Dataconda

User guide

Author:
Michele Samorani

January 9, 2015

Contents

1 Introduction to Dataconda **1**

2 Basic Concepts **3**
2.1 Tables 3
2.2 Attributes 4
2.3 Associations 4

3 An Extensive Tutorial **7**
3.1 Installation 7
3.2 Tables and Table settings 7
3.3 Associations 10
3.4 Attribute Generation 10
3.5 Analyzing the Output 11
3.5.1 Attribute Syntax 12
3.6 Extending Dataconda 14
3.6.1 Write New Analysis Tools 14
3.6.2 Write New Aggregating Functions 15

1

Introduction to Dataconda

One of the most time-consuming, value-adding, and yet undervalued tasks in data mining is *data preparation*. Algorithms from statistics and machine learning typically assume that the input is given: that is, someone prepared a table with one row per observation and one column per predictor. One more column is the dependent variable, which we wish to explain in terms of the predictors. Perhaps there are still some minor adjustments to make, such as standardize, discretize, or combine variables. Despite the need for these minor touches, all the information is assumed to be already contained in the table.

How is the table built in the first place?

Typically, the analyst collects predictors that she thinks are important to predict the dependent variables. For example, if we want to predict whether someone is at risk of a heart attack, we will probably include a binary variable that indicates if the person is a smoker, or whether the person had a heart attack in the past year. Which predictors to construct is often decided through discussions with domain experts. There are two problems with this approach:

1. **It is time consuming**. Building predictors often involves joining different tables, aggregating, and filtering information. These operations typically require writing and maintaining lots of SQL code.
2. **It does not find new knowledge**. This claim may sound surprising to data mining enthusiasts, who see classification and regression algorithms as generators of knowledge. Unfortunately, they are not. Regression or classification rules do not represent new knowledge, because they involve the information that was chosen as most likely to be correlated to the dependent variable. For this reason, this approach makes it very unlikely to find an unexpected pattern.

What should the data mining process be like then?

Given that the current data mining process is inefficient and incapable of finding new knowledge, we need to propose a new data mining process. Ideally, the user's only task should be to gather and describe the data available, and not to decide which predictors to build from it. The available data typically involves different *entities*, which are in certain *relationships* with one another. For example, "Person" is an entity which lives in another entity called "County". The relationship between them is that each county has many people living in it, while each person lives in only one county. Entities and relationships can obviously be modeled through a relational database. After organizing entities in the tables of a relational database, the user will ideally only have to specify what is the dependent variable, and the best predictors will be automatically found by the data mining algorithms.

What does Dataconda do?

Dataconda embeds the ideal data mining process: the user only has to (1) organize the available data in a relational database and (2) indicate a dependent variable, and Dataconda will automatically compute a large number of predictors without the need to formulate hypotheses. These predictors are built by selecting, aggregating, and filtering the information available in the database. From a practical point of view, Dataconda automatically generates the hypotheses to explain the dependent variables.

2

Basic Concepts

Let us introduce three important concepts in Dataconda: the concepts of *Table*, *Attribute*, and *Association.* If you are familiar with database jargon, you can skip this section.

2.1 Tables

A Table is a set of records organized in rows and columns. Tables 2.1 , 2.1 , and 2.1 report a set of purchases, the set of clients who made those purchases, and the set of products sold in those purchases. Each purchase is made by one client and involves one product. A client is characterized by Gender and Age; a product is characterized by a price; a purchase is characterized by a Date of purchase, by whether the purchase was made Online (0/1), and by whether the purchase was eventually Returned (0/1) to the store.

Table 2.1: The *Purchases* table

PurchaseID	Date	Online	ClientID	ProductID	Return
Pur1	Oct 10	1	Cli1	Pro1	1
Pur2	Oct 11	0	Cli2	Pro2	0
Pur3	Oct 14	0	Cli1	Pro2	0
Pur4	Oct 31	0	Cli3	Pro3	1

Table 2.2: The *Clients* table

ClientID	Gender	Age
Cli1	M	33
Cli2	F	45
Cli3	M	28

Table 2.3: The *Products* table

ProductID	Price
Pro1	$200
Pro2	$100
Pro3	$160

2.2 Attributes

The columns of a table are called *attributes*. Generally, attributes represent characteristics of the entities. For example, a purchase (Table 2.1) is characterized by the attributes *Date*, *Online*, and *Return*, which respectively denote the transaction date, whether the purchase was made online, and whether it was later returned. When convenient, we will refer to the attribute a of table T with the notation $T.a$ (e.g., *Purchases.online*). Each attribute is of one of the following types:

