ZEMENTIS for Hive

User Guide

10.4.0.0

ZEMENTIS for Hive

User Guide

This document applies to ZEMENTIS 10.4.0.0 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.

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

As advanced analytics becomes pervasive across the enterprise to derive better business decisions, the need for efficient execution of predictive models is paramount. An ever growing array of data mining tools and, all too often, custom specialized software is used to mine and derive statistical models from a wealth of historical data. The ultimate goal is to turn these models into business value by incorporating them into day-to-day business operations. This necessitates the ability to integrate them into the IT infrastructure where outcomes can easily flow into the finger-tips of the decision makers. At the same time, the accelerating growth rate of data collected implies that only the most scalable database architectures will be able to meet storage, and more importantly, processing requirements.

In the era of big data, more and more organizations are turning into the scalable architecture of Hadoop and Hive to meet this growing challenge. To bring the power of predictive models into this architecture, Software AG has developed the ZEMENTIS Predictive Analytics (ZEMENTIS) for Hive. ZEMENTIS offers Hive users the best combination of open standards and scalability for the application of predictive analytics. With the Predictive Model Markup Language (PMML) as the bridge between the model development environment and the IT data warehousing infrastructure, ZEMENTIS for Hive offers standards-based deployment of predictive models and execution on a highly scalable platform. This solution brings the power of ZEMENTIS Predictive Analytics server, the flagship product of Software AG, to the Hadoop and Hive infrastructure to deliver superior performance for mission-critical business intelligence, analytics and data warehousing solutions. As a result, a wide range of predictive models, possibly developed with different tools in different environments, can be effortlessly and seamlessly embedded directly in the warehouse. Practically, PMML becomes a HiveQL function offering execution performance that can meet the volume and performance requirements of the most demanding environments.

This document serves as a guide for installing and using ZEMENTIS for Hive. It first gives a brief overview of the plug-in, describes each of its components, and explains how these are combined. It then presents the simple installation process. Finally, it illustrates the use of ZEMENTIS with two PMML examples, a neural network and a decision tree. These examples show how to deploy and execute predictive models in Hive.

Note

With respect to the EU General Data Protection Regulation (GDPR), our product does not collect or store any personally identifiable information. However, as the input data might contain sensitive personal information, please anonymize any such data to ensure that the processing of personal data is in accordance with the GDPR.

Chapter 2. Overview

2.1. Predictive Model Markup Language (PMML)

As the de-facto standard for data mining models, PMML provides tremendous benefits for business, IT, and the data mining industry in general. Developed by the Data Mining Group (DMG), an independent, vendor-led consortium, PMML increases business agility by eliminating the need for proprietary solutions or custom code development. With PMML, a model can transit as is from the data scientist's desktop to the deployment platform where it will be executed.

Today, PMML is supported by all the leading data mining tools, commercial and open source. As an open standard, it enables project stakeholders to standardize on one common representation for data mining models. It practically eliminates the barriers and gaps between development and production deployment of predictive analytics. In effect, it minimizes the complexity, cost, and time to turn predictive models into operational IT and business assets.

As the lingua franca for predictive analytics, data mining models can be easily exchanged between PMML-compliant applications. In this way, a model may be built in one statistical tool and easily moved to another for production deployment or visualization. PMML also serves as a bridge between all the teams involved in the data mining process inside a company as it can be used to disseminate knowledge and best practices, thereby stimulating cross-team and inter-organization collaboration. In a world in which data-driven decisions are becoming more and more pervasive, predictive analytics and standards such as PMML make it possible for organizations to benefit from smart solutions that will truly revolutionize their business.

Besides offering a rich set of structures for describing all the intricate details of a predictive algorithm, PMML also provides information about the input and output of a model. This includes names and types of all input and output data fields, often along with the set of permissible values. In addition, a model expressed in PMML typically includes information about how to handle invalid, or missing or outlier input values. These elements are essential for the automatic migration of a model into the database and the necessary mappings into the HiveQL world.

Note

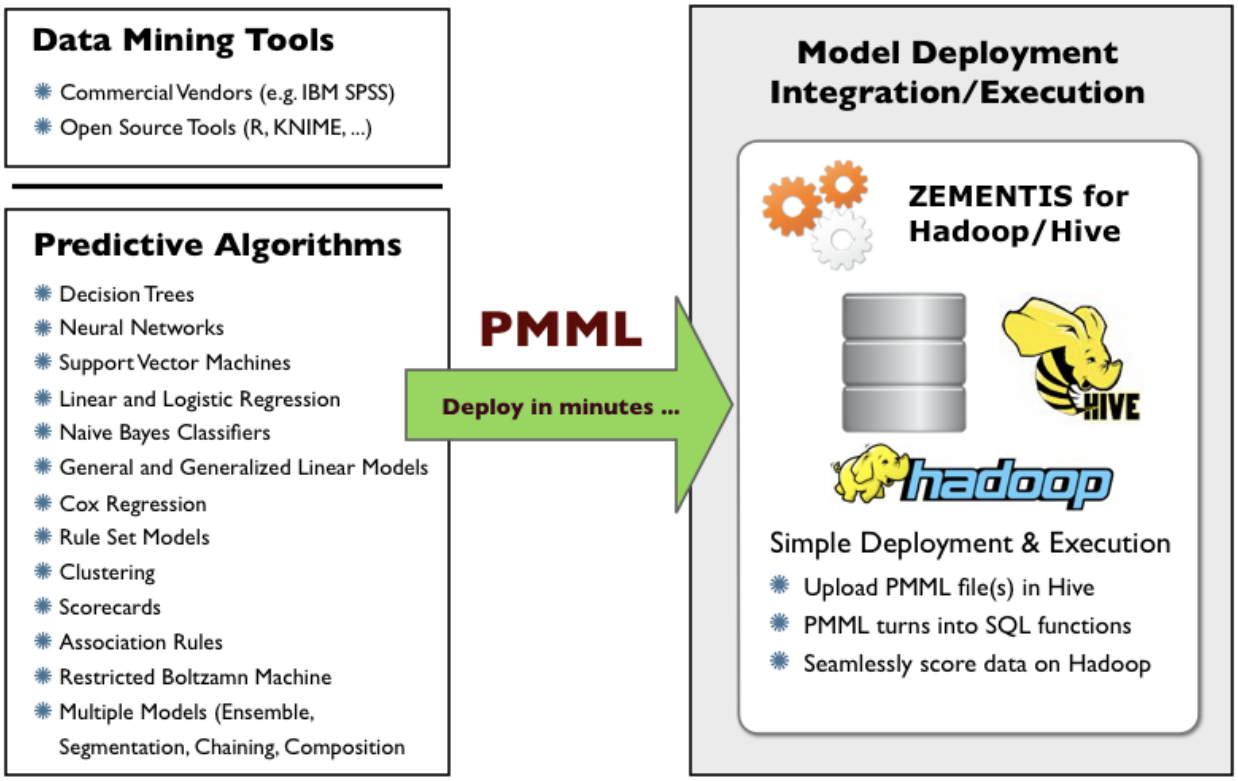
A variety of sample PMML models are included with the ZEMENTIS distribution package. In addition, a wealth of resources on PMML can be found from the PMML in Action.

2.2. ZEMENTIS Predictive Analytics (ZEMENTIS)

ZEMENTIS plugin enables execution of standards-based predictive analytics directly within a database or data warehouse application. It shares the PMML execution core with the ZEMENTIS server offered by Software AG. It is, however, optimized to be embedded within the database environment or data warehouse application in order to minimize data movement.

In addition, the plug-in takes on the responsibility of bridging the PMML and HiveQL world (see Figure 2.1). This means that it presents each loaded PMML model as a HiveQL function. The name, input parameters and outputs of each function matches the name, input fields, and output fields of the corresponding model as defined in the PMML file. This way, scoring a data set against one or more models requires nothing more than writing a HiveQL statement that invokes the HiveQL functions for the corresponding models. Predictions (scores, probabilities, categories, clusters, etc.) can be just as easily written back to the database, become part of a report, or passed on to an application.

