ZEMENTIS for Spark

User Guide

10.7.0.0

ZEMENTIS for Spark

User Guide

This document applies to ZEMENTIS 10.7.0.0 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 drive 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 decision makers. At the same time, the accelerating growth rate of data collected implies that only the most scalable deployment architectures which can offer robust continuous computation needs will be able to meet realtime analytics requirements.

In the era of big data, more and more organizations are turning to the scalable architecture of Hadoop and Spark to meet this growing challenge. To bring the power of predictive models into this fast and large-scale data processing engine, Software AG has developed the ZEMENTIS Predictive Analytics (ZEMENTIS) for Spark. ZEMENTIS offers Spark 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 a real-time cluster computing system, ZEMENTIS for Spark 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 Spark 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 a Spark application. Practically, PMML becomes a Spark function offering high performance execution 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 Spark. It gives a brief overview of the plug-in, describes each of its components, and explains how these are combined. It then outlines the simple installation process. Finally, it illustrates the use of ZEMENTIS for Spark with a PMML example, a decision tree. The example shows how to deploy and execute predictive models in Spark.

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, missing, or outlier input values. These elements make PMML a great candidate for automatic migration of a model into a complex data processing system like Spark which operates on a core abstraction of applying functions to distributed datasets.

Note

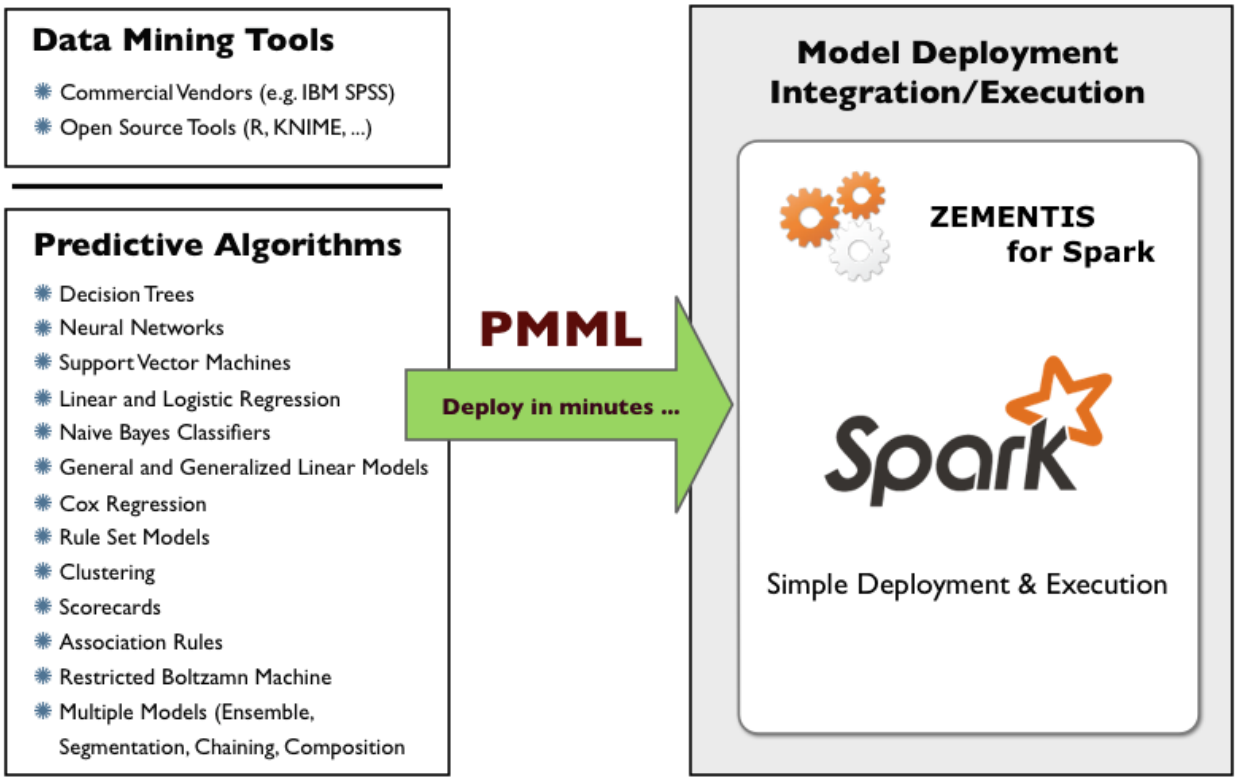
A variety of sample PMML models are included with the ZEMENTIS distribution package. In addition, a wealth of resources on PMML can be found from the PMML in Action.

