Automatic Generation of Relational Attributes: An Application to Product Returns

Michele Samorani
Leavey School of Business
Santa Clara University, USA
Email: msamorani@scu.edu

Farrukh Ahmed
Department of Computing Science
University of Alberta, Canada
Email: farrukh1@ualberta.ca

Osmar R. Zaïane
Department of Computing Science
University of Alberta, Canada
Email: zaiane@ualberta.ca

***Abstract*—Although statistical and machine learning methods require the input data to be in a tabular format, in real-world applications data are often stored across several tables in a relational database. How to build a single mining table from a relational database is a critical pre-processing step of any classification method, because including the right attributes may dramatically boost the accuracy of the classifier. We propose a methodology and implement a software program, Dataconda, to automatically mine a relational database. The user selects a class attribute contained in a table of the database and the procedure builds and selects predictors by exploring the whole database and aggregating information, without any user intervention. For example, our procedure may find that the best predictor for "product return" is the proportion of products returned by the same customer in the past, even if the user has not built any such attribute. Our procedure produces more expressive attributes than existing methods. Our experiments on the ISMS Durable Goods Datasets, a publicly available data set of product returns in retailing, suggest that our method allows new knowledge to emerge.**

***Keywords*-Feature construction; Knowledge discovery; Software tools**

I. Introduction

A. Motivation

The machine learning community traditionally assumes the input of classification or supervised attribute selection to be a flat mining table, in which each row represents an object and each column an attribute; one more attribute is the class, which indicates which group the object belongs to. By contrast, in a real-world environment, the flat mining table is never given, but it rather needs to be constructed manually from a variety of data sources, typically the tables of a relational database.

Consider, for example, the *Product Returns* database in Figure 1, which contains the information on the products purchased by the clients of a store. Each purchase is made by exactly one client and involves exactly one product; however, each client and each product may appear in any number of purchases. Suppose that the class is the binary attribute *Purchase.Return*, which indicates whether the purchased product was later returned to the store. For simplicity, let us assume that the value of *Purchase.Return* is known right after the purchase is made.

When searching for a product, customers may purchase a product that does not meet their requirements or expectations. When that happens, if the retailer's policy allows it, customers may decide to return the product for cash back or for store credit. The proportion of returned purchases, called "return rate", may be as high as 25% [1]. Overall, the monetary value of product returns in the US, inclusive of costs for lost sales and reverse logistics, is approximately \$100 billion [2]. For this reason, retailers are interested in understanding the causes of product returns.

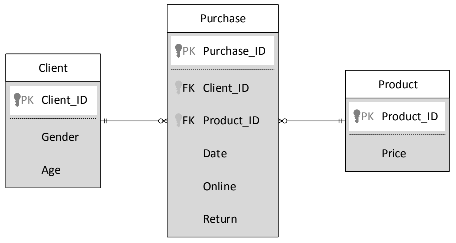

Figure 1. The *Product Returns* database.

To find the drivers of product returns, one could use any classification or attribute selection procedure. However, before doing so, it is necessary to build a flat mining table with one row per purchase and one attribute for each possible cause of *Return*. In other words, the table *Purchase* needs to be "enlarged" by adding handcrafted attributes that summarize information from other tables. Typically, what attributes to build is driven by the intuition of analysts, who use their domain knowledge to formulate hypotheses about the phenomenon analyzed. In this simple example, the analysts may decide to add the following attributes to the table *Purchase*:

1) Online – perhaps, online purchases are more likely to be returned?
2) Gender of the client – perhaps, a gender is more likely to return products than the other?
3) Age of the client – perhaps, younger clients are more inclined to return products?
4) Price of the product – perhaps, more expensive products are more likely to be returned?
5) Proportion of returned purchases made by the same client prior to the current purchase – perhaps, clients who have returned products in the past are more

inclined to return products in the future?

6) Number of times that the the same product was returned prior to the current purchase – perhaps, products that are returned often will be more likely returned in the future?
7) Total amount of money spent in the past by the client – perhaps, very active clients behave differently from less active clients?

The process of manually building these attributes has two disadvantages. Firstly, as illustrated by the example above, analysts tend to build only those attributes that they suspect being good predictors of the class. Thus, a manual analysis is unlikely to find unexpected knowledge. Secondly, this manual process is very time consuming and error prone, because it consists of writing a large amount of SQL code.

B. *Goals of this paper*

The first goal of this paper is to propose a new methodology and implement a software framework, Dataconda, freely available at www.dataconda.net, to find the predictors for a class of interest without the need for the user to hypothesize them. The user selects a class attribute contained in a table of the database, and our procedure builds and selects predictors by exploring the whole database and aggregating information, without any user intervention. While some of the predictors are generated by simply joining the target table with the other tables (e.g., attributes 1-4), many predictors are generated by joining in a non-trivial way many tables or the same table more than once (e.g., attributes 5-7). The main difference between Dataconda and other data mining software is that the user does not need to make any hypothesis on the predictors. The user can extend Dataconda by introducing custom aggregating functions and custom attribute selection procedures.

The second goal of the paper is to employ our framework to identify the drivers of product returns in simulated and real data. Our results on the data of a large retailer show that our methodology discovers good predictors of product returns that were unknown, thereby enriching our understanding of the consumer behavior.

II. Related Work

The problem of classifying objects in a relational database has been extensively studied within the multi-relational data mining community. The two main approaches are Inductive Logic Programming (ILP) and Propositionalization.