Figure 2.1. Overview of ZEMENTIS for Hive



At a high level, the process of using PMML models in Hive starts after the predictive models have been created and exported in PMML format from a data mining tool. With the PMML files in hand, it only takes two simple steps to import a predictive model in Hive:

1. *Preparation Step*: Validates the PMML files and prepares HiveQL scripts with definitions for the new HiveQL functions.
2. *Registration Step*: Registers generated HiveQL functions with Hive

These steps are described in more detail in Chapter 4 and illustrated with actual examples in Chapter 5.

After preparation and registration steps are completed, data on a Hadoop cluster can be processed to generate predictions by invoking HiveQL functions from Hive queries.

Like the ZEMENTIS server, ZEMENTIS plugin accepts PMML models of all versions (2.0, 2.1, 3.0, 3.1, 3.2, 4.0, 4.1, 4.2 and 4.3) generated by any of the major commercial and open source data mining tools.

ZEMENTIS supports a wide range of predictive analytics techniques, including:

- Decision Trees for classification and regression
- Neural Network Models: Back-Propagation, Radial-Basis Function, and Neural-Gas
- Support Vector Machines for regression, binary and multi-class classification
- Linear and Logistic Regression (binary and multinomial)
- Naïve Bayes Classifiers
- General and Generalized Linear Models
- Cox Regression Models
- Rule Set Models (flat decision trees)
- Clustering Models: Distribution-Based, Center-Based, and 2-Step Clustering
- Scorecards (including reason codes and point allocation for complex attributes)
- Segmented Models
- Model Ensembles (including Random Forest Models)
- Model Composition and Chaining

In addition, ZEMENTIS also implements a wide range of functions for data pre- and post-processing, including:

- Value Mapping
- Discretization

- Normalization
- Scaling
- Conditional Logic
- Logical and Arithmetic Operators
- Built-in Functions
- Business Decisions and Thresholds

Note

- ZEMENTIS for Hive does not support Association Rules models.
- ZEMENTIS for Hive does not support BlockIndicator elements, which might be used as part of Lag expressions.

Chapter 3. Installation

This chapter describes how to install ZEMENTIS for Hive.

3.1. Requirements

The requirements to install ZEMENTIS for Hive on your system are:

Table 3.1. The ZEMENTIS Installation Requirements

Requirement	Version	Notes
Hive	1.2.1 or above	The rest of this documentation assumes that Hive is already installed. Please see Hive documentation for details.
Java Development Kit	8 or above	Please make sure you use the Java Development Kit (JDK) and not the Java Runtime Environment (JRE).
ZEMENTIS for Hive License Key	10.4.0.0	Installing new PMML models with ZEMENTIS for Hive requires a valid Product License Key which can be obtained by contacting Software AG. Place the Product License Key file (named zementis.license) in the directory from which prepare-pmml.sh/prepare-pmml.bat script is executed. More information about prepare-pmml.sh/prepare-pmml.bat can be found in Section 4.3.1. Please note that execution of existing models will not be interrupted when the license expires.

3.2. Packaging

ZEMENTIS for Hive is distributed as a compressed archive file : `uppi-hive-10.4.0.0.zip`. The distribution package consists of several files, including this documentation and several sample files. When uncompressed, the package reveals a number of sub-directories as described in Table 3.2.

Table 3.2. Directory Structure of the ZEMENTIS for Hive package

Directory	Contents
`bin`	Contains the `prepare-pmml.sh` and `prepare-pmml.bat` scripts needed for generating the HiveQL functions from PMML files (see Section 4.3.1).
`docs`	Documentation in HTML and PDF format.

Directory	Contents
`lib`	The required library (JAR) file for installing ZEMENTIS for Hive (see Section 3.3).
`pmml`	Sample PMML files along with corresponding data files in CSV format and text files with step-by-step PMML model deployment instructions. These samples include the examples described in Chapter 5.

3.3. Installation

Once Hive is installed and configured, installing ZEMENTIS for Hive is straightforward.

First, library JAR files (in the `lib` sub-directory of the ZEMENTIS package) should be copied to the same file system where Hive is installed.

Then, Hive should be made aware of the location of these files. More specifically:

1. Assuming *`HIVE_DIR`* represents the directory where Hive is installed and *`UPPI_DIR`* represents the directory where the ZEMENTIS library files are installed, update the configuration file *`HIVE_DIR`*`/conf/hive-env.sh` to set the variable `HIVE_AUX_JARS_PATH` as follows:

```
export HIVE_AUX_JARS_PATH=/UPPI_DIR/lib/uppi-hive-10.4.0.0.jar
```

2. Note that if other JAR files are also required for the Hive installation, the `HIVE_AUX_JARS_PATH` environment variable can be updated as shown in the example below:

```
export HIVE_AUX_JARS_PATH=/otherpath/other.jar,/UPPI_DIR/lib/uppi-hive-10.4.0.0.jar
```

3. Optionally, in the same configuration file (*`HIVE_DIR`*`/conf/hive-env.sh`) you may want to set the variable `HADOOP_HEAPSIZE` to modify the maximum memory allocated for Hive tasks. For example, to allow allocating up to 1GB of memory, use the following setting:

```
export HADOOP_HEAPSIZE=1024
```

   Note that most PMML models work fine with the default maximum memory setting. However, additional memory may be required to accommodate some very large PMML files (e.g. large random forest models).

Once ZEMENTIS for Hive is installed, you can start creating HiveQL functions from PMML models. This process is described in the next chapter.

Chapter 4. From PMML to HiveQL

This chapter describes in detail how PMML models are made available to be used directly in Hive. As explained earlier, ZEMENTIS for Hive converts predictive models into HiveQL functions by mapping the model to a user-defined function (UDF). In the following sections we present the actual steps to embed the PMML models into Hive.

4.1. Model as a HiveQL Function

With ZEMENTIS for Hive, a predictive model is converted into an user-defined function (UDF) that can be used like any other built-in function. This way a model can be easily applied to rows of a Hive table or to the result of another query.

Every PMML model has a name, a number of input fields, and one or more output fields. Each input or output field has a name and a data type (`string`,`integer`,`float`, etc.). When translated into a HiveQL function, this information is used to derive the name, input, and return parameters of the function. More specifically, the name of the function comes from the name of the model, possibly altered to comply with the naming conventions or limitations of Hive. The input fields of the model, i.e., the active and supplementary mining fields, become the input parameters of the function. The input parameters are ordered based on the order of the fields in the mining schema of the PMML model. The name and type of each input parameter is derived from the name and type of the corresponding mining field. Table 4.1 presents which Hive data type corresponds to each of the PMML data types.

Similarly, the output fields of the model become the return values of the UDF. While many PMML models have only one output field (typically the predicted value), it is also very common that a model has more than one output field. For example, this is often the case with classification models where the probabilities of different classes are output along with the predicted class. To accommodate for the more generic cases, the UDF's generated by ZEMENTIS return `STRUCT` complex types. A `STRUCT` type is composed of one or more elements, each with a name and a type. For details on these complex types, please visit the Complex Types section of the Hive tutorial.

Consider, for example, a classification model trained with the Iris data set. The Iris dataset [1] is perhaps the best known data set to be found in the pattern recognition literature. It contains three classes representing different types of the Iris plant. Each class is represented by 50 records containing the sepal and petal lengths and widths of different plants. Such a model may produce four output values, the class or type of Iris plant (setosa, virginica, or versicolor) along with the computed probabilities for the three different flower types. With ZEMENTIS, the return type for the UDF created for this model would look like the following:

[1] For more information on the Iris data set, please refer to: Asuncion, A. & Newman, D.J. (2007). UCI Machine Learning Repository. Irvine, CA: University of California, School of Information and Computer Science.

```
STRUCT {class STRING; prob_setosa DOUBLE; prob_virginica DOUBLE; prob_versicolor DOUBLE}
```

The use of HiveQL functions as well as the complex types returned by ZEMENTIS will become evident with the examples that follow in Chapter 5.

