Learning Efficiently Over Heterogeneous Databases

Jose Picado Arash Termehchy Sudhansu Pathak
Oregon State University
{picadolj,termehca,pathaks}@oregonstate.edu

ABSTRACT

Given a relational database and training examples for a target relation, relational learning algorithms learn a Datalog program that defines the target relation in terms of the existing relations in the database. We demonstrate CastorX, a relational learning system that performs relational learning over heterogeneous databases. The user specifies matching attributes between (heterogeneous) databases through matching dependencies. Because the content in these attributes may not match exactly, CastorX uses similarity operators to find matching values in these attributes. As the learning process may become expensive, CastorX implements sampling techniques that allow it to learn efficiently and output accurate definitions.

PVLDB Reference Format:
Jose Picado, Arash Termehchy, and Sudhanshu Pathak. Learning Efficiently Over Heterogeneous Databases. *PVLDB*, 11 (12): 2066 - 2069, 2018.
DOI: https://doi.org/10.14778/3229863.3236261

1. INTRODUCTION

Users often would like to use machine learning methods to discover interesting and novel relations over relational databases [2]. For instance, consider the IMDb database (*imdb.com*), which contains information about movies, for which 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. Given a relational database and training examples for a new target relation, *relational machine learning* (relational learning) algorithms attempt to learn (approximate) relational definitions of the target relation in terms of existing relations in the database [2]. Definitions are usually restricted to Datalog programs. 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:

$$highGrossing(x) \leftarrow movies(y, x, z), mov2genres(y, `comedy'), \\ mov2releasedates(y, `June', u)$$

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, analytics, and machine learning applications, such as query learning and information extraction [2].

Table 1: Schema fragments for the IMDb database.

movies(id,title,year)	mov2countries(id,name)
mov2genres(id,name)	mov2releasedates(id,month,year)

Table 2: Schema fragments for Box Office Mojo.

mov2totalGross(title,gross)
highBudgetMovies(title)

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. However, this information is available in another database called Box Office Mojo (BOM) (*boxofficemojo.com*), for which schema fragments are shown in Table 2. Using this information may help the user learn a more accurate definition for the *highGrossing* relation, as high grossing movies may have high budgets. Currently, users have to first (manually) integrate the 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 representations and there is no simple rule to match titles of the same movie in these two databases. Further, integrating two or more databases may result in numerous integrated instances [1]. This is because when unifying the values that refer to the same real-world entity, we may have multiple choices. It is not clear which is the correct integrated instance to use in order to learn an accurate definition.

More importantly, the user does *not* always need to integrate databases to learn a definition for a target relation [4]. 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 does *not* need to use the in-

Proceedings of the VLDB Endowment, Vol. 11, No. 12
Copyright 2018 VLDB Endowment 2150-8097/18/8.
DOI: https://doi.org/10.14778/3229863.3236261

formation in the BOM database and can learn an effective definition by using only the data in the IMDb database.

We demonstrate CastorX, a relational learning system that learns 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 can meaningfully join but their values may not match exactly. For example, there is an MD between the *title* attribute in relation *mov2totalGross* in BOM and the *title* attribute in relation *movies* in IMDb. These constraints help CastorX find the join paths across multiple databases. Then, CastorX leverages these constraints and efficiently performs exact and similarity joins within the same database and across databases to learn an effective definition for the target relation. It is time-consuming to perform similarity joins between two large relations as a tuple from one relation usually matches far too many tuples from the other one [8]. CastorX uses efficient (stratified) 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. To our knowledge, there are no other systems that perform integration and relational learning simultaneously.

Our demonstration will allow the VLDB audience to perform relational learning over heterogeneous databases using CastorX. We will guide the user in the process of providing matching dependencies between attributes in different databases, creating training examples, and running CastorX to learn a definition for the target relation.

2. BACKGROUND

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. A *ground clause* is a clause that only contains ground atoms. 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 or union of conjunctive queries.