2.2. ZEMENTIS Predictive Analytics (ZEMENTIS)

ZEMENTIS for Spark enables execution of standards-based predictive analytics directly within a Spark cluster. It shares the PMML execution core with the ZEMENTIS server offered by Software AG. ZEMENTIS for Spark, however, is optimized to seamlessly integrate PMML models into Spark functions, thus enabling standards-based analytics in batch, streaming, and interactive mode on a Spark cluster.

With ZEMENTIS for Spark, a PMML model is programmatically embedded into a Spark function by using the Java API provided with the distribution. Once this is done, "predictions" such as scores, probabilities, categories, and clusters identifiers can then be computed from parallel operations performed on a Resilient Distributed Dataset (RDD). Each of these operations executes a PMML model.

Figure 2.1. Overview of ZEMENTIS for Spark



The process of using PMML models on a Spark cluster starts after the predictive models have been created and exported in PMML format from the data mining tool. Once the PMML files are ready, they can be used directly within a Spark application via the ZEMENTIS Java API. Alternatively, the PMML files can be validated and converted into a binary format by using the `prepare-pmml.sh` script available with the distribution. This way, comprehensive syntactic and semantic checks and corrections can be applied on the PMML files before they are introduced into a production system.

These steps are described in more detail in Chapter 5 and illustrated with actual examples in Section 5.2.

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.

The plug-in supports a wide range of predictive analytics techniques, including:

- Decision Trees for classification and regression
- K-Nearest Neighbors for regression, classification and clustering
- 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

Note

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

Chapter 3. Using PMML in Spark

This chapter describes how PMML models are made available as Spark functions to be passed in the driver program to run on a Spark cluster.

3.1. PMML Model as a Spark Function

Resilient Distributed Dataset (RDD) is the fundamental abstraction in Spark which represents a fault-tolerant collection of elements that can be operated on in parallel. RDDs support two types of operations: `transformations`, which create a new dataset from an existing one, and `actions`, which return a value to the driver program after running a computation on the dataset.

With ZEMENTIS for Spark, a predictive model is converted into a Java implementation of a Spark function which can be used within RDD `transformations`, specifically `map` and `mapPartitions`. The `map` transformation passes each dataset element through a Spark function and returns a new RDD representing the results. With ZEMENTIS for Spark, the results are the output generated by the PMML model. The `mapPartitions` transformation is similar to the `map` transformation, but runs separately on each partition (block) of RDD. Once a PMML model is represented as a Spark function, it can operate on key-value based RDD, where the key refers to the active and supplementary mining field names of the model as defined in the PMML. The output RDDs generated by these transformations contain values keyed by the output field names as defined in the PMML.

Note

Please refer to Spark Programming Guide for more information on programming with RDDs and Spark functions. Working examples are described in detail in Section 5.2.

3.2. PMML and Java Data Types

The table below shows how PMML data types are mapped to Java types. For more information on the PMML data types, you can visit the Data Dictionary page.