4.2. Data Types

The table below presents how PMML data types are mapped to Hive data types. For more information on the PMML data types, please visit the PMML Data Dictionary page.

Table 4.1. PMML and HiveQL Data Types

PMML Types	Hive Types
`string`	`STRING`
`integer`	`INT`
`float`	`FLOAT`
`double`	`DOUBLE`
`double`	`DECIMAL` [a]
`boolean`	`BOOLEAN`
`date`	`TIMESTAMP`
`time`	`TIMESTAMP`
`dateTime`	`TIMESTAMP`
`binary` [b]	`BINARY`

[a] `DECIMAL` type data will be processed as `DOUBLE` type data base on `dataType` definition of PMML.

[b] It is required to set `BINARY_BUFFERED` as `true` in `<Extension>` element. Please see Section 7.4 for details.

4.3. How To: Two Easy Steps

In this section, we describe the two steps that need to be taken to use PMML models in Hive. The first step is to prepare the HiveQL functions from PMML files. The second step is to install the HiveQL functions in Hive. Both steps are described in the following sub-sections where we assume that:

- Hive and ZEMENTIS for Hive are installed and configured, as described in Section 3.3.

- One or more PMML files have been created and are available under a directory on your system. If preferred, different PMML files can be nested within sub-directories as long as they all are under a common parent directory.

4.3.1. Prepare HiveQL Functions (Preparation Step)

The first part of the process is a preparation step for creating the HiveQL functions out of the available predictive models. The ZEMENTIS package contains the script `prepare-pmml.sh` (`prepare-pmml.bat` is also available for Windows environment) which performs all the necessary actions for this step. In short, this script will first validate the PMML models and then generate the files required for installing and using the models in Hive.

Note

The preparation step does not need to be performed in a system where Hive is installed and configured. However, it does require one of the library files contained in the Hive distribution as described below.

To configure a system to run the script `prepare-pmml.sh`, you need to:

- Copy and unpack the ZEMENTIS package, `uppi-hive-10.4.0.0.zip`, to your working directory.
- Set the environment variable *`JAVA_HOME`* to point to the installation directory of the Java Development Kit (JDK).
- From a Hive distribution package or installation, obtain a copy of the file `hive-exec-X.Y.Z.jar`, where `X.Y.Z` reflects the version of Hive you are using. For example, if you are using Hive version 1.2;, you need to obtain the a copy of the file `hive-exec-1.2.jar`. This file is usually located in the `lib` directory of the Hive distribution.
- Set the environment variable *`HIVE_EXEC_JAR`* to point to the library file `hive-exec-X.Y.Z.jar`.

Once your system is configured, you may run the preparation script by using the following command:

```
UPPI_DIR/bin/prepare-pmml.sh PMML_DIR
```

where *`UPPI_DIR`* refers to the directory where ZEMENTIS has been installed and *`PMML_DIR`* refers to the (top) directory where the PMML files are located. Additional Command Line Interface (CLI) options for `prepare-pmml.sh` script are listed in Table 4.2. Similar options are also available for the `prepare-pmml.bat` script.

Table 4.2. `prepare-pmml.sh` script options

CLI Option Flag	Value Type	Default Value	Required	Description
-pmml	*path*	""	Yes	Specify path to PMML file(s) top directory. The **-pmml** flag can be omitted only if the *path* is placed as first argument.
-extLib	*path*	""	No	Specify path to custom functions JAR file(s) directory. See Chapter 7 regarding custom PMML functions.
-out	*path*	`uppi-output`	No	Specify path to generated output files directory.
-abortOnError	`true` or `false`	`true`	No	We recommended that the treatment of invalid values should be handled in the PMML as described in Chapter 6. If the invalid value treatement for a `MiningField` is set to `returnInvalid`, the Hive query will abort when an invalid value is encountered. Setting the **-abortOnError** flag to `true` will have the same effect. Setting it to `false` will enable the query processing to continue by returning null values for the invalid ones. A value of `false` for this option should only be used for debugging/testing purposes. It is important to note that the **-abortOnError** flag applies to all `MiningFields`.
-awsPipeline	`true` or `false`	`false`	No	Generate AWS Data Pipeline script if `true`. Find out how to use ZEMENTIS for Hive on AWS EMR Cluster in Chapter 8.
-runWithPmml	`true` or `false`	`false`	No	If `true`, include PMML file in generated UDF JAR instead of binary representation of the PMML model. This option should be used only for debugging or diagnostic purpose.
-applyCleanser	`true` or `false`	`true`	No	If `true`, comprehensive syntactic and semantic checks and corrections are applied on the provided PMML files. If `false` the PMML files are processed as is.

While running, the script will generate a series of messages marked as `INFO`, `WARNING`, or `SEVERE`. Messages marked as `INFO` provide information on the files processed and the progress of the script. Messages of type `WARNING` indicate potential issues with a PMML file that may need to be reviewed. The detailed warning messages are provided in a copy of the original PMML file, annotated with comments at the appropriate locations. Note that the annotated version of the file may look different than the original, as the file may have been upgraded to the latest version of PMML. The corresponding model is fully functional and, more often than not, these warnings are not relevant to the scoring process. However, a review of these messages is highly recommended since, in some cases, they may have an impact on scoring. Finally, messages marked as `SEVERE` indicate that a valid model cannot be created from the provided file. As in the case of warnings, the detailed messages are provided in an appropriately annotated copy of the original file. In the case of `SEVERE` messages, the problems identified need to be corrected before the model can be used.

Once the preparation script has completed successfully, you will find a directory named `uppi-output` created under your current working directory. The contents of that directory are described in Table 4.3.

Table 4.3. Output generated from the `prepare-pmml.sh` script

File or Directory	Description
`MODEL_NAME``.jar`	A JAR file for each valid model contained in the PMML file(s) as well as generated Java code that serves as wrapper to these PMML file(s). The `MODEL_NAME` is derived from the name of the PMML model.
`MODEL_NAME.sql`	A HiveQL script for each valid model contained in the PMML file(s). Typically, this file will contain a `CREATE` statement for registering each model as an UDF and a corresponding `SELECT` statement which acts as a template for scoring data against that model via UDF. The `MODEL_NAME` is derived from the name of the PMML model.
`pmml`	A directory containing copies of the processed PMML files for which severe or warning messages were generated. The files get annotated with comments that contain the relevant messages. Please note that the annotated PMML files may be different than the original ones since they are upgraded to the latest PMML version (version 4.3) and known issues are corrected.
`java`	A directory containing the generated Java code. This is the code that gets compiled into the *`MODEL_NAME`*`.jar` file.
`classes`	A directory containing the compiled Java code, along with the original PMML files. This reflects the contents of the *`MODEL_NAME`*`.jar` file.

Out of all these files, only the *`MODEL_NAME.sql`* and *`MODEL_NAME`*`.jar` files are used in the next (and last) step. The rest of the files generated during the execution of the script remain in place for diagnostic purposes only.

4.3.2. Register HiveQL Functions (Registration Step)

To register HiveQL functions for the PMML files, all you need to do is execute *MODEL_NAME*.sql script using following Hive HiveQL command:

```
source uppi-output/MODEL_NAME.sql;
```

where uppi-output is path to the output directory generated by the execution of the prepare-pmml.sh script, and *MODEL_NAME*.sql is one of generated files for a PMML model.

Important

ZEMENTIS can work on both local file system and HDFS.