3. SYSTEM OVERVIEW

3.1 Relational Learning

Relational learning algorithms learn first-order logic definitions from an input relational database and training examples. Training examples E are usually 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 $I \wedge C \models e$, where $\models$ is the entailment operator, i.e., if I and C are true, then e is true. Definition H covers an example e if any of the clauses in H covers e.

Algorithm 1: Castor's cover-set algorithm.

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

Table 3: Example database.

movies(m1,Superbad,2007)	movies(m2,Zoolander,2001)
mov2countries(m1,USA)	mov2countries(m2,USA)
mov2genres(m1,comedy)	mov2genres(m2,comedy)
mov2releasedates(m1,August,2007)	
mov2releasedates(m2,September,2001)	

CastorX follows a covering approach, depicted in Algorithm 1. It constructs one clause at a time using the *LearnClause* function. If the clause satisfies the minimum criterion, CastorX adds 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. CastorX's *LearnClause* function follows a bottom-up method. It has two steps. 1) Build the most specific clause in the hypothesis space that covers a given positive example, called a *bottom-clause.* 2) Generalize the bottom-clause to cover as most positive and as fewest negative examples as possible.

3.2 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 be the input database instance. To compute the bottom-clause associated with an example e, CastorX maintains a hash table M that contains all known constants and a tuple set I_e that contains tuples related to e. First, CastorX adds the constants in e to M. Then, the algorithm selects all tuples in I that contain at least one constant in M and adds them to I_e. For each new tuple, it extracts new constants and adds them to M. The algorithm repeats this process for a fixed number of iterations d. After d iterations, CastorX creates the bottom-clause. First, CastorX maps each constant in M to a new variable. Then, it creates the head of the clause by creating a literal for e and replacing the constants in e with their assigned variables according to M. Then, for each tuple $t \in I_e$, CastorX creates a literal and adds it to the body of the clause, replacing each constant in t with its assigned variable according to M.

EXAMPLE 3.1. *Given example highGrossing(Superbad), using the database in Table 3, CastorX creates the following bottom-clause:*

$$highGrossing(x) \leftarrow movies(y, x, z), mov2genres(y, \text{`}comedy\text{'}), mov2countries(y, \text{`}USA\text{'}), mov2releasedates(y, \text{`}August\text{'}, u).$$

3.2.1 Learning over Heterogeneous Databases

We extend the bottom-clause construction algorithm to perform similarity joins over heterogeneous databases. It would be expensive to find all potential similarity joins.

Therefore, the user must specify, through matching dependencies, which attributes can be joined using similarity join. Let the domain of attribute A be denoted as $dom(A)$. Two attributes A and B are comparable if $dom(A) = dom(B)$. Given two pairs of pairwise comparable attribute lists X_1, X_2 and Y_1, Y_2 from relations R_1 and R_2, respectively, a *matching dependency (MD)* [3] is a sentence of the form

$$R_1[X_1] \approx_j R_2[X_2] \rightarrow R_1[Y_1] \rightleftharpoons R_2[Y_2],$$

where $\approx_j$ is a similarity operator. CastorX uses a restricted set of matching dependencies that make the following assumptions: 1) attribute lists X_1, X_2, Y_1 and, Y_2 consist of a single attribute, 2) $X_1 = Y_1$ and $X_2 = Y_2$, and 3) $R_1[Y_1] \rightleftharpoons R_2[Y_2]$ means that $R_1[Y_1]$ and $R_2[Y_2]$ can be joined using similarity operator $\approx_j$. For instance, given the schemas in Tables 1 and 2, the MD

$$movies[title] \approx_j mov2totalGross[title] \rightarrow$$
$$movies[title] \rightleftharpoons mov2totalGross[title],$$

indicates that if the values in attributes *movies[title]* and *mov2totalGross[title]* are similar, then they can be joined using similarity operator $\approx_j$.