Table 3.1. PMML and Java Data Types

PMML Types	Java Types
`string`	`java.lang.String`
`integer`	`long, java.lang.Long`
`float`	`float, java.lang.Float`

PMML Types	Java Types
double	double, java.lang.Double
boolean	boolean, java.lang.Boolean
date	org.joda.time.LocalDate
time	org.joda.time.LocalTime
dateTime	org.joda.time.DateTime
binary (buffered)	byte[]

3.3. 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 scientists 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 show case 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 the 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` explicitly set to `returnInvalid`.

When used within a Spark application, the option `returnInvalid` may have a significant (unintended) impact. Consider the case where an RDD with an invalid value for an input field marked with or defaulted to `returnInvalid`

treatment is applied to a model. In this case, the PMML execution engine will generate an error (exception) which in turn can cause the Spark job to abort due to a task failure if the exception is not handled properly.

In some cases, this may be the desired behavior in order to be able to detect invalid values. However, often it is more desirable to consider an approach in which invalid values do not cause the Spark function to fail. 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 not generate errors on encountering invalid values. Please note that, while not always the case, `NULL` input values result in `NULL` output values, indicating that the particular records cannot be scored.

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 4. ZEMENTIS Installation

This chapter describes how to install ZEMENTIS for Spark.

4.1. Requirements

The requirements to install ZEMENTIS for Spark in your system are:

Table 4.1. The ZEMENTIS Installation Requirements

Requirement	Version	Notes
Spark	2.3.0 or above	The rest of this documentation assumes that Spark is already installed. Please see the Spark Documentation for details.
Java Platform, Standard Edition (Java SE)	8 or above	Please make sure you use the Java Development Kit (JDK) and not the Java Runtime Environment (JRE).
ZEMENTIS for Spark License Key	10.7.0.0	Installation of new PMML models with ZEMENTIS for Spark requires a valid Product License Key which can be obtained by contacting Software AG. Please note that execution of existing models will not be interrupted when the license expires.

4.2. Packaging

ZEMENTIS for Spark is distributed as a compressed archive file: `uppi-spark-10.7.0.0.zip`. The distributed package consists of several files, including this documentation and several sample files. When uncompressed, the package reveals a number of directories as described in Table 4.2.

Table 4.2. Directory Structure of the ZEMENTIS for Spark package

Directory	Contents
`apidocs`	Contains the Java API documentation for using ZEMENTIS as a stand-alone library.
`bin`	Contains the `prepare-pmml.sh` script which takes as one or more PMML files as an input and generates a corresponding binary representation for each model. For detailed usage instructions about the `prepare-pmml.sh` script, please refer to the Java API documentation located in the `apidocs` folder of the distribution.
`docs`	Documentation in HTML and PDF format.

Directory	Contents
`lib`	The library (JAR) files required for the execution of ZEMENTIS for Spark.
`pmml`	A number of sample PMML files along with data files in CSV format. These include the examples described in Section 5.2.
`pmml.sh`	A script file which runs the sample code (contained in the `src` folder) illustrating how ZEMENTIS API can be integrated into Spark function.
`src`	The sample code which embeds each PMML file contained in the `pmml` folder into a Spark function and processes the data from the corresponding CSV file against it. The code is described in detailed in Section 5.2.

4.3. Installation

To install ZEMENTIS for Spark, simply uncompress the provided file (`uppi-spark-10.7.0.0.zip`) to a directory on your system. This will create a ZEMENTIS sub-directory with contents as described in Table 4.2. Once this is done, add `uppi-library-10.7.0.0.jar` to the classpath of your Spark application. The `driver program` of the Spark application can now use the ZEMENTIS API classes available in the `uppi-library-10.7.0.0.jar`.

When the Spark application is ready, create an application "uber JAR" file which contains the application code along with all the dependencies (including the contents of `uppi-library-10.7.0.0.jar` file). The application JAR should not include Hadoop or Spark libraries, as those are added at runtime. The application JAR file can then be submitted to a Spark cluster using the `spark-submit` script. For more information about submitting Spark applications, please refer to the Application Submission Guide.