To work on HDFS:

- Make sure that all the files are in the HDFS.
- Make sure that the location of the *MODEL_NAME*.jar file mentioned in the *MODEL_NAME*.sql file is pointing to the appropriate location in HDFS.
- Example: hdfs://namenode/path/to/the/jar;

Chapter 5. Examples

This chapter contains examples of using ZEMENTIS for Hive.

5.1. Sample Files

The ZEMENTIS for Hive package contains a number of sample PMML files, each with a CSV (Comma Separate Values) file containing test data, and a "README" text file showing all steps necessary to successfully execute the PMML model in Hive. The test data contains both input and output values. The output values are provided to allow for the validation of the results generated by ZEMENTIS for Hive. To run each example, the test data needs to be loaded into Hive tables. For the examples presented below, we describe the process of creating the necessary test tables and loading the data into them. For the sample PMML files, these instructions are also available in the "README" files corresponding to each sample model.

5.2. Model with Single Output

This section provides an example of a model with a single output field. The example uses a Neural Network model created for the El Nino data set[1]. This model is provided in the file `ElNino_NN.pmml` among the sample files in the `pmml` directory of the ZEMENTIS package. Along with it, there is a test data file `ElNino_NN.csv`.

The following listing contains a fragment of the PMML model that contains the model name (attribute `modelName` in the `NeuralNetwork` element) as well as the input and output fields (`MiningField` elements).

```
<DataDictionary numberOfFields="7">
    <DataField dataType="double" name="airtemp" optype="continuous"/>
    <DataField dataType="double" name="latitude" optype="continuous"/>
    <DataField dataType="double" name="longitude" optype="continuous"/>
    <DataField dataType="double" name="zon_winds" optype="continuous"/>
    <DataField dataType="double" name="mer_winds" optype="continuous"/>
    <DataField dataType="double" name="humidity" optype="continuous"/>
    <DataField dataType="double" name="s_s_temp" optype="continuous"/>
</DataDictionary>
...
<NeuralNetwork activationFunction="tanh" functionName="regression" modelName="ElNino_NN">
    <MiningSchema>
        <MiningField name="latitude"/>
        <MiningField name="longitude"/>
        <MiningField name="zon_winds"/>
        <MiningField name="mer_winds"/>
        <MiningField name="humidity"/>
        <MiningField name="s_s_temp"/>
        <MiningField name="airtemp" usageType="predicted"/>
    </MiningSchema>
```

[1]For more information on the El Nino data set, please refer to: Asuncion, A. & Newman, D.J. (2007). UCI Machine Learning Repository. Irvine, CA: University of California, School of Information and Computer Science.

```
    ...
```

The input of the model consists of location and weather data presented as numeric values (fields `latitude`, `longitude`, `zone_winds`, `mer_winds`, `humitidy`, and `s_s_temp`). It returns a single numeric value, the predicted air temperature (predicted field `airtemp`).

To test this sample file, execute the preparation and registration steps described in Section 4.3.1 and Section 4.3.2. As multiple sample PMML files are provided under the same directory, you may limit the number of files processed in the preparation step by specifying the subdirectory containing only the neural network models:

```
UPPI_DIR/bin/prepare-pmml.sh UPPI_DIR/pmml/NeuralNetwork
```

Note that the above command will prepare all models in the `NeuralNetwork` subdirectory, including `ElNino_NN.pmml`. The generated `ElNino_NN.sql` file will contain a statements for creating HiveQL function for the `ElNino_NN` model. Execute this script with following HiveQL command:

```
source uppi-output/ElNino_NN.sql;
```

After HiveQL function is registered in Hive (see Section 4.3.2), you may review its description. For example, you can run the following command to review the description of function `elnino_nn`:

```
describe function elnino_nn;
```

This command will output the name of the function along with the names and types of the input and output parameters:

```
elnino_nn(latitude DOUBLE, longitude DOUBLE, zon_winds DOUBLE, mer_winds DOUBLE, humidity DOUBLE,
 s_s_temp DOUBLE) -> STRUCT {airtemp DOUBLE}
```

Note that the generated function reflects closely the information found in the PMML file. The name of the function is derived from the name of the model (`ElNino_NN`). The function has six parameters of type `DOUBLE`, matching the name, order and types of the model input fields. The return type of the function is a `STRUCT` type, with only one element, `airtemp` of type `DOUBLE`, which matches the name and type of the predicted field of the model.

With the function installed in Hive, using the model requires nothing more than invoking it in a query. However, before we show an example of a query, we need to create a table and load the sample data into it.

The following command creates a table for the data in file `ElNino_NN.csv`:

```
CREATE TABLE elnino_nn_data(
    row_id INT,
    latitude DOUBLE,
    longitude DOUBLE,
```

```
    zon_winds DOUBLE,
    mer_winds DOUBLE,
    humidity DOUBLE,
    s_s_temp DOUBLE,
    airtemp DOUBLE
)ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
```

Table column "airtemp" contains expected output for this model, and can be used to verify the result from function.

The following command loads the data into the Hive table:

```
LOAD DATA LOCAL INPATH 'ElNino_NN.csv' INTO TABLE elnino_nn_data;
```

Note that the first line of this CSV file is the header line, containing the names of the data columns and no actual values. If the file is loaded as is, the header line will create a row containing `NULL` values since names cannot be stored in the table columns, which are of type `DOUBLE`. For this reason, we recommend that you delete the header line of the CSV file before loading it into Hive.

After the data is loaded, we can then execute queries that score the data in the table `elnino_nn_data` using the function `elnino_nn`. The first example below, is the simplest query that invokes the function `elnino_nn`. Note that the output for each row is a single element `STRUCT` per the function's description:

```
SELECT elnino_nn(latitude,longitude,zon_winds,mer_winds,humidity,s_s_temp)
FROM elnino_nn_data LIMIT 10;

{"airtemp":27.127614295247312}
{"airtemp":27.023705228506447}
{"airtemp":27.183023776611908}
{"airtemp":27.1786879512385}
{"airtemp":27.31386469460455}
{"airtemp":27.4043135759579}
{"airtemp":27.400061796152368}
{"airtemp":27.39820876609351}
{"airtemp":27.517896498091755}
{"airtemp":27.358719891352678}
```

To return the actual predicted value, i.e. the value of the `airtemp` element as opposed to the whole complex type, we can modify the query as follows:

```
SELECT row_id, elnino_nn(latitude,longitude,zon_winds,mer_winds,humidity,s_s_temp).airtemp
FROM elnino_nn_data
ORDER BY row_id;

     row_id       airtemp
------------- ------------------
        1      27.127614295247312
        2      27.023705228506447
        3      27.183023776611908
        4      27.1786879512385
        5      27.313864694604554
        6      27.4043135759579
        7      27.400061796152368
        8      27.39820876609351
        ...
```

As a last example, we can create a query that outputs the values computed by the function in ZEMENTIS along with the expected predictions obtained from the tool initially used to build the model. This is done to make sure that the model has been deployed correctly and that the scores computed by ZEMENTIS match the expected scores generated by the model development environment. Note that the provided CSV file and, consequently, the created sample table contain a column named `airtemp`. The contents of that column contain the expected values for each row as produced by the tool used to build the model. The example below is a simple score-matching query that allows for comparing the ZEMENTIS computed values to the expected ones.

```
SELECT row_id, elnino_nn(latitude,longitude,zon_winds,mer_winds,humidity,s_s_temp).airtemp AS predicted,
 airtemp AS expected
FROM elnino_nn_data
ORDER BY row_id;

 row_id        predicted            expected
 -------    ------------------    ---------------
   1        27.127614295247312     27.1276143
   2        27.023705228506447     27.02370523
   3        27.183023776611908     27.18302378
   4        27.1786879512385        27.17868795
   5        27.31386469460455       27.31386469
   6        27.4043135759579        27.40431358
   7        27.400061796152368      27.4000618
   8        27.39820876609351       27.39820877
```

In the above query, using `row_id` allows us to correlate input with the output.