ILP, proposed by [3], starts from the basic facts present in the tables (e.g., *Client(Joe, M, 33)*, *Client(Claire, F, 28)*) and uses induction engines, such as Prolog [4], to derive rules behind returning a purchase. These approaches suffer from several problems. First, aggregation is generally implemented only through a binary existence qualifier, which indicates whether the current record is associated to at least one record in another table that satisfies a certain condition. For example, if the target table is *Client*, a new binary attribute would be only of the form *"there exists a purchase with online = 1"*. Second, they do not allow nested aggregations, such as $MAX(COUNT(...))$. Third, the attribute generation and the classification happen at the same time: generated attributes are successively evaluated and used to recursively split the data set, with an approach similar to the greedy construction of a decision tree. For this reason, most of the existing statistical and machine learning algorithms, which work on a single table, cannot be applied under an ILP framework.

By contrast, Propositionalization approaches decouple the attribute generation phase from the classification phase. In practice, these approaches simply add attributes to the target table (*Purchase*) by collecting information from the other tables. Previous Propositionalization approaches are proposed by [5], [6], [7]. The approach [5] uses filters only in the presence of the aggregation operator *COUNT*, in the same way as a *COUNTIF* function. For example, while it would generate the attribute *"number of items returned (Count(*) where return = 1)"*, it would not generate the attribute *"average price of the items returned (Avg(price) where return= 1)"*. The Propositionalization engine ACORA [6] does not have these limitations, but it forbids paths that use the same id twice in a row. For example, it cannot generate any attribute concerning the previous purchases of a client, as the key *client_id* would be used twice in a row on the path *Purchase* $\rightarrow$ *Client* $\rightarrow$ *Purchase*. We call paths containing sub-paths $A \rightarrow B \rightarrow A$ "rolled" paths. The reason for not allowing rolled paths is that

> *"Tables resulting from a path that reuses the same key only result in a replication of information that is available on a shorter path that skips that table" ([6], page 100)*

However, this claim is true only if the database is navigated through joins made using foreign keys. For example, the claim above is valid for the following join:

```
Select * From Purchase p1
Inner join Client h
On p1.Client_id = h.Client_id
Inner join Purchase p2
On h.Client_id = p2.Client_id
```

because the output would be composed of all pairs (purchase, client) repeated n times, where n is the number of purchases made by a particular client. Thus, it may appear useless to use the same key twice in a row, as it may just result in replicating the same information multiple times.

However, navigating through the same table more than once allows to find past information. Unlike ACORA, our database navigation perform joins successively for each pairs of tables (A, B), every time making sure that the number of rows obtained is the same as in table A. For example, for the path *Purchase* $\rightarrow$ *Client* $\rightarrow$ *Purchase*, our method first considers the arc *Client* $\rightarrow$ *Purchase* and adds a new attribute to *Client* (for example, the average return of the past purchases), and then considers the arc *Purchase* $\rightarrow$ *Client* and attaches the new attribute to *Purchase*. The result is the addition of one attribute to *Purchase*.

The method proposed by [7] allows rolled paths, but, since it does not consider date attributes, it generates attributes that may potentially "spoil" (or "leak" see [8]) the class information. We say that an attribute is a *class spoiler* if it contains information that is known only after the class is revealed; clearly, Propositionalization approaches should not generate class spoilers because they would reveal the value of the class attribute before the learning phase, which would lead to learning incorrect rules.

In the product returns database, the only class spoiler is the target attribute *Purchase.Return* because the class should never be used to predict itself. Sometimes, however, class spoilers are attributes that are functionally dependent on the class. For example, if the table *Purchase* had an attribute *reasonForReturn*, which indicates the reason for a return (defective product, didn't like the product, didn't return the product, etc), this attribute would be a class spoiler because it would be built once the value of the class attribute is known. Including such attribute in the target table would be incorrect. Although class spoilers should be excluded from the target table, they can be used to aggregate information from past data. For example, while we want to exclude the attribute $Return$ from the target table, we can include the average value of $Return$ among the client's past purchases.

The navigation process without date attributes surely generates class spoilers because it explores rolled paths without removing future information. For example, along the path *Purchase* $\rightarrow$ *Client* $\rightarrow$ *Purchase*, it generates the number of purchased items returned by the client throughout *her or his entire lifetime*, that is, possibly after the current purchase. Clearly, this attribute is a class spoiler and it should not be generated. Hence, our procedure explores rolled paths while at the same time prevents the generation of class spoilers. A short description of Dataconda was provided in the demos [9], [10] and this paper describes the complete system with algorithms and experimental results.

III. Methodology

The input of our procedure is composed of 1) the database and 2) its metadata, that is, the information about the tables, their attributes, and the associations between tables. One table is the *target table*, i.e., the table to which we want to attach new attributes. One attribute of the target table is the *class*. In our example, the target table is *Purchase*. We next describe the metadata in greater detail.

A. Metadata

Each table is characterized by a name and by a set of attributes. Each attribute is characterized by:

- **name**: the name of the attribute. It must be unique within the table.
- **type** $\in \{ID, date, numerical, categorical\}$: the type determines which aggregation (AVG, MODE, etc) and refinement operators (i.e., "where" clauses) can be executed on the attribute. IDs are primary keys or foreign keys, they are only used to identify records in tables, and have no informational content. The other attributes are used to generate aggregated information.
- **dimension**: the unit of measurement of the attribute. This information is used to determined which attributes can be compared in a refinement. This characteristic is present in some works on attribute generation [7].
- **class spoiler** $\in \{0, 1\}$: this binary feature indicates whether the attribute was built after knowing the class of the instance.

In our procedure, associations can be of two types:

- ***1:1***: $A \rightarrow B$ is a *1:1* association if each record in A is associated with exactly one record in B. For example, $Purchase \rightarrow Client$ is *1:1* because for each purchase there is exactly one client;
- ***0:n*** $A \rightarrow B$ is a *0:n* association if each record in A is associated to a set of records in B. For example, $Client \rightarrow Purchase$ is *0:n* because each client may have done an arbitrary number of purchases.