- *ID or key*: An ID attribute has the goal of identifying and referring to records in a table. There are two types of IDs: *primary keys* and *foregin keys*. Primary keys are unique identifiers within a table. For example, the primary key of the Clients table is *ClientID* (note that primary keys are generally underlined in the headers of the table). This means that each client has a different *ClientID*. Foreign keys are "pointers" to primary keys. For example, to indicate which client made a purchase, we use *Purchases.ClientID*, which is a foreign key pointing to *Clients.ClientID*. note that the value of a foreign key is taken from the possible values of the referenced primary key. This implies that the value of *Purchases.ClientID* must be present among the values of *Clients.ClientID*.

- *Date*: A date attribute is the timestamp that characterize the entities of the table. For example, *Purchases.Date* is the timestamp at which the transaction occurred.

- *Numeric*: A numeric attribute takes only numeric (real) values. For example, *Products.Price* and *Clients.Age* are numeric.

- *Categorical*: A categorical attribute takes only a finite set of values. For example, *Clients.Gender* and *Purchases.Returned* are categorical. In Dataconda, categorical attributes are stored as text, even if they are number, as in the case of *Purchases.Returned*, which can be 0 or 1. Categorical attributes are also known as factor or nominal attributes.

Attributes are also characterized by a *dimension*, which represents the unit of measurement of that attribute. Dataconda allows attributes of the same dimension to be compared to each other.

2.3 Associations

An association $A \rightarrow B$ is a relationship between two tables A and B. Dataconda considers only two types of associations: *1-to-1* and *0-to-N*. $A \rightarrow B$ is a 1-to-1 association if every record of A is associated to exactly one record

of B. This is the case, for example, of the association *Purchases* $\rightarrow$ *Clients* because any given purchase is made by exactly one client. On the other hand, $A \rightarrow B$ is a 0-to-N association if every record of A is associated to any number of records in B. This is the case, for example, of the association *Clients* $\rightarrow$ *Purchases* because any given client may have made any number of purchases.

Typically, if there is a foreign key from table B pointing to a primary key in table A, as it is the case for B=*Purchases* and A=*Clients*, then $A \rightarrow B$ is 1-to-1 and $B \rightarrow A$ is 0-to-N. In Dataconda, whenever the user declares a 0-to-N associations $A \rightarrow B$, the software will automatically add a 1-to-1 association $B \rightarrow A$.

3

An Extensive Tutorial

In this chapter, we will install Dataconda and work on a problem whose goal is to classify purchases that are kept by the customer and purchases that are instead returned to the store.

3.1 Installation

Download and install the following files:

1. The installer *Dataconda 1.0.msi* from `http://www.dataconda.net/download.html`. If you have a Dataconda license number, insert it after launching Dataconda; if you do not have a license, you will be using Dataconda in "trial mode".

2. The statistical package *R*, available at `http://cran.r-project.org/bin/windows/base/`. This will allow you to run statistical analyses on the generated attributes.

3. (optional but recommended) The data mining software *Weka*, available for download at `http://www.cs.waikato.ac.nz/ml/weka/downloading.html`.

After installing these programs, Dataconda is ready for use. Now, download the "tutorial.zip" file from `http://www.dataconda.net/tutorial.html` and extract it into a "Tutorial" folder on your machine. This zip file contains the tables of the tutorial database (*Purchases.csv*, *Clients.csv*, and *Products.csv*) (as in Table 2.1) and a R file *RTemplate.R* that will analyze the generated attributes.

3.2 Tables and Table settings

Open Dataconda.