Running the `prepare-pmml.sh` script will generate a `SELECT` script for the UDF corresponding to the provided model(s). These `SELECT` statements are commented and they should be modified to suit your HiveQL needs.

```
-- Select script for Scoring ElNino_NN.pmml
-- Please change table names and column names in the following query based on your requirements.
-- CREATE TABLE ElNino_NN_Scalar_Out
--     AS SELECT row_identifier, elnino_nn(latitude, longitude, zon_winds, mer_winds, humidity,
 s_s_temp).predictedvalue_airtemp
--         FROM ElNino_NN ORDER BY row_identifier
```

5.3. Model with Multiple Outputs

Many predictive models are built to produce more than one output. This is often the case for classification models which output the winning class along with the predicted probabilities for one or more of the classes, or clustering models which output the winning cluster along with the affinity for that cluster.

An example of such a model is the Decision Tree model built for the Iris data set. This model is included in the provided samples (look for the file `Iris_CT.pmml` among the sample files in the `pmml` directory of the ZEMENTIS package). It is a classification model that, given the sepal and petal lengths and widths of an Iris plant, predicts the most likely species the plant belongs to (one of `Iris-setosa`, `Iris-versicolor`, or `Iris-virginica`) along with the predicted probability for each of the species.

The following listing presents the input and output fields of the model, as listed in the PMML file. The input fields are the `MiningField` elements from the `MiningSchema` section with the attribute `usageType="active"`. These are `petal_length`, `petal_width`, `sepal_length`, and `sepal_width`. The output fields are listed as `OutputField` elements. They are `class`, `Probability_setosa`, `Probability_versicolor`, and `Probability_virginica`. The first field outputs the predicted (winning) species and the other three the predicted probabilities for each of the species.

```
<DataDictionary numberOfFields="5">
    <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"/>
    <DataField dataType="string" name="target_class" optype="categorical">
      <Value property="valid" value="Iris-setosa"/>
      <Value property="valid" value="Iris-versicolor"/>
      <Value property="valid" value="Iris-virginica"/>
    </DataField>
  </DataDictionary>
  <TreeModel algorithmName="CART" functionName="classification" modelName="Iris_CT">
    <MiningSchema>
      <MiningField name="petal_length" usageType="active"/>
      <MiningField name="petal_width" usageType="active"/>
      <MiningField name="sepal_length" usageType="active"/>
      <MiningField name="sepal_width" usageType="active"/>
      <MiningField name="target_class" usageType="predicted"/>
    </MiningSchema>
    <Output>
      <OutputField dataType="string" feature="predictedValue" name="class" optype="categorical" />
      <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>
    ...
```

As in the previous example, to test this sample file, first execute the preparation step described in Section 4.3.1. This time, in order to reduce the number of files processed limit the preparation step to the sub-directory containing only decision tree models:

```
UPPI_DIR/bin/prepare-pmml.sh UPPI_DIR/pmml/TreeModel
```

After the preparation step, it is time to perform the registration step as described in Section 4.3.1. Note that the above command will prepare all models in the `TreeModel` sub-directory, including `Iris_CT.pmml`. The generated `Iris_CT.sql` file will contain HiveQL statement for registering the function for the `Iris_CT` model:

```
source uppi-output/ElNino_NN.sql;
```

Once HiveQL function is registered, you can use the following command to review its description:

```
describe function iris_ct;
```

This command will output the name of the function along with the names and types of the input and output parameters:

```
iris_ct(petal_length DOUBLE, petal_width DOUBLE, sepal_length DOUBLE, sepal_width DOUBLE) -> STRUCT
 {class STRING, probability_setosa DOUBLE, probability_versicolor DOUBLE, probability_virginica DOUBLE}
```

Note that this output shows the function with the four input parameters of type `DOUBLE` which correspond to the four input fields of the model and the complex `STRUCT` return type containing four elements corresponds to the four output fields of the model.

Before describing example queries for this function, we need to load the sample data into Hive from the file `Iris_CT.csv`. First, we create a table using the command:

```
CREATE TABLE iris_ct_data(
    row_id INT,
    petal_length DOUBLE,
    petal_width DOUBLE,
    sepal_length DOUBLE,
    sepal_width DOUBLE,
    class STRING,
    probability_setosa DOUBLE,
    probability_versicolor DOUBLE,
    probability_virginica DOUBLE
)ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
```

Table columns "class","Probability_setosa","Probability_versicolor","Probability_virginica" contain expected output for this model, and can be used to verify the result from function.

We can load the data from the file `Iris_CT.csv` using the command below. As in the previous example, we recommend that you remove the header line from the file before loading it into Hive.

```
LOAD DATA LOCAL INPATH 'Iris_CT.csv' INTO TABLE iris_ct_data;
```

With the sample data loaded, we can then run queries that apply the model. The simplest query to score the data in `iris_ct_data` is:

```
SELECT iris_ct(petal_length, petal_width, sepal_length, sepal_width) FROM iris_ct_data LIMIT 10;

{"class":"Iris-setosa","probability_setosa":1.0,"probability_versicolor":0.0,"probability_virginica":0.0}
{"class":"Iris-setosa","probability_setosa":1.0,"probability_versicolor":0.0,"probability_virginica":0.0}
{"class":"Iris-setosa","probability_setosa":1.0,"probability_versicolor":0.0,"probability_virginica":0.0}
{"class":"Iris-setosa","probability_setosa":1.0,"probability_versicolor":0.0,"probability_virginica":0.0}
{"class":"Iris-setosa","probability_setosa":1.0,"probability_versicolor":0.0,"probability_virginica":0.0}
{"class":"Iris-setosa","probability_setosa":1.0,"probability_versicolor":0.0,"probability_virginica":0.0}
{"class":"Iris-setosa","probability_setosa":1.0,"probability_versicolor":0.0,"probability_virginica":0.0}
{"class":"Iris-setosa","probability_setosa":1.0,"probability_versicolor":0.0,"probability_virginica":0.0}
{"class":"Iris-setosa","probability_setosa":1.0,"probability_versicolor":0.0,"probability_virginica":0.0}
{"class":"Iris-setosa","probability_setosa":1.0,"probability_versicolor":0.0,"probability_virginica":0.0}
```

The result of this query is a single column with complex values, as dictated by the return type of the function. Each value contains all four output fields of the model. Any one of these fields can be singled out. In this way, we can

execute a query to return an specific output value as the result. For example, to return only the predicted class (output field `class`) without the probabilities of the different Iris species, we can execute the following query:

```
SELECT row_id, iris_ct(petal_length, petal_width, sepal_length, sepal_width).class
FROM iris_ct_data
ORDER BY row_id;

   row_id        class
----------- ---------------
     1       Iris-setosa
     2       Iris-setosa
     3       Iris-setosa
     4       Iris-setosa
     5       Iris-setosa
     6       Iris-setosa
     7       Iris-setosa
     8       Iris-setosa
     9       Iris-setosa
     10       Iris-setosa
```

For the result to contain the values of more than one field as separate columns, the query becomes a little more involved. More specifically, the function must be called in a nested query which will return the complex structure from the function. From that result, the enclosing query selects a few or all of the elements from the structure and output them into separate columns. The following is an example of such a nested query:

```
SELECT s.row_id,r.class,r.probability_setosa,r.probability_versicolor,r.probability_virginica
FROM( SELECT row_id,iris_ct(petal_length,petal_width,sepal_length,sepal_width) AS r FROM iris_ct_data) s
ORDER BY s.row_id;


  s.row_id      r.class     r.probability_setosa r.probability_versicolor r.probability_virginica
---------- --------------- -------------------- ----------------------- ----------------------
     1       Iris-setosa          1.0                    0.0                    0.0
     2       Iris-setosa          1.0                    0.0                    0.0
     3       Iris-setosa          1.0                    0.0                    0.0
     4       Iris-setosa          1.0                    0.0                    0.0
     5       Iris-setosa          1.0                    0.0                    0.0
     6       Iris-setosa          1.0                    0.0                    0.0
     7       Iris-setosa          1.0                    0.0                    0.0
     8       Iris-setosa          1.0                    0.0                    0.0
     ...
```

In the above query, using `row_id` allows us to correlate input with the output.

