ZEMENTIS Predictive Analytics Extensions to PMML

ZEMENTIS Predictive Analytics
Extensions to PMML

Software AG

This document applies to ZEMENTIS Predictive Analytics and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

1 Table of Contents

1 Introduction

ZEMENTIS server and ZEMENTIS Plugins support PMML versions 2.0 through 4.3. They do that by incorporating a PMML converter, which is able to convert older versions of PMML to its latest. The converter is also tasked with the correction of known issues with automatically exported PMML code from different model building tools.

To represent certain features not present in standard PMML, a few extensions had to be defined so that the models using these features would be processed correctly when executed in one of our scoring products. In this way, whenever a known non-standard feature is encountered, the PMML converter automatically moves it into an extension that makes sense for scoring. These extensions and customizations are defined below.

2 Activation Functions

A Neural Network Model in PMML is represented by the element *NeuralNetwork*. This element is composed of a series of elements including *NeuralLayer* which encapsulates the neurons in that layer. In PMML, a single activation function may be defined for the entire network or a different activation function may be defined for each neural layer. Typical activation functions include logistic, hyperbolic tangent, and Gauss. ZEMENTIS server and ZEMENTIS Plugins define two extensions targeted towards activation functions.

2.1 Activation Function: Stretched Logistic

The first is an activation function modifier. It is a stretched up version of the logistic activation function and when used, it transforms the logistic activation function from *activation(Z) = 1/(1+exp(-Z))* to *activation(Z) = -1 + 2/(1+exp(-Z)).* The extension per se is defined as:

```
<Extension
  name="ACTIVATION_FUNCTION"
  value="Stretch"
  extender="ADAPA" />
```

For example, the PMML code shown in Figure 1 implements the setting up of the element *NeuralNetwork* for a neural network that uses the logistic activation function with the stretch modification.

```
<NeuralNetwork
   modelName="ADAPABackPropagationModel"
   functionName="classification"
   activationFunction="logistic"
   numberOfLayers="2">
   <Extension name="ACTIVATION_FUNCTION"
      value="Stretch" extender="ADAPA"/>
   <!-- MiningField elements -->
   <!-- NeuralInput elements -->
   <!-- NeuralLayer elements -->
   <!-- NeuralOutput element -->
</NeuralNetwork>
```

Figure 1. Setting up a *NeuralNetwork* element in PMML with the stretch modification

2.2 Activation Function: MaxPool

The second extension defines an activation function designed to mimic a network layer which simply collects and down-samples data from the previous layer; this is a common layer in deep networks. When used, it takes the input to all the neurons in the layer and outputs the maximum value of the inputs. In other words, using the notation from the PMML schema definition, *activation(Z) = max(output(i)).* Note that the step combining the outputs is skipped.

```
<Extension
  name="ACTIVATION_FUNCTION"
  value="maxPool"
  extender="ADAPA" />
```

2.3 Activation Function: Gauss

The third extension defines an alternative to the Gaussian activation function defined by the PMML specification. When used, it converts it from *activation(Z) = exp(-(Z*Z))* to *activation(Z) = exp(-(Z*Z)*0.5)*. The extension is defined as:

<Extension
  name="*ACTIVATION_FUNCTION*"
  value="*GaussMean0StD1*"
  extender="*ADAPA*" />

Note that the *Extension* element, when present, takes precedence in defining the activation function even if attribute *activationFunction* is explicitly defined as part of the *NeuralNetwork* or *NeuralLayer* element.

3 Thermometer Coding

When thermometer coding is specified, the outputs of a neural network are to be interpreted as cumulative probabilities. In this way, to compute the output of any category other than the first, one must take the difference between successive outputs.

The extension for representing thermometer coding is define as:

```
<Extension
  name="NORM_DISCRETE_METHOD"
  value="Thermometer"
  extender="ADAPA" />
```