Chapter 5. Using ZEMENTIS

To use ZEMENTIS in a Spark application, familiarity with the ZEMENTIS Java API is necessary. Section 5.1 provides an overview of the ZEMENTIS Java API. Section 5.2 then describes how this API can be used within a Spark application with some examples.

5.1. ZEMENTIS Stand-alone Java Library

The `uppi-library-10.7.0.0.jar` can be used as a stand-alone Java library within a Spark application. The Java API documentation is available under the `apidocs` sub-folder of the `uppi-spark-10.7.0.0.zip` distribution. The API consists of two interfaces:

- `ModelWrapperFactory`
- `ModelWrapper`

and their default implementations:

- `DefaultModelWrapperFactory`
- `DefaultModelWrapper`, `SerializableModelWrapper`

These classes encapsulate all the functionality that is necessary for processing a PMML file and execute predictive models from it. A `ModelWrapperFactory` object is constructed using the PMML file as an input. From this factory, one or more `ModelWrappers` can be created, one for each model found in the PMML file (a PMML file may contain more than one model). A `ModelWrapper` provides information about the wrapped model, including its name as well as the names and data types of the input and output fields. The `ModelWrapper` is also used to execute/apply the model, i.e. process data using the model.

Important

A distributed system like Spark needs to be able to serialize and deserialize objects when they are passed between tasks. To facilitate serialization and deserialization of the `ModelWrapper` instances created from the PMML files, `SerializableModelWrapper` should be used.

The code below illustrates how to use the `DefaultModelWrapperFactory` and `SerializableModelWrapper` for a desired model in a PMML file:

```
/*
 * Copyright (c) 2004-2016 Zementis, Inc.
 * Copyright (c) 2016-2017 Software AG, Darmstadt, Germany and/or Software AG USA Inc., Reston, VA, USA,
 and/or its
```

```
 * subsidiaries and/or its affiliates and/or their licensors.
 * Use, reproduction, transfer, publication or disclosure is prohibited except as specifically provided
 for in your
 * License Agreement with Software AG.
 */
ModelWrapperFactory modelWrapperFactory = null;
 try {
     modelWrapperFactory = new DefaultModelWrapperFactory(inputStream, pmmlFile.getName());
 } catch (RuntimeException rte) {
     // If the provided file is not a valid XML file.
     LOGGER.severe(rte.getMessage());
 }

 // Most of the PMML files contain only one model. Let's pick the first (and probably only one) to score.
 String modelName = modelNames.iterator().next();

 // Create a serializable model wrapper for the selected model
 SerializableModelWrapper modelWrapper = new
 SerializableModelWrapper(modelWrapperFactory.create(modelName));

 // Score the data. The predicted values are returned as an array.The size and order of the values in
 // the array must match the fields as returned by the getOutputFieldNames() method.
 Object[] inputValues = new Object[INPUT_SIZE];
 // Populate the input values.
 Object[] predictions = modelWrapper.apply(inputValues);

 // Alternatively, the following method can be used to apply the model to a key/value map.
 Map<String, Object> input = new HashMap<String, Object>();
 input.put("input_1", value_1);
 input.put("input_2", value_2);
 ...
 input.put("input_n", value_n);

 Map<String, Object> output = modelWrapper.apply(input);
```

If the provided input is indeed of PMML format but it has syntactic or semantic errors, then the construction will succeed but the constructed factory will not contain any models. The generated errors can be retrieved as an annotated PMML document as follows:

```
InputStream annotatedPmml = modelWrapperFactory.getAnnotatedPmml();
```

To use a new PMML file or replace an existing one, just create a new `DefaultModelWrapperFactory` and create the appropriate `SerializableModelWrapper` with it. For more information about using `upi-library-10.7.0.0.jar`, please refer to the Javadoc contained in the `apidocs` sub-folder of ZEMENTIS for Spark distribution.