Running the `prepare-pmml.sh` script will generate a `SELECT` script for the UDF corresponding to the provided model(s). These `SELECT` statements are commented and they should be modified to suit your HiveQL needs.

```
-- Select script for Scoring Iris_CT.pmml
-- Please change table names and column names in the following query based on your requirements.
-- CREATE TABLE Iris_CT_Table_Out AS
--     SELECT s.row_identifier, r.class, r.probability_setosa, r.probability_versicolor,
 r.probability_virginica
--         FROM (SELECT row_identifier, iris_ct(petal_length, petal_width, sepal_length, sepal_width)
--         AS r FROM Iris_CT) s ORDER BY s.row_identifier;
```

Chapter 6. Handling of Invalid Values

PMML offers a rich set of options for defining the data types of the different input fields as well as the set or range of valid values for each field in the Data Dictionary. Along with those, it allows data scientist to specify what the model should do in the presence of invalid values as specified in the Mining Schema section of the PMML file. The three options for the treatment of invalid values are `returnInvalid`, `asIs`, and `asMissing`. Among these, `returnInvalid` is the most frequently used, since it is the default option in PMML. The option `returnInvalid` instructs the model execution engine not to attempt to apply the model in the presence of an invalid value and, instead, abort with an error. The other two options allow the model to execute by either allowing the invalid value to be processed as is or by treating it as a missing value.

The following listing contains a fragment of the `Iris_CT.pmml` model. The original code was edited to showcase the PMML `MiningSchema` element with and without the explicit use of the attribute `invalidValueTreatment`.

```
...
<MiningSchema>
  <MiningField name="petal_length" usageType="active" invalidValueTreatment="returnInvalid"/>
  <MiningField name="petal_width" usageType="active" invalidValueTreatment="returnInvalid"/>
  <MiningField name="sepal_length" usageType="active"/>
  <MiningField name="sepal_width" usageType="active"/>
  <MiningField name="target_class" usageType="predicted"/>
</MiningSchema>
...
```

Note that although the option for treatment of invalid values is not set for mining fields `sepal_length` and `sepal_width`, the default value for treating invalid values in PMML is `returnInvalid`. In this way, the invalid value treatment for these two fields is the same as the one used for fields `petal_length` and `petal_width` which have PMML attribute `invalidValueTreatment` explictly set to `returnInvalid`.

When used in a database and through queries, the option `returnInvalid` may have a more significant (not-intended) impact. Consider the case where a query is used to apply a model on millions of data records. Also assume that within all these records, there happens to be just a single record with an invalid value for an input field marked with or defaulted to `returnInvalid` invalid value treatment. In this case, the PMML execution engine will generate an error which in turn will cause the whole query to abort with an error. In other words, just a single invalid value among all the input rows may prevent the query from completing.

In some cases, this may be the desired behavior in order to be able to detect invalid values. However, it is often the case that an alternative approach where invalid values do not cause the queries to abort is more desirable. This requires the PMML model to be modified in order to change the invalid value treatment of one or more mining fields from `returnInvalid` (or nothing which is equivalent) to, typically, `asMissing`. With these changes, all invalid input values will be treated as missing values (`NULL`) and the model will be applied to all the input rows, allowing

the query to complete. Please note that, while not always the case, `NULL` input values result in `NULL` output values, indicating that the particular record cannot be processed, without causing the whole query to fail.

The following listing contains the same PMML fragment as shown above, but modified so that the invalid value treatment for all mining fields is `asMissing`.

```
...
<MiningSchema>
  <MiningField name="petal_length" usageType="active" invalidValueTreatment="asMissing"/>
  <MiningField name="petal_width" usageType="active" invalidValueTreatment="asMissing"/>
  <MiningField name="sepal_length" usageType="active" invalidValueTreatment="asMissing"/>
  <MiningField name="sepal_width" usageType="active" invalidValueTreatment="asMissing"/>
  <MiningField name="target_class" usageType="predicted"/>
</MiningSchema>
...
```

Note

It is highly recommended that any such changes to a model are reviewed and approved by the person or team that created the model to ensure that the model is still valid for the assumptions under which it was built.

Chapter 7. Custom PMML Functions

Predictive models may require external resources such as custom functions. ZEMENTIS provides a facility to create and use custom PMML functions. This capability enables, for example, the implementation of intricate calculations that cannot be easily described in PMML, functions that access external systems to retrieve necessary data, or even specialized algorithms not supported by PMML. One class of functions that can be easily implemented using custom functions are aggregations over a period of time or window of transactions. Aggregations are used to obtain, for example, the count, average, maximum and minimum for a set of records. One example is to use custom functions to obtain the average transaction amount for a certain account for the last 30 days.

ZEMENTIS currently supports custom functions written in Java. Once created and made available to ZEMENTIS, custom functions are used the same way as the built-in ones. The steps to achieve this are explained in the following sections.

7.1. Create Custom PMML Functions

Custom functions are implemented as public static methods of Java classes. For a method to be recognized as a custom PMML function, the containing class needs to be annotated with the ZEMENTIS specific `@PMMLFunctions` annotation containing parameter `namespace`. This parameter must specify fully qualified Java class name. Within each annotated class, only methods that are declared as `public static` can be used as PMMLfunctions. In addition, the types of the method parameters as well as its return type must be compatible with the PMML data types. Table 7.1 provides the Java primitive types and classes that correspond to the different PMML data types. The types of the parameters must be either among those listed in the table or among one of their super-classes or super-interfaces (`java.lang.Object`, `java.lang.Comparable`, or `java.lang.Number`). Methods can also declare variable number of parameters (`varargs`). Finally, methods declared as `void` cannot be used as PMML functions.

Caution

Make sure these methods are thread-safe as ZEMENTIS may need to execute these methods concurrently in different threads.

Table 7.1. PMML and Java types in ZEMENTIS

PMML Data Type	Java Primitive Type	Java Class
`boolean`	`boolean`	`java.lang.Boolean`
`date`		`org.joda.time.LocalDate`

PMML Data Type	Java Primitive Type	Java Class
dateTime		org.joda.time.DateTime
double	double	java.lang.Double
float	float	java.lang.Float
integer	long	java.lang.Long
string		java.lang.String
time		org.joda.time.LocalTime
binary (buffered)	byte[]	byte[]

An example of properly declared custom function is shown in Figure 7.1.

Figure 7.1. Custom PMML Function Example

```
package com.company.udf;

import com.zementis.stereotype.PMMLFunctions;

@PMMLFunctions(namespace = "com.company.udf.CustomFunctions")
class CustomFunctions {

    public static Long factorial(Long n) {
        if (n == null) {
            return null;
        } else if (n < 0) {
            throw new IllegalArgumentException();
        } else if (n == 0) {
            return 1;
        } else {
            return n * factorial(n-1);
        }
    }
}
```

In this example, Java class `RecursiveFunctions` has been annotated with `@PMMLFunctions`. This annotation informs ZEMENTIS that the class contains methods which may be used as PMML functions. The value of parameter `namespace "com.company.udf.CustomFunctions"` is the fully qualified class name for `CustomFunctions` class with `com.company.udf` package declaration. The class contains public static method `factorial` with one input parameter of type `Long` and return value of the same type. Both types correspond to PMML `integer` type and declared method is thread safe.

7.2. Use Custom PMML Functions

To make custom functions available to ZEMENTIS, compile the corresponding classes into a JAR file and place it in Hive accessible directory. To compile a class using the `@PMMLFunctions` annotation, include the `up-pi-hive-10.4.0.0.jar` file in Java classpath. This file is included with the ZEMENTIS distribution package.