The type of association determines which operators can be used to aggregate information.

B. Generation of new attributes

The attribute generation procedure implemented in Dataconda is summarized in Algorithm 1. The generation of new attributes is performed similarly to other works on Propositionalization [5], [7]: first, paths are generated from the target table; then, for each path, information is "rolled-up" from the end of the path back to the target table in many different ways, obtaining different attributes.

The first phase (step 2) consists of generating paths that start from the target table and end in any table. Note that the same table may be traversed more than once; however, for reasons that will be clear later, the method forbids the generation of subpaths $A \rightarrow B \rightarrow A$ where $A \rightarrow B$ is a *0:n* association. Despite this restriction, it may still be possible to generate an infinite number of paths. For example, in the Product Returns database (Figure 1), we can do so by traversing the database back and forth an infinite number of times: *Purchase* $\rightarrow$ *Client* $\rightarrow$ *Purchase* $\rightarrow$ *Product* $\rightarrow$ *Purchase*$\rightarrow$ *Client*...

The second step, also called Roll-up [5] (steps 3–22), starts from the last table of the path found in step 1, and iteratively adds a virtual attribute to the previous table, until the target table is finally reached. Let the path found in step 1 be $P = T_0 \rightarrow T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_{l-1}$, where T_0 is the target table and l is the depth of the path. Since the tables

Algorithm 1 Attribute Generation Procedure in Dataconda

```
1: procedure ATTRIBUTE GENERATION(Target table $T_0$, max path depth $L$)
2:     Let $\Pi$ be the set of paths starting from $T_0$ of depth up to $L$        ▷ Path Generation
3:     for each path $P = (T_0, ..., T_{l-1}) \in \Pi$ with $l \leq L$ do        ▷ Roll-up
4:         for $i = l - 2$ down to 0 do
5:             Let $a_1$, $a_2$,...,$a_h$ be the attributes of $T_{i+1}$
6:             Let $v_{i+1}$ be the virtual attribute of $T_{i+1}$
7:             if $T_i \rightarrow T_{i+1}$ is *1:1* then
8:                 Choose any non-id attribute $a_j$ and attach it (i.e., set $v_i = a_j$); or, if present, attach $v_{i+1}$ (i.e., $v_i = v_{i+1}$)
9:             else
10:                Choose any attribute pair $(a_j, a_k)$
11:                Choose an aggregating operator *Agg* compatible with $a_j$
12:                Optionally, choose a refinement operator *Ref* and a comparison quantity $c$ compatible with $a_k$
13:                If $v_{i+1}$ is present in $T_{i+1}$, make sure $a_j = v_{i+1}$ or $a_k = v_{i+1}$
14:                If both $T_0$ and $T_{i+1}$ have a *date* attribute $d$, we will add the refinement $T_{i+1}.d < T_0.d$
15:                for each record $x$ in $T_i$ do        ▷ Compute $v_i$ for all records in $T_i$
16:                    Let $R$ be the records in $T_{i+1}$ associated to $x$
17:                    Set $v_i(x)$ = *select* *Agg*$(R.a_j)$ from $R$ *where* $R.a_k$ *Ref* $c$ [and $R.d < T_0.d$)]
18:                end for
19:            end if
20:        end for
21:        $T_0.v_0$ is a new generated attribute. Depending on the choices in steps 7–13, different attributes may be obtained
22:    end for
23: end procedure
```

in the path will be modified throughout the procedure, let us assume that $T_0, T_1, ..., T_{l-1}$ are copies of the original tables[1]. For $i = l-2$ down to 0, the Roll-up algorithm considers the association $T_i \rightarrow T_{i+1}$ and virtually adds to T_i an attribute v_i built using the information currently contained in T_{i+1}. How to construct v_i depends on the association type and will be explained in greater detail in the next subsection. If $T_i \rightarrow T_{i+1}$ is a *1:1* association, v_i is built by copying any non-ID attribute from T_{i+1} to T_i. For example, given the *1:1* association $Purchase \rightarrow Client$, we can attach to $Purchase$ the attribute $Client.age$, that is, the age of the client who makes the purchase. Note that if T_{i+1} contains a virtual attribute v_{i+1}, then v_{i+1} is the only attribute that must be attached to T_i (step 8). Otherwise, we would generate an attribute that is already generated along a shorter path. If $T_i \rightarrow T_{i+1}$ is a *0:n* association, the value of v_i of each record $x \in T_i$ is computed by applying an aggregation function (and, optionally, a refinement) to the records of T_{i+1} associated to x.

1) Aggregations and Refinements: Let us analyze the case when the Roll-up procedure encounters a *0:n* association $T_i \rightarrow T_{i+1}$ (steps 9–19). Let $a_1, a_2, ..., a_h$ be the attributes of table T_{i+1} and P the path up to T_i, i.e., $P = T_0 \rightarrow T_1 \rightarrow ... \rightarrow T_i$. To generate a virtual attribute v_i, the Roll-up procedure applies 1) an aggregation operator *Agg* to any non-ID attribute a_j and 2) a refinement operator *Ref* for the comparison of any attribute a_k to a quantity c. Aggregation operators summarize the information contained in T_{i+1}, while the refinement operator is a filter on the rows used for the aggregation. The following SQL query, executed in steps 15–18, clarifies the roles of *Agg* and *Ref*

```
Select T1.pk, Agg(T2.aj) From T1
Inner join T2 On T1.pk = T2.fk
Where T2.ak Ref c
Group by T1.pk
```