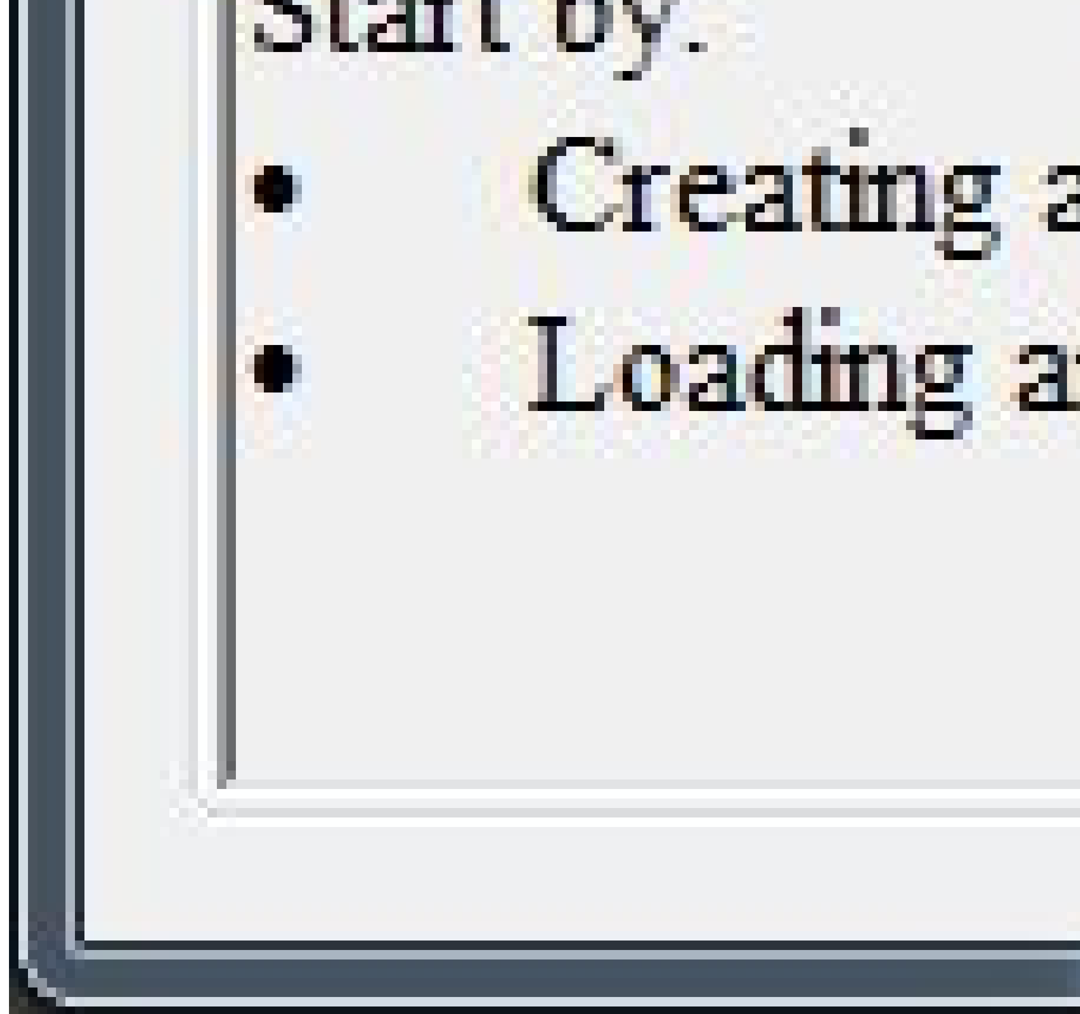

Figure 3.1: Snapshot of Dataconda

The first step is to load the individual tables (*Purchases*, *Products*, and *Clients*) in memory. Click on “New Table” and select the file Purchases.csv. Then, do the same for Products.csv and Clients.csv. Note that the software will generate an error if the file is being used by another process (e.g., by Excel). Also, note that the software expects the first line of the file to be the headers.

The settings that appear in the lower part of the window allow the user to change the metadata relative to each table. First, the user can define the type and dimension of each attribute. By clicking on the button *“Set refinements”*, the user can decide which refinements will be enabled for each attribute. For example, select the table *Clients* from the table list and click the “Set refinements” button relative to its attribute *Age*. There are two types of refinements: “Comparison” refinements and “ToValue” refinements. Let us explain what they are through examples. For more details and examples, the user is referred to [2].

By enabling Comparison refinements and selecting the operator “<” or the operator “>”, the software will generate attributes that compare the age of other customers to the age of the current customer. For example, the software will generate this attribute: *the average price of products purchased by customers older than the current customer*. Comparison refinements will compare attributes of the same dimension. By contrast, by enabling To-

Value refinements, the software will compare the customers' age to fixed values. For example, the software will generate this attribute: *the number of purchases of those customers younger than 23 who bought the current product*. Obviously, for numeric attributes it is often undesirable to consider all possible values; rather, it is advisable to split the range into bins. In Dataconda, bins are built so that their intervals are of the same width, regardless to how many points fall into each bin.