Custom functions can be used exactly like built-in PMML functions within `Apply` transformations. Within PMML, the namespace is used as a prefix for the name of the custom function and Java method name as postfix. The PMML fragment in Figure 7.2 contains a simple example that uses the function defined in Figure 7.1.

Figure 7.2. Example Using a Custom Function in PMML

```
<DerivedField name="field1" optype="continuous" dataType="integer"/>
<DerivedField name="field2" optype="continuous" dataType="integer">
    <Apply function="com.company.udf.CustomFunctions:factorial">
        <FieldRef field="field1"/>
    </Apply>
</DerivedField>
```

In this example, `field2` of type `integer` is derived by applying custom function `com.company.udf.CustomFunctions:factorial` to derived field `field1` also of type `integer`. The function name is divided by single colon character `:` where name prefix corresponds to the `namespace` parameter of annotation `@PMMLFunctions`, and name postfix corresponds to Java method name `factorial`.

To deploy a PMML model with custom functions, follows steps described in Section 4.3 with following additional steps. To prepare HiveQL functions, run `prepare-pmml.sh` script with additional command line flag `-extLib` and path to custom functions JAR file(s):

```
UPPI_DIR/bin/prepare-pmml.sh PMML_DIR -extLib CUSTOM_FUNCTIONS_DIR
```

where *`UPPI_DIR`* refers to the directory where ZEMENTIS has been installed, *`PMML_DIR`* refers to the (top) directory where the PMML files are located, and *`CUSTOM_FUNCTIONS_DIR`* refers to the (top) directory where custom functions JAR files are located. Other `prepare-pmml.sh` script options are listed in Table 4.2.

Then, register the HiveQL function with Hive by executing *`MODEL_NAME`*`.sql` script as described in Section 4.3.2.

```
source uppi-output/MODEL_NAME.sql;
```

Finally, add the path for each custom function JAR file to the `HIVE_AUX_JARS_PATH` environment variable:

```
export HIVE_AUX_JARS_PATH=$HIVE_AUX_JARS_PATH,CUSTOM_FUNCTIONS_DIR/custom-functions.jar
```

7.3. Non-Deterministic Functions

When processing PMML models, ZEMENTIS performs certain performance optimizations which assume that functions are deterministic, i.e. when presented with the same input values they always return the same result. However, this may not be the case for all functions. For example, the result of a function may depend on the current time and date. Another example might be a call to an external source that retrieves information that is being modified by other systems.

With ZEMENTIS, a custom function may be declared as non-deterministic by annotating the corresponding implementation Java method with the `@NonDeterministicFunction` annotation. Note that this annotation marks a method, and not the containing class. This means a class implementing multiple functions may contain a combination of deterministic and non-deterministic functions.

The following is an example of a non-deterministic function which provides the current time value for a specific a time zone.

Figure 7.3. Custom PMML Function Example

```
package com.company.udf;

import com.zementis.stereotype.PMMLFunctions;
import com.zementis.stereotype.NonDeterministicFunction;
import org.joda.time.DateTime;
import org.joda.time.DateTimeZone;

@PMMLFunctions(namespace = "com.company.udf.CustomFunctions")
class CustomFunctions {

    @NonDeterministicFunction
    public static DateTime dateTimeAtZome(String timeZone) {
        if (timeZone == null) {
            return null;
        }
        return new DateTime(DateTimeZone.forID(timeZone));
     }
}
```

7.4. Binary Sources

Some predictive models use binary data as input for scoring or classifying results. ZEMENTIS supports applying models to binary data by ultilizing an external custom function. Given a proper binary input definition and a custom function deployed in ZEMENTIS, the input binary data can be seamlessly integrated into the scoring/classifying process. Binary data can be retrieved as a `byte[]`. The types of data are listed in Table 7.1. Set `BINARY_BUFFERED` as `true` in `<Extension>` element like the PMML fragment in Figure 7.4 to guarantee the binary data will not be `null` after being consumed.

Figure 7.4. Binary (Buffered) DataType Example

```
<DataDictionary numberOfFields="1">
  <DataField dataType="binary" name="field1" optype="categorical">
    <Extension extender="ADAPA" name="BINARY_FORMAT" value="image/jpeg" />
    <Extension extender="ADAPA" name="BINARY_BUFFERED" value="true" />
  </DataField>
</DataDictionary>
```

Here are the steps to create a corresponding custom function:

- Implement a custom function as a static method of a Java class.
- Annotate it with a ZEMENTIS specific `@PMMLFunctions` annotation.
- Specify the type of the method parameter as `byte[]`.

The custom function can be compatible with the PMML data type of `field1` defined in PMML fragment Figure 7.4. An example of a custom function is shown in Figure 7.5.

Figure 7.5. Custom Function of Buffered Binary Data Example

```
package com.company.udf;

import com.zementis.stereotype.PMMLFunctions;

@PMMLFunctions(namespace = "com.company.udf.CustomFunctions")
class CustomFunctions {

    public static String convert(byte[] byteArray) {
        String convertedString = ... ;
        return convertedString;
    }
}
```

Once the custom function in Figure 7.5 is compiled and deployed, `com.company.udf.CustomFunctions:convert` can be used exactly like a built-in function within `Apply` transformation expression. The PMML fragment in Figure 7.6 contains a simple example that uses the function defined in Figure 7.5.

Figure 7.6. Example Using Custom Function of Buffered Binary Data in PMML

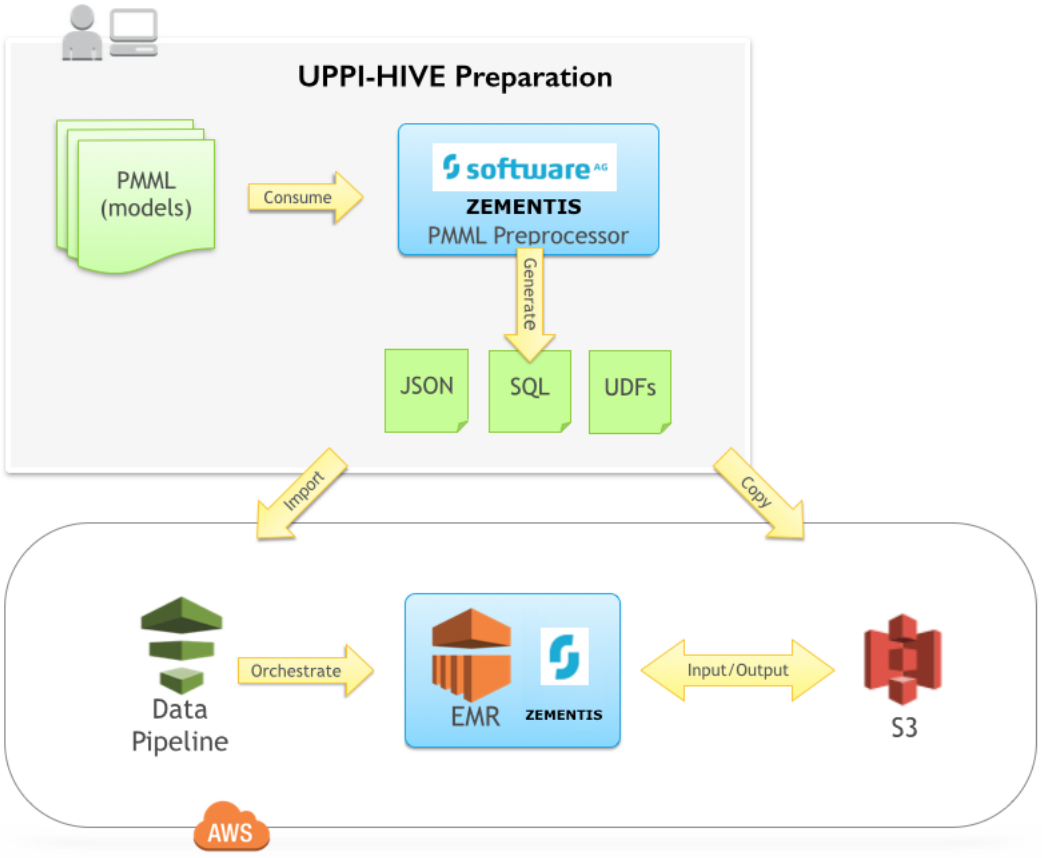
```
<DerivedField name="field2" optype="categorical" dataType="string">
  <Apply function="com.company.udf.CustomFunctions:convert">
    <FieldRef field="field1"/>
  </Apply>
</DerivedField>
```