For example, if T_i is *Client*, T_{i+1} is *Purchase*, a_j = *Purchase.Return*, a_k = *Purchase.Online*, Agg = *AVG*, Ref = *EQUALS*, $c = 1$, the query builds a virtual attribute v_i equal to the proportion of returned online purchases (v_i = AVG(*Purchase.Return*) WHERE *Purchase.Online* = 1) made by the same client. Note that this attribute is a class spoiler, because it uses information on future returns. To avoid this problem, the average needs to be computed on prior purchases. Since *n:n* associations are not allowed, each record y in table T_{i+1} is used to build the virtual attribute of exactly one record in each preceding table. Thus, before the Roll-up procedure starts, it is possible to add to each table in the path a virtual attribute *curDate*, which is the threshold to use (in our case, it is the value of $T_0.Date$). Thus, the query above becomes:

```
Select T1.Client_id, AVG(T2.Return)
```

[1]If the same table is traversed more than once, the path will contain more copies of the same table.

```
From Client T1
Inner join Purchase T2
On T1.Client_id = T2.Client_id
Where T2.Online = 1
And T2.Date < T1.curDate
Group by T1.client_id
```

Whether an aggregation operator *Agg* can be applied to an attribute a_j depends on the type of a_j (step 11), as follows [2]:

- a_j **is an ID**: The only aggregate operator that can be used is "count distinct", which counts the occurrences of distinct values of a_j corresponding to each record in T_i. For example, if $a_j = Purchase.Product\_id$, then we virtually add to the table $Client$ an attribute that counts the number of distinct products purchased by the same client in the past.
- a_j **is categorical**: The only aggregate operators that can be used are "count distinct", "most frequent", and, in case of just two possible values, the proportion of a value. For example, if $a_j = Purchase.Return$, then we can virtually add to the table $Client$ an attribute that is equal to the proportion of purchases returned by the same client in the past.
- a_j **is numeric**: The aggregate operators that can be used are "average", "max", "min". For example, if we define *Return* to be a numerical attribute and $a_j = Purchase.Return$, then we can virtually add to the table $Client$ an attribute that is equal to the maximum value of the attribute *Return* among his or her past purchases. This attribute indicates whether the client has ever returned a purchase.
- a_j **is a date**: The aggregate operators that can be used are "max" and "min". For example, if a_j = *Purchase.Date*, then we can virtually add to the table $Client$ an attribute that is equal to the most recent date when the client purchased something. However, if the goal of the classification problem is to find general rules that are not specific to particular periods in time, then aggregating date attributes should not be allowed.

The optional refinement has the form a_k *Ref* c, where a_k is an attribute of T_{i+1}, *Ref* a suitable refinement operator, and c a quantity compatible with a_k (step 12). If the attributes involved are numerical or dates, the suitable operators are $>$, $\leq$, $=$, $\neq$; if they are categorical or ids, the suitable operators are $=$ and $\neq$. As in [7], we consider two types of refinements: $toValue$ refinements and $comparison$ refinements. In a $toValue$ refinement, c is a constant value; in a $comparison$ refinement, c is a non-virtual attribute of the same dimension as (i.e., compatible with) a_k and contained in a table of P preceding T_{i+1}. Consider, for example, the path $Purchase(1) \rightarrow Client \rightarrow Purchase(2)$, where the numbers *1* and *2* are used to distinguish multiple occurrences of the same table. Let also assume $a_k = Purchase(2).Return$. A possible $toValue$ refinement is

$$\text{where } Purchase(2).Return = 1$$

and a possible $comparison$ refinement is

$$\text{where } Purchase(2).Return \neq Purchase(1).Return$$

Either refinement can be used, together with an aggregation operator, to aggregate information from $Purchase(2)$ to $Client$. For example, if the aggregation operator is "count distinct", then virtual attributes to add to Client may be:

$$v_i = \text{count distinct } Purchase(2).Return \quad \text{where } Purchase(2).Return = 1$$

or

$$v_i = \text{count distinct } Purchase(2).Return$$
$$\text{where } Purchase(2).Return \neq Purchase(1).Return$$

In the first case, v_i is 1 if the client has a return, and 0 otherwise; in the second case, v_i is 1 if the client has a purchase with a return outcome different from the current one. Recall that the goal is to add a new attribute to the table $Purchase$. As explained above, either refinement is added to the date refinement that prevents class spoilers. In this case, the date refinement is: where Purchase(2).Date ¡ Purchase(1).Date.

Analogously to *1:1* associations, if T_{i+1} contains a virtual attribute v_{i+1}, then v_{i+1} must be used either in the aggregation or in the refinement of the current operation (step 13). Consequently, it is useless to generate subpaths $A \rightarrow B \rightarrow A$ where $A \rightarrow B$ is a *0:n* association, because no interesting information can be aggregated on such path. For example, consider the path $Client \rightarrow Purchase \rightarrow Client$: at the first step of the roll-up procedure we copy the attributes of *Client* to *Purchase*; at the second step, when we need to aggregate the purchases of each client, we have to use the virtual attributes in the table *Purchase*, which are the client demographics. If this were allowed, the result would be to compute for each client the average, maximum, minimum, and sum of her or his age, which is clearly useless information. At the end of the Roll-up procedure (step 21), the target table T_0 has a new attribute v_0.

IV. The Dataconda Software

The process of knowledge discovery with Dataconda is performed in three phases:

1) Choose the attribute generation settings (optional);
2) Attribute generation (described in the previous section). This step ends with the construction of the flat mining table;
3) Attribute selection, which results in finding the best predictors.