The next column of the Table settings is called "Carries Information". By indicating that an attribute a carries information, the user enables the generation of attributes and refinements based on a. For example, by selecting that the attribute *Purchases.Client_ID*, the software will generate an attribute *Client_ID* and an attribute *Number of times that Client_ID 5267 purchased the same product as the current client*. This may have important consequences when solving the classification problem because such an attribute will generate classification rules based on the client identifiers, such as *"If Client_id = 190290, then Return = 1 with a 30% probability"*. In Dataconda, IDs by default do not carry information.

If an attribute carries information, then the user may select what *aggregating functions* (displayed in columns) should be applied to it. Aggregating functions receive a list of values as input and return a single value as output. Obviously, the type of attribute determines which aggregating functions can be selected. For example, the function *max* is not compatible with categorical attributes. Table 3.1 reports the default functions, their description, and the list of compatible attribute types.

Table 3.1: Aggregating functions

Function	*Description*	*Compatible with*
Max	Computes the maximum	Numeric, Date
Min	Computes the minimum	Numeric, Date
Avg	Computes the average	Numeric
Sum	Computes the sum	Numeric
CountDistinct	Count the number of distinct values	Numeric, ID, Categorical, Date
Count	Count the number of values	Numeric, ID, Categorical, Date
MostFrequent	Returns the most frequent value	Numeric, ID, Categorical, Date
MostRecent	Returns the most recent value (if they are sorted by increasing date)	Numeric, ID, Categorical, Date
Slope	Returns the slope of the values (x is assumed to be 1,2,...,n)	Numeric, Date

By selecting a pair (function, attribute), the user enables the application of a function on an attribute. For example, by selecting the pair (*Max*,*Return*), the software will generate attributes such as: *The maximum value of Return among the past purchases of the client.* Since Return is binary, this attribute can be interpret as a binary indicator of whether the client has ever returned a purchase.

At this point, the user defined the type of attributes, their dimension, and which refinements and aggregating functions can be applied on each of them. The next step is to define the associations among the tables.

3.3 Associations

We now need to declare the associations among tables. In our example, *Clients* is in a 0-to-N association with *Purchases* and *Products* is in a 0-to-N association with *Purchases*. To declare these associations, click on "New Association" and then set Table1 to Clients and Table2 to Purchases. Then, select the IDs that join the two tables: *Clients.Client_ID* and *Purchases.Client_ID*. Finally, click "Create". Then, declare the association *Products*→ *Purchases* through the IDs *Products.Product_ID* and *Purchases.Product_ID*. The metadata has now been completely entered. We can finally proceed to the generation of the attributes.

3.4 Attribute Generation

Click on the central button "Click here to generate attributes" to open the *Generate Attributes* form. This form contains the main options to perform the attribute generation.

First, we need to select the target table *Purchases* (top-left) and the class attribute *Return*. This information will result in the generation of a new table with one row per purchase and a large number of columns, which will hopefully explain the class *Return*. You might notice that upon selecting the class attribute, the same attribute is automatically included in the list of *"Class spoilers"*. Class spoilers are attributes whose most recent value cannot be used to predict the class attribute. The concept of class spoiler is based on the concept of "data leak" [1]. Obviously, we cannot use the current value of *Return* to predict the current purchase, because we would use future information; however, we can use aggregations of past values of *Return*, such as the client's past number of returns. Although in this example *Return* is the only class spoiler, it is possible to include more. For example, if the table *Purchases* had a categorical attribute *Reason for return* ∈ {*"Product was defective", "Client is unsatisfied", "Product was not returned"* } indicating the reason for return the product, this attribute would also be a class spoiler,

because the reason for return would immediately reveal whether the current purchase is going to be returned.

The form also allows the user to choose between generating all the attributes (using the aggregation and refinement operators previously defined) and generating only a subset of attributes. By choosing the second option, the user can write the names of the attributes to generate (one per line). Right now, we'll leave the default option selected ("Generate all attributes").