Chapter 8. Using ZEMENTIS for Hive in AWS EMR Cluster

This chapter describes how to use AWS Data Pipeline to create and orchestrate complex, fault tolerant, and highly available data work-flows by using ZEMENTIS for Hive on AWS EMR Cluster. This enables the user to periodically access data where it is stored, process it at scale by leveraging ZEMENTIS for Hive and efficiently transfer the results to AWS supported data sources.

Creating AWS Data Pipeline for a PMML model is a two-step process. First, a Data Pipeline definition file (represented in JSON format) is created for one or more PMML files using the `prepare-pmml.sh` script. This file needs to be modified to meet your data and computation requirements. As a second step, this modified Data Pipeline definition is then imported into AWS Data Pipeline service. An overview of these steps is illustrated in Figure 8.1 and described in detail in the following sections.

Figure 8.1. Overview of ZEMENTIS for Hive in AWS EMR Cluster



8.1. Prepare AWS Data Pipeline Definition

Using the `awsPipeline` parameter of the `prepare-pmml.sh` script, the ZEMENTIS for Hive package can generate AWS Data Pipeline definition file in JSON format for the corresponding PMML files. The AWS Data Pipeline definition enables the user to create an AWS Data Pipeline service that automates the movement of data and launches AWS EMR Cluster for scoring data against a PMML model. In this section, we describe the process to generate and customize the AWS Data Pipeline definition file with the following steps:

1. Execute the preparation script:

   Running the `prepare-pmml.sh` script with the additional `awsPipeline` parameter generates a JSON file containing the AWS Data Pipeline definition for each provided PMML model.

```
UPPI_DIR/bin/prepare-pmml.sh PMML_DIR -awsPipeline true
```

   Once the preparation script is completed successfully, you will find a directory named `uppi-output` created under your current directory. The contents of that directory are described in Table 8.1.

Table 8.1. Output generated from the `prepare-pmml.sh` script

File or Directory	Description
MODEL_NAME`.jar`	A JAR file for each valid model contained in the PMML file(s) as well as generated Java code that serves as wrapper to these PMML file(s).
MODEL_NAME`.sql`	A HiveQL script for each valid model contained in the PMML file(s). Typically, this file will contain a `CREATE` statement for registering each model as an UDF and a corresponding `SELECT` statement which acts as a template for scoring data against that model via UDF.
MODEL_NAME`.JSON`	A JSON file containing the AWS Data Pipeline definition for each valid model contained in the PMML file(s).
`pmml`	A directory containing copies of the processed PMML files for which severe or warning messages were generated. The files get annotated with comments that contain the relevant messages. Please note that the annotated PMML files may be different than the original ones since they are upgraded to the latest PMML version (version 4.3) and known issues are corrected.
`java`	A directory containing the generated Java code. This is the code that gets compiled into the *MODEL_NAME*`.jar` file.

File or Directory	Description
`classes`	A directory containing the compiled Java code, along with the original PMML files. This reflects the contents of the *`MODEL_NAME`*`.jar` file.

2. Create the following folders in the S3 bucket.
   - Logs Folder
   - HiveQL Scripts Folder
   - UPPI Libs Folder
   - Input Data Folder
   - Output Data Folder
   - Shell Script folder
3. Customize the generated AWS Data Pipeline definition file, *`MODEL_NAME`*`.JSON`:

   All the following entries in the generated *`MODEL_NAME`*`.JSON` need to be replaced with your corresponding paths in the S3 bucket.

   - s3://<<Logs Folder Location>> - Logs folder name on S3

     Example: s3://Bucket-Name/logs-folder-name
   - s3://<<HiveQL Scripts Folder Location>> - HiveQL Scripts folder on S3

     Example: s3://Bucket-Name/HiveQL-scripts-folder-name
   - s3://<<UPPI Libs Folder Location>> - UPPI Libs folder on S3

     Example: s3://Bucket-Name/UPPI-libs-folder-name
   - s3://<<Input Data Folder Location>> - Input Data folder on S3

     Example: s3://Bucket-Name/input-data-folder-name
   - s3://<<Output Data Folder Location>> - Output Data folder on S3

     Example: s3://Bucket-Name/output-data-folder-name
   - s3://<<Shell Scripts Folder Location>> - Shell Scripts Folder on S3

     Example: s3://Bucket-Name/shell-scripts-folder-name

The following entry needs to be replaced with your key pair.

- "keyPair": "<<Your Key Pair Name>>"

Note

The Key pair name can be found in the Key Pairs which is located in the EC2 Management Console under Network and Security.

The generated pipeline definition has a schedule defined with start and end times in UTC time format. The schedule is used for populating data onto the staging area for your AWS EMR Cluster. By default, preparation step populates start and end times with one hour gap. The user has an option to change those based on their scheduling needs.

- "startDateTime": "<<Start Time>>"
- "endDateTime": "<<End Time>>"

4. Customize the generated HiveQL statement in *MODEL_NAME*`.sql`:
   - Before creating a UDF function in Hive, users have to add the corresponding `MODEL_NAME.jar` file in Hive. Please refer to the sample script in the beginning of *MODEL_NAME*`.sql` and modify the path s3://<<HiveQL Scripts Folder Location>> to the directory of your `MODEL_NAME.jar` file.
   - Users can create a new table using `CREATE TABLE` HiveQL statement to store the output of the data processing. If you prefer to have a CSV output in s3://<<Output Data Folder Location>>, the HiveQL statement starting with `INSERT OVERWRITE TABLE ${output1}` needs to be uncommented in the generated *MODEL_NAME*`.sql` file. You will also have to review the data format specified in the *MODEL_NAME*`.sql` and *MODEL_NAME*`.JSON` files. The field names of `TABLE ${output1}` should match the field names of Output_data_format element in *MODEL_NAME*`.JSON` file. For more information, please refer to AWS Data Pipeline Data Format.
5. Add ZEMENTIS for Hive libraries (located in the `lib` directory of the `uppi-hive-10.4.0.0.zip` distribution) to `s3://<<UPPI Libs Folder Location>>`
6. Add *MODEL_NAME*`.sql` and `MODEL_NAME.jar` files to `s3://<<HiveQL Scripts Folder Location>>`
7. Compose a shell script for adding the libraries path in Hive:

```
mkdir /home/hadoop/uppi_lib
touch /home/hadoop/hive/conf/hive-env.sh
echo "export HIVE_AUX_JARS_PATH=/home/hadoop/uppi_lib" >> /home/hadoop/hive/conf/hive-env.sh
```

and put the shell file in `s3://<<Shell Scripts Folder Location>>`

8.2. Create and activate AWS Data Pipeline service

After the steps listed above are done, create a AWS Data Pipeline by importing the modified definition file *`MODEL_NAME`*`.JSON` in the source section. The detailed steps on creating a AWS Data Pipeline service with a *`MODEL_NAME`*`.JSON` file can be found in Creating a Pipeline by Using the AWS Data Pipeline CLI. After a AWS Data Pipeline service is created, the user can activate the AWS Data Pipeline service to launch AWS EMR Cluster for processing data against PMML models.