Assume there is a matching dependency $R_1[A] \approx_j R_2[A] \rightarrow R[A] \rightleftharpoons R_2[A]$, and that the tuple set I_e created in bottom-clause construction contains a tuple t_1 for R_1 with $t_1[A] = a$. Instead of searching R_2 for all tuples that contain constant a in attribute $R_2[A]$, we perform a similarity search, according to the similarity operator $\approx_j$. CastorX uses efficient similarity search operators [8]. The similarity search operation outputs a set W of similar values to a in $R_2[A]$. We create a literal for each tuple in R_2 containing a value $b \in W$ and add it to the bottom-clause. Because $t_1[A] = a$ and $t_2[A] = b$ are similar but different values, they are assigned different variables. Therefore, to indicate that t_1 and t_2 can join through a similarity operator, we also add the similarity literal $sim_j(x, y)$ to the bottom-clause, where x and y are the variables assigned to values a and b, respectively. The bottom-clause that contains the literal $sim_j(x, y)$ is a *compact representation* of two bottom-clauses: one where all occurrences of variable x are replaced by y, and another one where all occurrences of variable y are replaced by x.

3.2.2 Sampling in Bottom-clause Construction

Rationale for sampling: The tuple set I_e created in bottom-clause construction may be large if e is related to many tuples in I. Thus, bottom-clause C_e would be very large, making the learning process time-consuming. This problem is exacerbated when using similarity operators to find relevant tuples, as many entities have similar names. Therefore, CastorX samples from the tuples in I_e to obtain a subset $I_e^s \in I_e$, and creates bottom-clauses from the tuples in I_e^s. A bottom-clause containing similarity literals that is created by sampling the heterogeneous databases is actually a compact representation of all bottom-clauses created by sampling over all integrated instances. Therefore, we can safely sample and create a bottom-clause over heterogeneous data. Current algorithms [7] do not use any reliable sampling operator, and they simply pick tuples arbitrarily. CastorX implements the following sampling techniques.

Random sampling: We sample I_e such that the sample tuple set I_e^s is a random sample of I_e. Olken [6] proposed techniques for doing random sampling over relational databases. We integrate these techniques into the bottom-clause construction algorithm.

Stratified sampling: We introduce the notion of stratified sampling [5] to bottom-clause construction. The sample tuple set I_e^s must contain at least one occurrence of every possible join path in I_e This condition guarantees that the relational learning algorithm will be able to explore a wide variety of definitions. This technique allows CastorX to be efficient and output accurate definitions.

3.3 Generalization

After creating the bottom-clause C_e for example e, CastorX generalizes C_e iteratively. CastorX randomly picks a subset $E_+^s \subseteq E_+$ of positive examples to generalize C_e. 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.

EXAMPLE 3.2. *Consider the bottom-clause C_e in Example 3.1 and positive example e' = highGrossing(Zoolander). To generalize C_e to cover e', CastorX drops the literal $mov2releasedates(y, \text{‘}August\text{’}, u)$ because the movie Zoolander was not released in August.*

3.3.1 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' iff there is some substitution θ such that $C\theta \subseteq C'$. $C\theta \subseteq C'$ means that the result of applying substitution θ to clause C is a subset of clause C'. To evaluate whether a clause C covers an example e, we first build a ground bottom-clause C'_e for e, and then check whether C *θ-subsumes* C'_e.

To improve the efficiency of evaluation, we can approximately evaluate clauses by using the sampling techniques introduced in Section 3.2.2 to build ground bottom-clauses. The learning algorithm involves many (thousands) coverage tests. Because CastorX reuses ground bottom-clauses, it can run efficiently, even over large databases.

3.3.2 Applying Chase

If the definition learned by CastorX contains similarity literals after generalization, 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 a 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 [1].

EXAMPLE 3.3. *Assume that CastorX learns the following clause for the target relation highGrossing over IMDb and BOM databases.*

$$highGrossing(x) \leftarrow movies(y, t, z), mov2genres(y, \text{‘}comedy\text{’}),$$
$$highBudgetMovies(x), sim(x, t).$$

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.