[2]The examples in the list below are for $T_i = Client$ and $T_{i+1} = Purchase$

A. Attribute Generation Settings

When loading the tables of the database, Dataconda automatically determines the settings to use during the attribute generation procedure. First, it detects the type of attributes (numeric, categorical, date, ID). Then, it automatically determines the aggregate functions and refinements to apply on each attribute based on its type. Aggregate functions for categorical attributes include Count, Count Distinct, and Most Recent Value, whereas aggregate functions for numeric attributes also include Max, Min, Sum, and Avg. Refinements for categorical attributes include $=$ and $\neq$, whereas refinements for numeric attributes also include $\leq$ and $\geq$.

If the user is satisfied with the default settings, then she can proceed with the attribute generation step; otherwise, she can use the graphical interface to override the default settings. A more comprehensive illustration of this step is provided by an online video[3].

The output of the attribute generation procedure is composed of the flat table that includes the generated attributes (both in *csv* and *arff* format) and a dictionary with an English-like description of the generated attributes.

B. Attribute Selection

After generating the attributes, Dataconda will execute an attribute selection procedure and report the result (selected predictors) to the user, as shown in Figure 2. The attribute selection procedure is specified in the *R* language, which can be easily replaced or modified by the user.

The default attribute selection executes a series of Lasso logistic regressions ([11]) by iteratively changing the value of the shrinkage coefficient λ until it finds the value that results in 20 selected attributes. Then, it executes a simple logistic regression using only these 20 attributes, and only those whose *pval* is less than 0.05 are finally returned.

The reason for using Lasso to select attributes lies in the fact that many attributes are highly correlated to each other – for example, *the client's past return rate among products that cost more than $1,000* and *the client's past return rate among products that cost more than $1,200*. In this situation, the Lasso technique will tend to select only the most relevant attribute among each group of highly correlated attributes. The reason for retrieving at most 20 attributes lies in the focus on knowledge discovery rather than predictive performance: a user interested in understanding what drives product returns does not want to be swamped by a large number of predictors to consider.

Despite its simplicity, we show in the next sections that the default attribute selection procedure is very effective in selecting the true predictors among all those generated by Dataconda.

[3]https://www.youtube.com/watch?v=V8dhnddgXEo

V. Simulation Experiments

The goal of the simulation experiments is to assess whether our procedure can (1) generate and (2) retrieve the true predictors in an artificial database.

We build 10 artificial databases with the schema of *Product Returns* (Figure 1) as follows. We first generate the *Product* table by randomly creating 10 products whose prices are uniformly distributed between $10 and $500; then, the *Client* table is randomly populated with 30 clients whose gender is male with probability 0.6 and female with probability 0.4, and whose age is uniformly distributed between 18 and 80 years; finally, the table *Purchase* is populated by simulating purchases through time as follows. Each client makes the first purchase on a random day between Jan 1 2012 and Jan 1 2014, and then makes a number of subsequent purchases uniformly distributed between 9 and 24 throughout their lifetime. Each client's purchases are made according to a Poisson process with a rate of $\frac{1}{15}$ purchases per day. A purchase is made online with a probability of 0.5. Finally, the probability of a return is set as a function of the three *true predictors* p_1, p_2, and p_3 (Table I).

$$\textit{P(Return=1)} = \textit{logit}^{-1}(-3 + 0.02p_1 + 2p_2 - 0.1p_3)$$

The probability of return is positively correlated to p_1, the product price, as suggested by empirical studies in the marketing literature [12]. It is also positively correlated to p_2, the proportion of the client's purchases that were returned, as those who have returned purchases in the past are more likely to return purchases in the future. It is finally negatively correlated to p_3, the age of that client who most recently purchased the current product online. While the correlation between p_3 and *Purchase.Return* may not exist in reality, it allows us to test whether we can detect complex (but also interpretable) true attributes.

In the settings, we enable the aggregate functions and refinements that are "logically compatible" for each attribute. For example, for the attribute *Client.Age*, we enable Max, Min, and Avg, but disable Sum, as the sum of the ages of a group of client does not make sense.

A. Classification Performance

To test whether the generated attributes increase the classification performance, we conduct the following experiment. For each of the 10 databases, we generate 4 flat tables *Purchase* obtained by constructing attributes up to depth 2, 3, 4, and 5. For each flat table, we record the number of attributes generated and the average area under the curve (AUC) obtained by the Weka's [13] implementation of the Random Forest classifier in a 10-fold cross validation. Table II reports the results.

From Table II, it is clear that the AUC increases with the depth only up to depth 4, and then it decreases. This is unsurprising, since the three true variables (Table I) are at depth 2, 3, and 4; adding attributes at depth 5 only increases

Table I
TRUE PREDICTORS

Attr.	**Coeff.**	**Path Depth**	**Description**	**SQL code to manually compute attribute**
p_1	0.02	2	Product price	SELECT pr.Price FROM Purchases pu INNER JOIN Products pr ON pu.Product_ID = pr.Product_ID
p_2	2	3	Client's past return rate	SELECT AVG(pu2.Return) FROM Purchases pu1 LEFT OUTER JOIN Purchases pu2 ON pu1.Client_ID = pu2.Client_ID AND pu1.Date ¿ pu2.Date GROUP BY pu1.Purchase_ID
p_3	-0.1	4	The age of the last client who bought the same product online	SELECT a.Age FROM (SELECT c.Age, RANK() OVER (PARTITION BY pu1.Purchase_ID ORDER BY pu2.Date desc) ranking FROM Purchase pu1 LEFT OUTER JOIN Purchase pu2 ON pu1.Product_ID = pu2.Product_ID AND pu1.Date ¿ pu2.Date AND pu2.Online = 1 INNER JOIN Client c ON pu2.Client_ID = c.Client_ID) a WHERE a.ranking = 1