Note

One of the `apply` methods of `ModelWrapper` expects `Object[]` as an input argument and returns `Object[]` as an output. The order and type of inputs provided to this method must match the order and type of input fields defined in the PMML mining schema for the corresponding model. Similarly, the order and type of outputs returned by the `apply` method matches the order and type of output fields defined in the corresponding PMML file. If the model returns only one output, `scalarApply` method of `ModelWrapper` object can be used.

Though, for most Spark applications, the `apply(Map)` method will be convenient and sufficient.

5.2. Integrate ZEMENTIS with Spark

With ZEMENTIS for Spark, the `SerializableModelWrapper` instances can be used within RDD transformations, specifically `map` and `mapPartitions`. The following sections describes this integration with examples.

5.2.1. Sample PMML and Data Files

The ZEMENTIS for Spark package contains a number of sample PMML files, each with a CSV (Comma Separate Values) file containing test data. The test data provides both input and output values. The output values are provided to validate the results generated on a Spark cluster. To run these samples, the test data needs to be converted into an RDD. In the examples presented below, we describe the process of creating `SerializableModelWrapper` instances from the sample PMML files and use them within Spark function to apply a transformation on an RDD which represents the data in the sample CSV files.

5.2.2. Example Model

This section provides an example of scoring data against a model on Spark in local mode. The example uses a Decision Tree model built for the Iris data set 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 of 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 fields output 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>
   ...
```

The following code snippet shows how to create a `SerializableModelWrapper` from this PMML file:

```
/*
 * Copyright (c) 2004-2016 Zementis, Inc.
 * Copyright (c) 2016-2017 Software AG, Darmstadt, Germany and/or Software AG USA Inc., Reston, VA, USA,
and/or its
 * subsidiaries and/or its affiliates and/or their licensors.
 * Use, reproduction, transfer, publication or disclosure is prohibited except as specifically provided
for in your
 * License Agreement with Software AG.
 */
   InputStream is = UPPIFunction.class.getResourceAsStream("Iris_CT.pmml");
   DefaultModelWrapperFactory modelWrapperFactory = new DefaultModelWrapperFactory(is, "Iris_CT");

   // If the PMML has any errors, they are reported as an annotated PMML file
   modelWrapperFactory.getAnnotatedPmml();

   // If no errors, get the name of the model (typically, the PMML will contain one model)
   String modelName = modelWrapperFactory.getModelNames().iterator().next();
   SerializableModelWrapper modelWrapper =
       new SerializableModelWrapper(modelWrapperFactory.create(modelName));
```

The following code snippet shows how the data from `Iris_CT.csv` is converted into RDD and transformed by applying the `Iris_CT` model to it as a Spark function:

```
/*
 * Copyright (c) 2004-2016 Zementis, Inc.
 * Copyright (c) 2016-2017 Software AG, Darmstadt, Germany and/or Software AG USA Inc., Reston, VA, USA,
and/or its
 * subsidiaries and/or its affiliates and/or their licensors.
 * Use, reproduction, transfer, publication or disclosure is prohibited except as specifically provided
for in your
 * License Agreement with Software AG.
 */
   // Initializing Spark Context
   JavaSparkContext sparkContext = new JavaSparkContext(SparkConf);

   // Retrieving input data from Iris dataset and converting it into RDDs
   JavaRDD<String> csvData = sparkContext.textFile("Iris_CT.csv");
   String header = csvData.first();
   String[] keys = header.split(",");
   JavaRDD<String> csvDataNoHeader = csvData.filter(new Function<String, Boolean>() {
       @Override
       public Boolean call(String line) {
           return !line.contains(header);
       }
   });
```

```
    JavaRDD<Map<String, Object>> input = csvDataNoHeader.map(
        new Function<String, Map<String, Object>>() {
        @Override
        public Map<String, Object> call(String line) {
            Map<String, Object> inputMap = new HashMap<String, Object>();
            String[] values = line.split(",");
            for (int i = 0; i < Math.min(keys.length, values.length); i++) {
                inputMap.put(keys[i], values[i]);
            }
            return inputMap;
        }
    });