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 + Box Office Mojo				
Naïve (k=10)		0.86	0.78	59.9
Stratified (k=10)		0.95	0.78	95

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

$$highGrossing(x) \leftarrow movies(y, x, z), mov2genres(y, \text{'}comedy\text{'}),$$
$$highBudgetMovies(x).$$

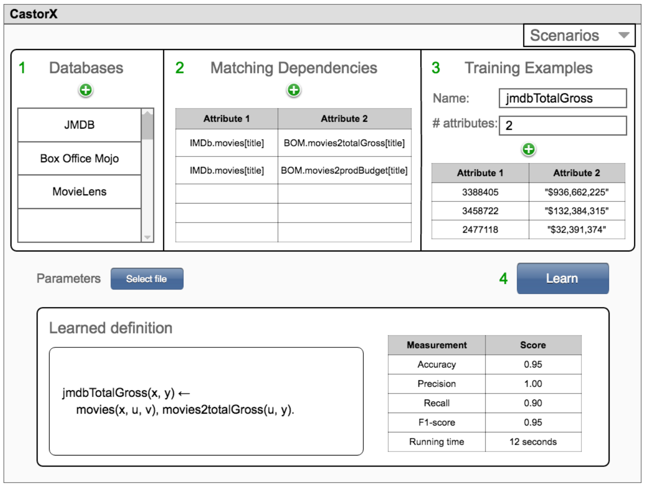
3.4 Experiments

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 3.4 (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 3.4 (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.

4. DEMONSTRATION PROPOSAL

We will demonstrate CastorX by guiding the audience through the process of learning over heterogeneous databases. Figure 1 shows CastorX's user interface. The process consists of the following steps: 1) specify the databases to learn from, 2) specify matching attributes between (heterogeneous) databases through matching dependencies, 3) create or specify training examples for a target relation, and 4) run the learning algorithm. In order to create an interactive demonstration, we will use subsets of the movie databases.



JMDB
Box Office Mojo
MovieLens

Attribute 1	Attribute 2
IMDb.movies[title]	BOM.movies2totalGross[title]
IMDb.movies[title]	BOM.movies2prodBudget[title]

Name: jmdbTotalGross

attributes: 2

Attribute 1	Attribute 2
3388405	"$936,662,225"
3458722	"$132,384,315"
2477118	"$32,391,374"

jmdbTotalGross(x, y) ←
movies(x, u, v), movies2totalGross(u, y).

Measurement	Score
Accuracy	0.95
Precision	1.00
Recall	0.90
F1-score	0.95
Running time	12 seconds

Figure 1: CastorX's user interface.

Create MDs: CastorX shows all relations in the available databases, e.g., JMDB and BOM. To create the MD, the user browses through the relations and selects the two attributes that will appear in the MD, and the maximum edit distance between values in these attributes.

Run learning algorithm: The audience will be able to run the learning algorithm with the provided input, and examine CastorX's output. CastorX will show the learned definition and accuracy measurements.

Predefined scenarios: Because this process may take more than a few minutes, we will have predefined scenarios. A predefined scenario consists of a set of databases, a set of MDs over the databases, and a set of training examples for some target relation. We will show scenarios for performing novel concept discovery, e.g., learning the *highGrossing(movie)* relation, as well as query learning, e.g., learning the *jmdbTotalGross(jmdbId, totalGross)* relation indicating that movie with id *jmdbId* has a total grossing *totalGross*. The user will be able to edit these scenarios, e.g., remove an MD, and see how this change affects CastorX's output.

5. 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*, 52:441–482, 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*, 2(1):407–418, 2009.

[4] A. Kumar, J. F. Naughton, J. M. Patel, and X. Zhu. To join or not to join?: Thinking twice about joins before feature selection. In *SIGMOD*, 2016.

[5] B. Mozafari and N. Niu. A handbook for building an approximate query engine. *IEEE Data Eng. Bull.*, 38:3–29, 2015.

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

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

[8] J. Wang, G. Li, D. Deng, Y. Zhang, and J. Feng. Two birds with one stone: An efficient hierarchical framework for top-k and threshold-based string similarity search. *ICDE*, pages 519–530, 2015.