Table II
AVERAGE CROSS-VALIDATION AUC VALUES

Max Depth	2 (i.e., ACORA [6])	3	4	5
Attributes	4	94	221	3,791
DB #1	.75	.82	.87	.82
DB #2	.86	.90	.92	.89
DB #3	.80	.85	.88	.85
DB #4	.82	.89	.91	.89
DB #5	.70	.77	.85	.77
DB #6	.80	.86	.90	.86
DB #7	.75	.81	.86	.82
DB #8	.84	.88	.91	.89
DB #9	.84	.89	.91	.88
DB #10	.74	.80	.86	.81
Average	.790	.847	.887	.848

Table III
ATTRIBUTE SELECTION PERFORMANCE

DB#	Attributes selected out of 221	True predictors selected (out of 3)	Proxies for true predictors
1	8	3	
2	7	3	
3	8	3	
4	5	2	Client's min return
5	10	3	
6	6	3	
7	7	3	
8	6	2	Client's most recent return
9	5	2	Client's min return
10	6	3	

the chances of overfitting. So, even if the true variables were in fact unknown, Table II would correctly suggest that the attribute generation should be stopped at depth 4.

Note that ACORA ([6]) generates only 4 attributes, as it cannot traverse the *Purchase* table twice, and it therefore results in a much lower classification performance than our procedure.

B. Attribute Selection Performance

In real-world settings, analysts are not only interested in prediction, but they are also (if not more) interested in knowledge discovery. So, let us now turn our attention to the selection of the best predictors. Since the previous experiment suggested to generate attributes up to depth 4, we executed Dataconda's default attribute selection procedure on the flat table generated at depth 4 (with 221 attributes) for each of the 10 databases. Table III reports how many attributes were selected and, among them, how many true predictors (Table I) were selected.

Table III shows that Dataconda's default attribute selection procedure selects 5–10 attributes out of the 221 generated. All three true predictors (p_1, p_2, and p_3) were selected in 7 out of 10 cases, while only two (p_1 and p_3) were selected in the remaining 3 cases. Although the method did not retrieve p_2 in those 3 cases, it retrieved proxies for it, such as the client's minimum return rate (i.e., 1 if the client returned all his or her past purchases, 0 otherwise).

Our results suggest that despite its simplicity, the Dataconda's default attribute selection procedure is capable of retrieving the true predictors most of the times.

VI. REAL-WORLD EXPERIMENTS

We validated our attribute generation procedure on the data of *Circuit City*, which was a large electronic US retailer. The data, described in detail by [14], was purchased from the INFORMS Marketing Science website[4]. It contains two different tables: one with purchases and one with promotions. In this paper, we only use the table with purchases, which contains all the purchases (115,317) made by a subset of clients (19,784).

The data set, which has one row per transaction, contains information on the transaction (subcategory, price, location, whether it is a return or a purchase), information on the client who made the transaction (client_id and demographics characteristics), information on the product (the brand, called "transaction_type_description", the subcategory, the category). The class attribute "return_ind" indicates whether the purchased product is later returned to the store.

[4]https://www.informs.org/Community/ISMS

Table IV
DISCRIMINANT PREDICTORS AND THEIR VALUES IN THE TEST SET

Attribute Description	Attr. Value	Return Probability
1. Did the client return his/her last purchase of $< \$1,500$? i.e., most recent value of *return*(0 or 1) among client's past purchases of $< \$1,500$	0	$7.0\% \pm 0.2\%$
	1	$37.5\% \pm 0.8\%$
2. Has the client ever returned a purchase of less than $1,500? i.e., max value of *return*(0 or 1) among client's past purchases of $< \$1,500$	0	$43.8\% \pm 0.3\%$
	1	$73.5\% \pm 0.8\%$
3. Max *income* among clients with *income* > 7.4 who purchased the same brand in the past	8	$13.5\% \pm 2.1\%$
	9	$10.6\% \pm 0.2\%$
4. Number of purchases made in the past by clients older than 34 in the same location	0-99	$11.0\% \pm 0.2\%$
	100-199	$10.3\% \pm 0.3\%$
	200-199	$9.9\% \pm 0.5\%$
	300-199	$9.8\% \pm 1.2\%$
	400 or more	$3.4\% \pm 2.9\%$

To use our procedure, it is necessary to normalize the data into a set of tables. So, we organized the available data into the following schema:

- *Purchase*: for each purchase, the date, price, quantity, whether the purchase was made online and whether it was returned (class). Each purchase has also foreign keys to all other tables.
- *Client*: for each client, their age, income (as a score from 1 to 9), gender, and whether they have children.
- *Brand*: for each brand, its id.
- *Location*: for each location, its id.
- *Subcategory*: for each subcategory, its id.

The tables are organized in a star schema with the table *Purchase* at the center.

The tables *Brand*, *Location*, and *Subcategory* have only one attribute, their primary key. The only reason for dedicating a table to these entities is to enable Dataconda to explore paths through them, such as *Purchase* $\rightarrow$ *Brand* $\rightarrow$ *Purchase*, which would consequently enable the generation of attributes on the past purchases of the current brand.

Also, note that the database schema above is just one of the possible ways to represent these data in a database, and each different way would result in different attributes generated by Dataconda. For example, an alternative schema would include a table *Product* so as to capture the entire hierarchy *Purchase* $\rightarrow$ *Product* $\rightarrow$ *Subcategory*. However, since the original data set was built by sampling clients and not products, there are a large number of products which are bought by just one client. So, paths that traverse the table *Product* (e.g., the proportion of times that the current product was returned) will result in poor-quality attributes.