The rest of the settings are:

- *Time Limit Scan*: the time spent generating attributes in the default order, that is, form the simplest to the most complex, and using a "breadth-first" way of generating the paths along which to generate the attributes.
- *Time Limit Random Pick*: the time spent generating attributes in random order. If the algorithm runs out of time before generating all attributes in the deafult order, it starts generating the remaining ones in random order (that is, not necessarily from the least complex). This strategy has been shown successul in [2].
- *Output Directory*: the folder in which to generate the output. By default, it is the same folder as the .csv files.
- *Max Depth*: the maximum "depth" of the attributes to generate. An attribute at depth d is built along a path formed of d tables (tables may appear more than once). For example, the client's past return rate is at depth 3 because it is built along the path *Purcahses* → *Clients* → *Purchases*: its generation involves (1) joining the table *Clients* with the table *Purchases* in order to attach to the table *Clients* the virtual attribute "Client's past return rate", and (2) joining the table *Clients* with the table *Purchases* to attach this attribute to the target table.

Let us leave all the options unchanged and press "Run". Within a few seconds, Dataconda generates 144 attributes. If you have *R* on your machine, then the list of selected attributes will appear. Note that you can generate a different set of attributes by modifying the selection of aggregation or refinement operators. Finally, note that in the trial version the attribute generation procedure is stopped after 50 attributes.

The attribute generation procedure creates several new files in the output folder. These files are generated not only at the end, but also every x minutes during the attribute generation procedure, where x can be set by the user from the Dataconda settings (default is $x = 1$ minute).

3.5 Analyzing the Output

Let us analyze the files created in the output folder.

- *Data.csv*: the new target table, after adding all generated attributes. In our example, the table in *data.csv* has 145 columns (144 generated attributes and the class attribute *Return*). Obviously, the table still contains one row per purchase, and thus has the same number of rows (487) as the table in *Purchases.csv*. The names of the attributes are randomly generated; their meaning is reported in the file *attributes.csv*.

- *Data.arff*: the new target table in a format compatible with *Weka* [3]. Unlike data.csv, this file contains information on the type of the generated attributes (categorical, numeric, date). So, if you are using Weka, you should use this file rather than data.csv.

- *attributes.txt*: this file reports the list of attributes with a query-like description of their meaning. The syntax used to describe the generated attributes is discussed in section 3.5.1.

- *RScript.R*: the R file, generated from *RTemplate.R*, used to analyze the file data.csv. More details are reported in section 3.6

- *AnalysisOutput.txt*: the output file generated by *Rscript.R*.

In creating these files, Dataconda performs the following actions:

1. Generate *data.csv* and *data.arff*;

2. Generate *RScript.R*;

3. If R is installed, execute the file RScript.R, which will write the output of the analysis into *AnalysisOutput.txt*.

4. Extract the list of the selected attributes from *AnalysisOutput.txt* and display them on the console, together with their description.

3.5.1 Attribute Syntax

The file *attributes.txt* reports the description of the generated attributes. Let us consider a few examples.

Example 1: Client's gender

```
A3287710348852517861_3_3:
Categorical,Gender
DESCRIPTION: Gender of Clients
0:Target->Gender
1:Purchases->Gender
2:Clients.Gender
```

The first line (*A3287710348852517861_3_3*) is the name of the attribute as it appears in the files *data.csv* and *data.arff*. The second line reports its type (*Categorical*) and dimension (*Gender*). The third line is an English-like description ("Gender of Clients"). In most cases, this description is easily understandable. However, for complex attributes, it may be easier to understand the lines that follow. To this end, we need to recall that each attribute is generated in two steps, as explained in greater details in [2]: (1) a path starting from the target table is created and (2) information is iteratively summarized from the end of the path to the beginning of the path.

So, this attribute is built along the path *0:Target* → *1:Purchases* → *2:Clients*, where *Target* is a fictitious target table internally used by Dataconda, which can be safely ignored. So, the attribute is actually generated along the path *1:Purchases* → *2:Clients*. The algorithm considers the table *2:Clients* and summarizes its content into the previous table by adding to *1:Purchases* a virtual attribute "Gender", which represents the gender of the client. In summary, this attribute is the gender of the client who makes the purchase.