For example, the PMML code shown in Figure 2 implements the use of thermometer coding for the *NeuralOutput* elements of a neural network. Note that although the example assumes eleven *NeuralOutput* elements, only three are explicitly defined.

```
<NeuralNetwork ...>
   <!-- MiningField elements -->
   <!-- NeuralInput elements -->
   <!-- NeuralLayer elements -->
    <NeuralOutputs numberOfOutputs="11">
      <NeuralOutput outputNeuron="Output_10">
        <DerivedField dataType="string" optype="categorical">
          <NormDiscrete field="Final" value="10">
            <Extension extender="ADAPA"
             name="NORM_DISCRETE_METHOD" value="Thermometer"/>
          </NormDiscrete>
        </DerivedField>
      </NeuralOutput>
      <NeuralOutput outputNeuron="Output_9">
        <DerivedField dataType="string" optype="categorical">
          <NormDiscrete field="Final" value="9">
            <Extension extender="ADAPA"
             name="NORM_DISCRETE_METHOD" value="Thermometer"/>
          </NormDiscrete>
        </DerivedField>
      </NeuralOutput>
      <!- ... -->
      <NeuralOutput outputNeuron="Output_0">
        <DerivedField dataType="string" optype="categorical">
          <NormDiscrete field="Final" value="0">
            <Extension extender="ADAPA"
             name="NORM_DISCRETE_METHOD" value="Thermometer"/>
          </NormDiscrete>
        </DerivedField>
      </NeuralOutput>
   </NeuralOutputs>
</NeuralNetwork>
```

Figure 2. Setting up a *NeuralNetwork* element in PMML with the help of an *Extension* element used to implement neural outputs with thermometer coding

4 Topology Representing Network

Topology Representing Network (TRN) is a self-organizing network that can be formulated as a combination of a vector quantization scheme and a competitive Hebbian rule. The vector quantizer implemented in TRN is known in the literature as the neural-gas algorithm (T. M. Martinetz and K. J. Schulten. Topology Representing Networks. *Neural Networks*, Vol. 7, No. 3, pp. 507-522, 1994).

TRN networks are usually represented by a single manifold of neurons. Figure 3 shows the self-organizing dynamics of this manifold during learning. It depicts the development of a two-dimensional network. Initially, the network is presented with equally distributed random numbers (*t = 0*) and the neural gas vector quantization algorithm distributes the weights matching the input probability distribution. At the same time, the competitive Hebb-rule introduces connections between the units resembling the topology of the input manifold.

Note that in *t = 0*, the neurons have not yet started to map the input data. In *t = 100*, the first connections have already been established. Finally, in *t = 100,000* the network resembles the topology of the input space.

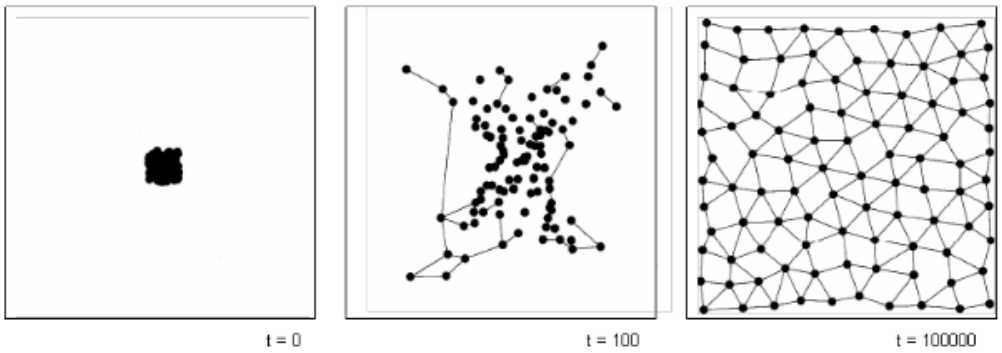

Figure 3. Neuron dynamics in a TRN

TRN is not a modeling technique supported by the PMML standard. The representation depicted here, although particular to ZEMENTIS, is very much in line with how PMML represents neural networks. A TRN network is therefore implemented as a multi-layer feed-forward entity in which the input variables are mapped to a neural input layer, the self-organizing network (or manifold of neurons) is represented as a hidden layer, and the output of the network is represented by an output layer containing a single node.

4.1 TRN in PMML

TRN is represented in PMML by the element *NeuralNetwork* in which the following attribute-value pair needs to be specified:
Attribute *activationFunction* should contain the value "identity".