To assess the robustness of the findings detected by our procedure, we split the database into a small training database and a large test database. We will use the training database to automatically find new knowledge without the need to make hypotheses; then, we will use the test database to verify the validity of this new knowledge.

The training database was built using all data (purchases, brands, locations, and subcategories) relative to the transactions made by 1,000 randomly selected clients (out of 19,784 available clients), whereas the test database was built using all data relative to the transactions made by the remaining 18,784 clients. This resulted in a training database with 6,305 purchases and a test database with 109,012 purchases.

We used Dataconda to generate all attributes up to depth 4 on the training database. The procedure resulted in the generation of 2,496 attributes. Although the computational time may at first appear long, Section 6.1 shows that our procedure finds drivers of product returns that have not been detected through years of empirical research on the same data set. Thus, our automatic analysis does not only lead to more findings than a manual analysis, but it does so in a much shorter time. The performance of our attribute generation procedure has been improved by generating SQL queries and submitting them to a DBMS. Section 7 shows that this implementation is scalable.

A. Finding New Knowledge

One important advantage of generating attributes automatically rather than manually lies in the ability to find unexpected discriminant predictors. Among the 2,496 attributes generated by Dataconda on the training set, the default attribute selection procedure selected 10. We computed the value of these 10 attributes for all purchases in the test set; then, we computed how their value impacts the class. Due to space limitations, we discuss only four of them, which are reported in Table IV together with the return probability (with the 95% confidence interval) computed in the test set for each attribute value.

The first discriminant attribute is the most recent value of *return* (0 or 1) among the client's past purchases of less than $\$1,500$. This attribute is 1 if the current client returned the last purchase and 0 otherwise. We computed the value of this attribute for the 109,012 purchases of the test set, and found that when this attribute is 0 (i.e., the client did not return the last purchase), the return probability for the current purchase is $7.0\%\pm0.2\%$, whereas when this attribute is 1 (i.e., the client did return the last purchase), the return probability for the current purchase is $37.5\% \pm 0.8\%$. The second discriminant attribute, which is conceptually very similar to the first one, is an indicator of whether the client

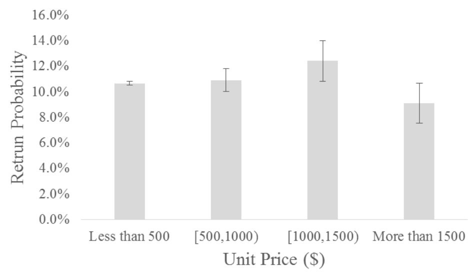

Figure 2. Return probability by price on test set.

has ever returned a product that costs less than $1,500. Also this attribute seems to be highly discriminant, as the return probability is 43.8% for those purchases where its value is 0, and 73.5% for those where its value is 1.

While it is known that clients who returned products in the past tend to return products in the future [12], it is unclear why the refinement *"where Price ¡ $1,500"* appears in both attributes. This specific value ($1,500) was generated by Dataconda's default binning procedure, which splits each numeric attributes in equally wide bins. To understand the significance of this threshold, we divided the purchases of the test set into four partitions based on the price ($0–$500, $500–$1,000, $1,000–$1500, and more than $1,500), and computed the proportion of returned purchases with 95% confidence intervals. Figure 3 reports our results and suggests the first new finding.

FINDING 1: the $1,500 price threshold
As well known ([12]), the price is positively correlated with the probability of return; however, the correlation becomes negative after $1,500. This could be explained by the fact that the best products on the market (which are likely the most expensive) are purchased by customers who are willing to spend a large amount of money; by contrast, good (but not the best) products are purchased by customers who are more mindful about how they spend their money. The second set of customers will be less likely to accept a poor fit between their needs and the product characteristics than the first set, and thus will be more likely to return the product.

Notably, these attributes cannot be generated by other feature construction methods such as ACORA, since they are constructed along the path *Purchase* $\rightarrow$ *Client* $\rightarrow$ *Purchase*, which traverses the table *Purchase* twice.

Another discriminant attribute is attribute 3, the maximum income computed among all clients with income greater than 7.4 who purchased the same brand in the past. Since *income* is an ordinal attribute with values $\{1,2,...,9\}$, the value of this attribute can be only 8 or 9. This attribute seems to suggest that if the richest client who has purchased the same brand has an income of 9, then the return probability is 10.6%, whereas if she or he has an income of 8, then the return probability is 13.5% – note that the richest client has almost always an income of at least 8.

A possible explanation is that this attribute is a proxy of a simpler attribute: the number of clients who have purchased the same brand in the past. However, it can be easily shown that the simpler attribute is not a good predictor of returns. Thus, we formulate the following explanation.

FINDING 2: biased perception of brand
Some brands are mostly used by wealthier clients, either because their products cost more than those of other brands or because the marketing effort of that brand is directed to the more affluent part of society. When a client perceives that a certain brand is used by the wealthy, his or her perception of the brand also improves. Hence, those clients who decide to purchase a product of that brand will be less likely to return it even if the product does not fully meet their expectations, as their perception of the product will be positively biased.

Also attribute 3 cannot be built by ACORA, because it is on a path (*Purchase* $\rightarrow$ *Brand* $\rightarrow$*Purchase* $\rightarrow$ *Client*) that traverses the *Purchase* table twice.

Another discriminant predictor is attribute 4, the number of purchases made in the same location by those older than 34. While we lack a convincing explanation for its high predictive power, its presence shows the ability of our methodology to find non-trivial causes for the phenomenon under study.

VII. Scalability

To assess the scalability of our technique, we executed the attribute generation procedure by varying the sample size of the *Circuit City* database. Here, we use a DBMS-based implementation that submits SQL queries to a DB, instead of computing the attributes in memory.