Example 2: Indicator of client's return

```
Am7101183015660599292_4p1_4:
Numeric,Return
DESCRIPTION: Max(Return) among past Purchases of Clients
0:Target->Max(Return)
1:Purchases->Max(Return)
2:Clients=>Max(Return) where Date LessThan 1:Date
3:Purchases.Return
```

This attribute represents the maximum value of the attribute *Return* among the past purchases of the current client. It is built along the path *1:Purchases* → *2:Clients* → *3:Purchases*. Let us consider the last association *2:Clients* → *3:Purchases*. Since this association is *0-to-N*, the algorithm needs to aggregate information from *3:Purchases* to *2:Clients*. In this case, the algorithm added to the table *2:Clients* a virtual attribute with the maximum value of the attribute *Return* computed among the client's past purchases. We know that the computation involves only the past purchases because of the refinement `where Date LessThan 1:Date`, which specifies that the date in the rows *3:Purchases* must be less than the date of the current purchase in *1:Purchases*. At the next step, the algorithm considers the path *1:Purchases* → *2:Clients* and attaches the new attribute of *2:Clients* to the table *1:Purchases*.

At the end, we have added to the target table *1:Purchases* an attribute whose value is 1 if the client has has at least one return prior to the current purchase.

Example 3: Average price among the client's non-returned past purchases

```
A6618156281959293617_5p2_5:
Numeric,Price
DESCRIPTION: Avg(Price of Products) where Return LessThan 0.5,
             among past Purchases of Clients
0:Target->Avg(Price)
1:Purchases->Avg(Price)
2:Clients=>Avg(Price) where Date LessThan 1:Date
                      where Return LessThan 0.5
3:Purchases->Price
4:Products.Price
```

From the description, this attribute is the average price of products purchased in the past by the current client, limitedly to the non-returned ones. This attribute is generated by allowing a maximum depth of 4 on Dataconda because it is built along the path *1:Purchases* → *2:Clients* → *3:Purchases* → *4:Products*. Let us consider the last association *3:Purchases* → *4:Products*. Since this association is *1-to-1*, the algorithm simply attaches the product price to each purchase. Then, let us consider the association *2:Clients* → *3:Purchases*. Dataconda needs to aggregate information from *3:Purchases* to *2:Clients*. In this case, the algorithm added to the table *2:Clients* a virtual attribute with the average value of the attribute *Price* computed among the client's past purchases that were not returned. We know that the computation involves only this limited set of purchases because of the refinement `where Date LessThan 1:Date` and of the refinement`where Return LessThan 0.5`. At the next step, the algorithm considers the path *1:Purchases* → *2:Clients* and attaches the new attribute of *2:Clients* to the table *1:Purchases*.

At the end, we have added to the target table *1:Purchases* an attribute whose value is the average price among the client's non-returned past purchases.

3.6 Extending Dataconda

Dataconda can be extended in two ways: by writing new analysis tools or by writing new aggregating functions.

3.6.1 Write New Analysis Tools

As shown in section 3.5, Dataconda generates two files *data.csv* and *data.arff*, which contain the target table obtained after adding the relational attributes

in csv and Weka format. The analysis of this file can be obviously performed outside Dataconda (e.g., in R or weka). However, to simplify the iterative process of defining the correct relational model, Dataconda has the capabilities of automatically running the analysis immediately after the attribute generation procedure.

To this end, the user needs to place a file *RTemplate.R* in the output folder. Dataconda will copy this file into another file *Rscript.R* and execute this second file every time the attribute generation procedure ends. The file *RTemplate.R* should analyze data.csv and write the results of its analysis on a file *AnalysisOutput.txt* located in the same folder. If the output contains a list of selected attributes, Dataconda will display it on its console. The list of selected attributes should be denoted by the header *$attributeNames* or *Selected attributes*. If AnalysisOutput.txt contains either of these headers, Dataconda will retrieve all the attribute names that follow and will report them on the console together with their description.

When writing the *RTemplate.R*, the user may use the following "pseudo-R" code to read from data.csv:

```
data={read.csv("$$csvFileName$$", colClasses=c($$colClasses$$)};
```

Dataconda will substitute *$$csvFileName$$* with the complete path of data.csv (e.g., C:/Dataconda/Tutorial/data.csv) and *$$colClasses$$* with the correct list of attribute types (e.g., *rep('numeric',1),rep('factor',1),rep('numeric',143))*). In this way, the user does not need to know the specific working folder or the type of attributes a-priori.

3.6.2 Write New Aggregating Functions

Aggregating functions have a set of values as input and produce one value as output. They are used by Dataconda to generate attributes along a *0-to-N* association. For example, attributes generated along the association *Clients* → *Purchases* include the client's return rate, which is computed by the aggregating function *Average*, or the client's number of purchases, which is computed by the aggregating function *Count*.

