Assume that CastorX learns the following clause for the target relation highGrossing over IMDb and BOM databases.*

$$highGrossing(x) \leftarrow movies(y, t, z), movies2genres(y, \text{'}comedy\text{'}), highBudgetMovies(x), sim(x, t).$$

Given MD $highGrossing[title] \approx movies[title] \rightarrow highGrossing[title] \rightleftharpoons movies[title]$, CastorX unifies variables t and x and generates the following clause:

$$highGrossing(x) \leftarrow movies(y, x, z), movies2genres(y, \text{'}comedy\text{'}), highBudgetMovies(x).$$

Since there is no MD left to apply, Chase terminates.

The relationships between different databases can be expressed using more complex MDs. For instance, one may want to assert that two tuples in *IMDb* and *BOM* databases represent information about the same movies if their titles are similar and the movies share the same set of genres. Supporting more complex MDs raises multiple challenges. For instance, the order in which MDs are applied may affect the output of Chase. Further, it is *not* clear how to correctly evaluate the resulting clause over the underlying databases.

Database	Sampling in bottom-clause construction	Sampling in clause evaluation	Precision	Recall	Time (min)
HIV	Naïve (k=10)	Naïve (k=10)	0.55	0.93	3.08
		Naïve (k=20)	0.77	0.86	5.05
		No sampling	0.84	0.87	27.99
	Random (k=10)	Random (k=10)	0.55	0.90	13.25
		Random (k=20)	0.75	0.83	28.84
		No sampling	0.79	0.81	12.57
	Stratif. (k=10)	Stratif. (k=10)	0.54	0.95	6.15
		Stratif. (k=20)	0.83	0.89	9.92
		No sampling	0.84	0.90	24.97
IMDb +BOM	Naïve (k=10)		0.86	0.78	59.9
	Stratified (k=10)		0.95	0.78	95

Table 4: Results of learning over the HIV and IMDb+BOM databases. Sample size is denoted by k. 'No sampling' indicates that the full ground bottom-clause was used for clause evaluation.

6 EXPERIMENTS

We use the HIV database, which contains information about chemical compounds. We learn the target relation *antiHIV(comp)*, which indicates that *comp* has anti-HIV activity. The database contains 7.8M tuples, 2K positive, and 4K negative examples. Table 4 (top) shows the results for learning over the HIV database with different sampling techniques using 10-fold cross validation. We refer to the method of arbitrarily picking samples as *naïve sampling*.

We use the JMDB database (*jmdb.de*), which contains information from IMDb, and the Box Office Mojo (BOM) database, to learn a definition for the target relation *highGrossing(title)*. The JMDB and BOM databases contain 9M and 100K tuples, respectively. We use the top 1K grossing movies in BOM as positive examples, and the lowest 2K grossing movies in BOM as negative examples. Because training data is created from the BOM database, we create the MD $highGrossing[title] \approx JMDB.movies[title] \rightarrow highGrossing[title] \rightleftharpoons JMDB.movies[title]$ to allow CastorX to access information in JMDB, where $\approx$ represents a maximum edit distance of 10. Table 4 (bottom) shows the results using 5-fold cross validation. In both experiments, stratified sampling delivers the best trade-off between precision, recall, and running time.

7 CONCLUSION AND FUTURE WORK

We proposed CastorX, a relational learning system that enables users to leverage information in multiple databases to learn the definition of a target relation. We will extend CastorX to allow the user to establish the relationships between multiple databases using relatively more complex matching dependencies.

REFERENCES

[1] L. E. Bertossi, S. Kolahi, and L. V. S. Lakshmanan. Data cleaning and query answering with matching dependencies and matching functions. *Theory of Computing Systems*, 2011.

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

[3] W. Fan, X. Jia, J. Li, and S. Ma. Reasoning about record matching rules. *PVLDB*, 2009.

[4] F. Olken. *Random Sampling from Databases*. PhD thesis, UC Berkeley, 1993.

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