    // Scoring using Spark Function object with models wrapped inside
    JavaRDD<Map<String, Object>> scoredOutput = input.map(
        new Function<Map<String, Object>, Map<String, Object>>() {
        @Override
        public Map<String, Object> call(Map<String, Object> tuple) throws Exception {
            return modelWrapper.apply(tuple);
        }
    });

    sparkContext.close();
```

The following code snippet shows the same example, but in a streaming context:

```
/*
 * Copyright (c) 2004-2016 Zementis, Inc.
 * Copyright (c) 2016-2017 Software AG, Darmstadt, Germany and/or Software AG USA Inc., Reston, VA, USA,
 and/or its
 * subsidiaries and/or its affiliates and/or their licensors.
 * Use, reproduction, transfer, publication or disclosure is prohibited except as specifically provided
 for in your
 * License Agreement with Software AG.
 */
    // Initializing Streaming Context
    JavaStreamingContext streamingContext = new JavaStreamingContext(sparkConfig, batchDuration);

    // Retrieving data from streaming
    Map<String, Integer> topicMap = new HashMap<String, Integer>();
    topicMap.put("test_in", 1);
    JavaPairReceiverInputDStream<String, String> messages =
        KafkaUtils.createStream(streamingContext, "localhost", UUID.randomUUID().toString(), topicMap);
    JavaDStream<String> csvLines = messages.map(new Function<Tuple2<String, String>, String>() {
        public String call(Tuple2<String, String> tuple) throws Exception {
            String line = tuple._2;
            return line;
        }
    });
    String[] keys = csvHeaders.split(",");
    JavaDStream<Map<String, Object>> inputRecords = csvLines.map(
        new Function<String, Map<String, Object>>() {
        public Map<String, Object> call(String csvLine) throws Exception {
            Map<String, Object> inputMap = new HashMap<String, Object>();
            String[] values = csvLine.split(",");
            for (int i = 0; i < Math.min(keys.length, values.length); i++) {
                inputMap.put(keys[i], values[i]);
            }
            return inputMap;
        }
    });

    // Scoring using Spark Function object with models wrapped inside
    JavaRDD<Map<String, Object>> scoredOutput = inputRecords.map(
        new Function<Map<String, Object>, Map<String, Object>>() {
        @Override
        public Map<String, Object> call(Map<String, Object> tuple) throws Exception {
```

```
            return modelWrapper.apply(tuple);
        }
    });

    // Start Streaming Context
    streamingContext.start();
```

Chapter 6. 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.

6.1. 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 which has a parameter `namespace`. This parameter must specify a 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 6.1 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 6.1. 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 6.1.

Figure 6.1. 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.

6.2. Use Custom PMML Functions

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 6.2 contains a simple example that uses the function defined in Figure 6.1.

Figure 6.2. 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 make custom functions available to ZEMENTIS, compile the corresponding classes into a JAR file and include the contents of this file in the final application JAR file. To compile a class using the `@PMMLFunctions` annotation, include the `uppi-library-10.7.0.0.jar` file in Java classpath. This file is included with the ZEMENTIS distribution package.

6.3. 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 time zone.

Figure 6.3. 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));
     }
}
```

6.4. Binary Sources

Some predictive models use binary data as input for scoring or classifying results. ZEMENTIS supports applying models to binary data by utilizing 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 6.1. Set `BINARY_BUFFERED` as `true` in `<Extension>` element like the PMML fragment in Figure 6.4 to guarantee the binary data will not be `null` after being consumed.

Figure 6.4. 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 6.4. An example of a custom function is shown in Figure 6.5.

Figure 6.5. 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 6.5 is compiled and deployed , `convert` can be used exactly like a built-in function within `Apply` transformations. The PMML fragment in Figure 6.6 contains a simple example that uses the function defined in Figure 6.5.

Figure 6.6. 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>
```