The default aggregating functions are included in the file *Dataconda.aggregatingFunctions.dll*. In Dataconda, the user has the option of specifying the folder in which to find the aggregating functions (*Options* → *Settings*. Upon initialization, Dataconda will scan the *.dll* files in that folder and load all the aggregating functions, which can be recognized by the fact that they implement the interface *dataconda.core.IAggregatingFunction*.

The user can also define her own aggregating functions by creating a class that implements the methods of *dataconda.core.IAggregatingFunction*. The logic of these methods is intuitive and can be easily explained through an example. Suppose that the user wants to implement the aggregating

function *StdDev*, which computes the standard deviation of a set of values. The C# code of the StdDev function is as follows.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Dataconda.core; // need to reference
   dataconda.core.dll and import this namespace

namespace AggregationExample
{
/// <summary>
/// Computes the standard deviation of a set of numbers
/// </summary>
public class StandardDeviation : IAggregatingFunction //
   need to implement this interface
{
    List<double> _values; // will collect the values on
       which to compute the standard deviation

     /// <summary>
     /// The name of the function.
     /// </summary>
     public string Name
     {
         // the name of the function is StdDev
         get { return "StdDev"; }
      }

     /// <summary>
     /// The description of the function: it will appear if
        the mouse passes over the corresponding column.
     /// </summary>
     public string Description
     {
         get { return "Computes the standard deviation"; }
     }

     /// <summary>
     /// The list of attribute types that this function can
        be applied to.
     /// </summary>
```

```
public ICollection<AttributeType>
    SupportedAttributeTypes
{
    // StdDev can be computed only among numeric
        attributes
    get { return new List<AttributeType> {
        AttributeType.Numeric }; }
}

/// <summary>
/// A subset of the SupportedAttributeTypes. They will
    be checked by default.
/// </summary>
public ICollection<AttributeType> DefaultAttributeTypes
{
    // standard deviation will not be selected by
        default
    get { return new List<AttributeType>() { }; }
}

/// <summary>
/// The dimension of the value returned, given the
    dimension of the values in input.
/// </summary>
/// <param name="value"></param>
/// <returns></returns>
public string GetDimension(string value)
{
    // if we compute the StdDev of an attribute of
        dimension d, the result is of dimension d as
        well
    return value;
}

/// <summary>
/// The type of the output attribute, given the type
    of the input attribute.
/// </summary>
/// <param name="value"></param>
/// <returns></returns>
public AttributeType GetResultType(AttributeType value)
{
    // the result of StdDev is always a numeric
    return AttributeType.Numeric;
```

```
}

/// <summary>
/// This method is called before starting to scan the
   values among which to compute the aggregation.
/// </summary>
public void Initialize()
{
    // we will hold all the values in a List<double>
    _values = new List<double>();
}

/// <summary>
/// This method is called every time a value is
   scanned.
/// </summary>
/// <param name="value"></param>
public void TreatDataRow(IComparable value)
{
    if (value == null)
        return;
    // value can be of three types: string, double,
       and date. Because we declared that
    // StdDev only takes numeric attributes as input,
       value will always be double.

    double dv = (double)value;
    if (Double.IsNaN(dv))
        return; // if it is null, skip it from the
           computation

    _values.Add(dv); // add it to the list of values
}

/// <summary>
/// This method is called at the end to compute the
   result.
/// </summary>
/// <returns></returns>
public IComparable GetValueFor()
{
    // compute the average
    double average = 0;
    foreach (double d in _values)
```

```
                average += d;
            average /= (_values.Count + 0.0);

            // compute the sum of the squared difference
            double sumSq = 0;
            foreach (double d in _values)
                sumSq += Math.Pow(d - average, 2);

            // return the standard deviation
            return Math.Sqrt(sumSq / (_values.Count + 0.0));
        }
    }
}
```

Bibliography

[1] S. Rosset, C. Perlich, G. Świrszcz, P. Melville, and Y. Liu. Medical data mining: insights from winning two competitions. *Data Mining and Knowledge Discovery*, 20(3):439–468, 2010.

[2] M. Samorani, M. Laguna, R. K. DeLisle, and D. C. Weaver. A randomized exhaustive propositionalization approach for molecule classification. *INFORMS Journal on Computing*, 23(3):331–345, 2011.

[3] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.