Also, to identify that the TRN algorithm is to be run, ZEMENTIS requires the PMML code representing TRN to be defined as an *Extension* as follows:

```
<Extension
  name="NEURAL_NETWORK_TYPE"
  value="TopologyRepresentingNetwork"
  extender="ADAPA" />
```

For example, the PMML code shown in Figure 4 implements the setting up of the element *NeuralNetwork* for TRN. Note that the number of layers is defined as "2" (attribute *numberOfLayers*) since the input layer is not considered. Also, note the required *Extension* element.

```
<NeuralNetwork
   modelName="ADAPATRNModel"
   functionName="classification"
   activationFunction="identity"
   numberOfLayers="2">
   <Extension name="NEURAL_NETWORK_TYPE"
      value="TopologyRepresentingNetwork" extender="ADAPA"/>
   <!--
      Other PMML elements such as MiningSchema
      as well as model specific elements
   -->
</NeuralNetwork>
```

Figure 4. Setting up a TRN *NeuralNetwork* element in PMML with the help of an *Extension* element

As mentioned above, TRN networks are represented in ZEMENTIS server and ZEMENTIS Plugins as feed-forward networks in a very similar manner as back-propagation networks. The only difference between these is the number of hidden nodes. In back-propagation networks, the number of hidden neurons is usually small, whereas the manifold of neurons in TRN can be composed of hundreds of neurons. Therefore the representation of the hidden-layer in TRN can be quite large.

5 Combining Multiple PMML Files

ZEMENTIS provides a flexible way to combine multiple models in different PMML files. It does this by allowing a model to invoke one or more external models as functions. Note that this feature is not available in ZEMENTIS Plugins.

In ZEMENTIS, each model that gets imported is automatically made available to be invoked by another model as a function. The name of the generated function is “model:model_name”, where “model:” is a fixed prefix, creating a separate name space for functions backed by models, and “model_name” is the name of the corresponding model. The input parameters of the function are created by the input fields of the model, in the order they appear in the mining schema. The names of the input fields are not important. Their data types are as they determine the data type of the corresponding input parameter of the function.

The output data type of the function, i.e. the data type of the returned values, depends on the number of output fields of the model. For models with a single output field, the output data type is that of the single output field. For models with more than one output field, the output data type of the function is always “string” and the returned values are a JSON representation of all the output values of the model. As discussed in section 6.2, the JSON representation allows a function to return multiple values in a structured object. ZEMENTIS also provides functions that help pick individual values out of the structured object.

Let’s consider first an example of a model that invokes another model with a single output field. Figure 5 shows the input and output fields of a regression model, which we will be using as an example of a “child” model. It has six input fields of type double and a single output field, also of type double. Note that the model element contains attribute *modelName*. Although this attribute is optional in PMML, it is required if being used to

define external models as functions. In the example shown here, the “child” model is named *ElNino_LR*.

```
<DataDictionary numberOfFields="6">
   <DataField name="airtemp" optype="continuous"
      dataType="double" />
   <DataField name="humidity" optype="continuous"
      dataType="double" />
   <DataField name="latitude" optype="continuous"
      dataType="double" />
   <DataField name="longitude" optype="continuous"
      dataType="double" />
   <DataField name="mer" optype="continuous"
      dataType="double" />
   <DataField name="zon" optype="continuous"
      dataType="double" />
</DataDictionary>
<RegressionModel modelName="ElNino_LR" functionName="regression"
   modelType="linearRegression">
   <MiningSchema>
      <MiningField name="airtemp" usageType="active" />
      <MiningField name="humidity" usageType="active" />
      <MiningField name="latitude" usageType="active" />
      <MiningField name="longitude" usageType="active" />
      <MiningField name="mer" usageType="active" />
      <MiningField name="zon" usageType="active" />
   </MiningSchema>
   <Outputs>
      <OutputField name="temp" optype="continuous"
         dataType="double" feature="predictedValue"/>
  </Outputs>
```

Figure 5. Multiple Models: Input and output fields of a “child” model with a single output field.

Another model, which we may think of as the “parent” model, may invoke the *ElNino_LR* model by using the generated function *model:ElNino_LR* anywhere where an *Apply* expression may be used. Typically, this is done in a derived field in the *LocalTransformations* section, as shown below in Figure 6.

```
<DerivedField name="elnino_lr_score" optype="continuous"
   dataType="double">
   <Apply function="model:ElNino_LR">
      <FieldRef field="my_airtemp"/>
      <FieldRef field="my_humidity"/>
      <FieldRef field="my_latitude"/>
      <FieldRef field="my_longitude"/>
      <FieldRef field="my_mer"/>
      <FieldRef field="my_zone"/>
   </Apply>
</DerivedField>
```

Figure 6. Multiple Models: Example of "parent" model invoking "child" model with a single output field.