We generated five different samples of the Circuit City database, which contain about 5% (the same sample as in Section 6), 25%, 50%, 75%, and 100% of the available data. We generated attributes up to depth 3 and up to depth 4. Figure 4 reports the time taken to generate these attributes and the size (in MB) of the resulting flat mining table. All the experiments were conducted on a machine with Intel Core i7-3770 processor (8M Cache, 3.4GHz) and 16GB RAM.

The results presented in Figure 4 show that increasing the size of the database increases the execution time in a nearly-linear fashion if we generate attributes up to depth 3 and in a nearly-quadratic fashion if we generate attributes up to depth 4. This difference is explained by considering that the task of joining four tables is more sensitive to the database size than the task of joining only three tables. The fact that computational times grow faster when generating deeper attributes is largely inconsequential, because excessively deep attributes are generally too hard to interpret and,

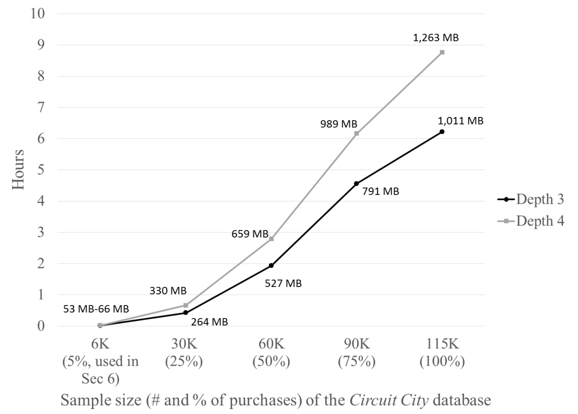

Figure 3. Execution time of the DBMS-based implementation and size of the resulting flat table.

therefore, should not even be generated. In our case study, for instance, most of the attributes at depth 5 are practically uninterpretable.

Although we showed in Section 6 that a small sample (made of 6K purchases) is sufficient to detect interesting findings, the analyst may be interested in running our procedure on the whole data set (made of 115K purchases). Whereas this task will take few hours of computational time on a commodity computer, this time will be certainly smaller than the time needed to formulate hypotheses and to manually build hundreds of handcrafted attributes. Additionally, this computational time will be further reduced in an industrial environment with powerful machines.

VIII. Conclusions

In this paper, we implemented a new methodology which automatically generates the attributes of a mining table from a relational database. Compared to existing approaches, our methodology can explore the same table more than once without generating "class spoilers" and by ensuring that future data, in the case of timestamped transactions, is not used. We applied our methodology to explain the causes of product returns. Our experiments on simulated data confirm that our method can effectively retrieve the true predictors without the need to hypothesize them; our experiments on Circuit City dataset suggest that our method can not only confirm the presence of patterns whose existence is known, but also find new patterns that have never been detected.

We envision a new paradigm for data analysis and empirical research, in which the data analysts use tools like Dataconda on their database in order to find patterns without the need to hypothesize them beforehand.

Our methodology has room for improvement. First, it does not generate some aggregated attributes: for example, attributes summarizing identifiers (as in [6]). Second, although it is scalable, its performance can be further improved. For example, instead of selecting attributes only after generating the entire flat mining table, it is possible to select them while they are being generated, so that they can be deemed to be good or bad predictors after computing their values for only a subset of instances. Finally, our attribute generation procedure could be implemented on platforms like Hadoop or Spark.

References

[1] J. D. Hess and G. E. Mayhew, "Modeling merchandise returns in direct marketing," *Journal of Direct Marketing*, vol. 11, no. 2, pp. 20–35, 1997.

[2] D. Blanchard, "Supply chains also work in reverse," *Industry Week*, vol. 1, pp. 48–49, 2007.

[3] N. Lavrac and S. Dzeroski, "Inductive logic programming." in *WLP*. Springer, 1994, pp. 146–160.

[4] W. F. Clocksin, C. S. Mellish, and W. Clocksin, *Programming in PROLOG*. Springer, 1987, vol. 4.

[5] A. J. Knobbe, M. De Haas, and A. Siebes, "Propositionalisation and aggregates," in *Principles of Data Mining and Knowledge Discovery*. Springer, 2001, pp. 277–288.

[6] C. Perlich and F. Provost, "Distribution-based aggregation for relational learning with identifier attributes," *Machine Learning*, vol. 62, no. 1-2, pp. 65–105, 2006.

[7] M. Samorani, M. Laguna, R. K. DeLisle, and D. C. Weaver, "A randomized exhaustive propositionalization approach for molecule classification," *INFORMS Journal on Computing*, vol. 23, no. 3, pp. 331–345, 2011.

[8] S. Rosset, C. Perlich, G. Świrszcz, P. Melville, and Y. Liu, "Medical data mining: insights from winning two competitions," *Data Mining and Knowledge Discovery*, vol. 20, no. 3, pp. 439–468, 2010.

[9] M. Samorani, "Automatically generate a flat mining table with dataconda," in *2015 IEEE International Conference on Data Mining Workshop (ICDMW)*, Nov 2015, pp. 1644–1647.

[10] ——, "Dataconda: A software framework for mining relational databases," in *DBKDA 2015, The Seventh International Conference on Advances in Databases, Knowledge, and Data Applications*, 2015, pp. 132–133.

[11] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 267–288, 1996.

[12] J. A. Petersen and V. Kumar, "Are product returns a necessary evil? antecedents and consequences," *Journal of Marketing*, vol. 73, no. 3, pp. 35–51, 2009.

[13] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[14] J. Ni, S. A. Neslin, and B. Sun, "Database submission-the isms durable goods data sets," *Marketing Science*, vol. 31, no. 6, pp. 1008–1013, 2012.