Now let's consider an example of a "child" model with multiple output fields. Figure 7 presents the input and output fields of a classification model. The model has four input fields of type double and four output fields, one of type string (the predicted class) and three of type double (the probability for each of the possible classes).

```
<DataDictionary numberOfFields="4">
  <DataField dataType="double" name="sepal_length"
    optype="continuous" />
  <DataField dataType="double" name="sepal_width"
    optype="continuous" />
  <DataField dataType="double" name="petal_length"
    optype="continuous" />
  <DataField dataType="double" name="petal_width"
    optype="continuous" />
</DataDictionary>
<NeuralNetwork activationFunction="tanh"
  functionName="classification" modelName="Iris_NN">
  <MiningSchema>
    <MiningField name="sepal_length" />
    <MiningField name="sepal_width" />
    <MiningField name="petal_length" />
    <MiningField name="petal_width" />
  </MiningSchema>
  <Output>
    <OutputField dataType="string" feature="predictedValue"
      name="class" />
    <OutputField dataType="double" feature="probability"
      name="Probability_setosa" optype="continuous"
      value="Iris-setosa" />
    <OutputField dataType="double" feature="probability"
      name="Probability_versicolor" optype="continuous"
      value="Iris-versicolor" />
    <OutputField dataType="double" feature="probability"
      name="Probability_virginica" optype="continuous"
      value="Iris-virginica" />
  </Output>
```

Figure 7. Multiple Models: Example of "child" model with multiple output fields.

The way to invoke this model from another one is similar to the single output field example. However, the return value of the model will be a JSON representation of the predicted values. This would require defining additional derived fields to select the individual values from the JSON result. This is shown in the example in Figure 5. The

Apply element for the derived field *json_nn_result* invokes the model *Iris_NN* and the result of that invocation is stored in that field as a JSON string. The other four derived fields use the provided JSON functions (see section 6.2) to pick the different values out of this JSON string. Note that, depending on the type of an output field, the appropriate JSON function should be used to retrieve the value of the correct data type. In this example, in order to retrieve the *class* prediction, which is of type string, we use the function *zementis:jsonString*, and for the three predicted probabilities, which are of type double, we use the function *zementis:jsonDouble*.

```
<DerivedField dataType="string" optype="categorical"
  name="json_nn_result">
  <Apply function="model:Iris_NN">
    <FieldRef field="my_sepal_length" />
    <FieldRef field="my_sepal_width" />
    <FieldRef field="my_petal_length" />
    <FieldRef field="my_petal_width" />
  </Apply>
</DerivedField>
<DerivedField name="nn_class" dataType="string"
  optype="categorical">
  <Apply function="zementis:jsonString">
    <FieldRef field="json_nn_result" />
    <Constant>$.class</Constant>
  </Apply>
</DerivedField>
<DerivedField name="nn_setosa_prob" dataType="double"
  optype="categorical">
  <Apply function="zementis:jsonDouble">
    <FieldRef field="json_nn_result" />
    <Constant>$.Probability_setosa</Constant>
  </Apply>
</DerivedField>
<DerivedField name="nn_versicolor_prob" dataType="double"
  optype="categorical">
  <Apply function="zementis:jsonDouble">
    <FieldRef field="json_nn_result" />
    <Constant>$.Probability_versicolor</Constant>
  </Apply>
</DerivedField>
<DerivedField name="nn_virginica_prob" dataType="double"
  optype="categorical">
  <Apply function="zementis:jsonDouble">
    <FieldRef field="json_nn_result" />
    <Constant>$.Probability_virginica</Constant>
  </Apply>
</DerivedField>
```

Figure 5. Multiple Models: Example of "parent" model calling a "child" model with multiple output fields.

6 Functions

PMML implements a large set of built-in functions. Nonetheless, ZEMENTIS server and ZEMENTIS Plugins offer extra functions. These functions can be divided into three categories: 1) functions to make it easier to handle date values; 2) functions that allow string manipulations using regular expressions and patterns; and 3) functions that implement JSON functionality. The three groups of functions are described below. Note that whenever used, these functions need to be preceded by the namespace "zementis:" which identifies that these are ZEMENTIS' extensions to PMML and not standard built-in functions.

6.1 Functions for Date/Time Handling

Function zementis:toDate

This function is somewhat the reverse of the PMML built-in function "formatDatetime". It parses a string representation of a date value to a value of type "date". Its arguments are a string representation of a date value and a string representing the format that value is in. It returns the parsed date value. This format string follows the pattern described in the Java class:

org.joda.time.format.DateTimeFormat

For more information, refer to:

http://joda-time.sourceforge.net/apidocs/org/joda/time/format/DateTimeFormat.html

For example, the PMML code shown in Figure 9. PMML code using the function *zementis:toDate* contains three derived fields. While the first one is used to define the desired format, the last two use the function “toDate” to parse a string value to an actual date value.

```
<DerivedField name="date_format"
  optype="categorical" dataType="string">
  <Constant>MM/dd/yyyy</Constant>
</DerivedField>
<DerivedField name="bankruptcy_discharged_date"
  optype="ordinal" dataType="date">
  <Apply function="zementis:toDate">
    <FieldRef field="string_bankruptcy_discharged_date"/>
    <FieldRef field="date_format"/>
  </Apply>
</DerivedField>
<DerivedField name="loan_created_date"
  optype="ordinal" dataType="date">
  <Apply function="zementis:toDate">
    <FieldRef field="string_loan_created_date"/>
    <FieldRef field="date_format"/>
  </Apply>
</DerivedField>
```

Figure 9. PMML code using the function *zementis:toDate*

Function zementis:toTime

```
This function parses a string representation of a time value to a value of type
“time”. Its arguments are a string representation of a time value and a string
representing the format that value is in. It returns the parsed time value.
This format string follows the pattern described in the Java class:
org.joda.time.format.DateTimeFormat
```

For more information, refer to:

http://joda-time.sourceforge.net/apidocs/org/joda/time/format/DateTimeFormat.html

For example, the PMML code shown in Figure 10 shows two examples of using the “toTime” function to parse a string value to an actual time value.

```
<DerivedField name="purchase_time"
  optype="ordinal" dataType="time">
  <Apply function="zementis:toTime">
    <Constant>15:24:32.000</Constant"/>
    <Constant>HH:mm:ss.SSS</Constant"/>
  </Apply>
</DerivedField>
 <DerivedField name="purchase_time"
  optype="ordinal" dataType="time">
  <Apply function="zementis:toTime">
    <Constant>03:24:32PM</Constant"/>
    <Constant>KK:mm:ssa</Constant"/>
  </Apply>
</DerivedField>
```

Figure 10. PMML code using the function *zementis:toTime*

Function zementis:toDateTime

This function parses a string representation of a date/time value to a value of type "dateTime". Its arguments are a string representation of a date/time value and a string representing the format that value is in. It returns the parsed dateTime value. This format string follows the pattern described in the Java class:
org.joda.time.format.DateTimeFormat
For more information, refer to:

http://joda-time.sourceforge.net/apidocs/org/joda/time/format/DateTimeFormat.html

For example, the PMML code depicted in Figure 11 shows two examples of using the "toDateTime" function to parse a string value to an actual time value. Notice, that the format string in the second example does not specify a time zone. In such a case, the time zone is assumed to be UTC.

```
<DerivedField name="purchase_date_time"
  optype="ordinal" dataType="dateTime">
  <Apply function="zementis:toDateTime">
    <Constant>2013-05-19 14:24:32-0700</Constant>
    <Constant>yyyy-MM-dd HH:mm:ssZ</Constant>
  </Apply>
</DerivedField>
 <DerivedField name="purchase_date_time_UTC"
  optype="ordinal" dataType="time">
  <Apply function="zementis:toDateTime">
    <Constant>2013-05-19 21:24:32</Constant>
    <Constant>yyyy-MM-dd HH:mm:ss</Constant>
  </Apply>
</DerivedField>
```

Figure 11. PMML code using the function *zementis:toDateTime*

Function zementis:daysFromTo

This function computes the number of days between two date values. For example, the PMML code shown in Figure 12 depicts the “daysFromTo” function being invoked to calculate a derived value representing the number of days since bankruptcy in which the first date attribute contains the bankruptcy discharge date and, the second, the date in which the loan was created. Note that both of these dates were obtained from using the function “toDate” as depicted in Figure 9 and so are represented in the date format MM/dd/yyyy.

```
<DerivedField name="days_since_bankruptcy"
  optype="continuous" dataType="double">
  <Apply function="zementis:daysFromTo">
    <FieldRef field="bankruptcy_discharged_date"/>
    <FieldRef field="loan_created_date"/>
  </Apply>
</DerivedField>
```

Figure 12. PMML code using the function *zementis:daysFromTo*

6.2 JSON Functions

When a PMML model relies on an external system to retrieve values required for the score computation, a custom function can be developed to implement such a requirement. Often, multiple values may be needed from the external system. However, with PMML, each function may return only a single value. This means that retrieving multiple values from an external system would require multiple function calls. In turn, these function calls result in multiple remote calls to the external system, which can be expensive in terms of performance.

To overcome this limitation, ZEMENTIS allows creating custom functions that return several values (a structured object) in the form of a JSON string in a single call (one round trip Vs. multiple round trips). In addition, it provides functions that, using JSON path expressions, pick the values you care about out of the JSON structure (locally).

Below is a list of the JSON functions supported. Each of these functions takes as input a JSON string and a JSON path. It returns a value (of the appropriate type) located in the provided path. For details on the format of the supported JSON paths, please refer to:

https://code.google.com/p/json-path/

1) zementis:jsonString
2) zementis:jsonDouble
3) zementis:jsonFloat
4) zementis:jsonInteger
5) zementis:jsonBoolean
6) zementis:jsonDate
7) zementis:jsonTime
8) zementis:jsonDateTime

As an example, assume you would like to look for specific information about a particular product or item. Assume you have a system that, given the barcode of a particular item, returns all the information pertaining to that item such as name, price, origin, etc. Such information can be implemented through a custom function. In this example, this function is called “findItemInfo”. When invoked, “findItemInfo” needs to return more than one value. This cannot be done through a normal PMML function. However, “findItemInfo” can pack all the information it needs to return into a JSON object and

represent it as a string. For example, if we pass barcode “4131” as the input argument, function “findItemInfo” will return a JSON string such as:

{“name”:FujiApple, “price”:1.29, “organic”:true, “state”:Oregon}

The PMML code that uses function “findItemInfo” is shown in Figure 13. Note that field “barcode” is used as the input argument. Once the function is invoke, the resulting JSON string will be assigned to derived field “itemInfoJSONString”.

```
<DerivedField name="itemInfoJSONString"
  dataType="string"                                   optype="categorical">
  <Apply                                          function="custom:findItemInfo">
                                          <FieldRef              field="barcode"/>
  </Apply>
</DerivedField>
```

Figure 13. PMML code using custom function *custom:findItemInfo*

We can then use one or more of the JSON functions listed above to find out, for example, the name and the price of item “4131”. Figure 14 shows the PMML code for two derived fields that do just that. These are named “itemName” and “itemPrice”. They use functions “jsonString” and “jsonDouble”, respectively, to parse out name and price from the JSON string returned by custom function “findItemInfo”.

```
<DerivedField name="itemName"
```

```
  dataType="string"                                          optype="categorical">
  <Apply                                                  function="zementis:jsonString">
    <FieldRef field="itemInfoJSONString"/>

                                                          <Constant>name<constant/>
  </Apply>
</DerivedField>

<DerivedField name="itemPrice"

  dataType="string"                                          optype="categorical">
  <Apply                                                  function="zementis:jsonDouble">
    <FieldRef field="itemInfoJSONString"/>

                                                          <Constant>price<constant/>
  </Apply>
</DerivedField>
```

Figure 14. PMML code using JSON functions

7 Conclusion

The extensions and functions presented in this document aim to extend the functionality of PMML. It is our goal to eventually have them as part of the standard itself. For that, a proposal needs to written and subsequently presented to the Data Mining Group (DMG). If approved by the DMG, the new functionality is then reflected in a new version of the PMML standard. Typically, a new version is released every two years.

In addition to the extensions described in this document, the ZEMENTIS also allows for custom functions coded in Java to be dynamically uploaded as resources. In this way, custom functionality can be directly used from inside the PMML code. This feature gives ZEMENTIS yet another powerful way to represent computations that are not part of the PMML standard, but that may still be required for the full deployment of a predictive solution. For details on this ZEMENTIS feature, please refer to the “ZEMENTIS Solutions Guide”.