ZEMENTIS for Storm

User Guide

10.1.0.0

ZEMENTIS for Storm

User Guide

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 Storm 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 Storm. ZEMENTIS offers Storm 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 distributed realtime computation infrastructure, ZEMENTIS for Storm 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 Storm infrastructure to deliver superior performance for mission-critical business intelligence and analytics 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 Storm infrastructure. Practically, PMML becomes a Storm Bolt/Trident 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 Storm. 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 Storm with a PMML example, a decision tree. The example shows how to deploy and execute predictive models in Storm.

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 realtime streaming data processing system like Storm which operates on a core abstraction of unbounded sequence of tuples.

Note

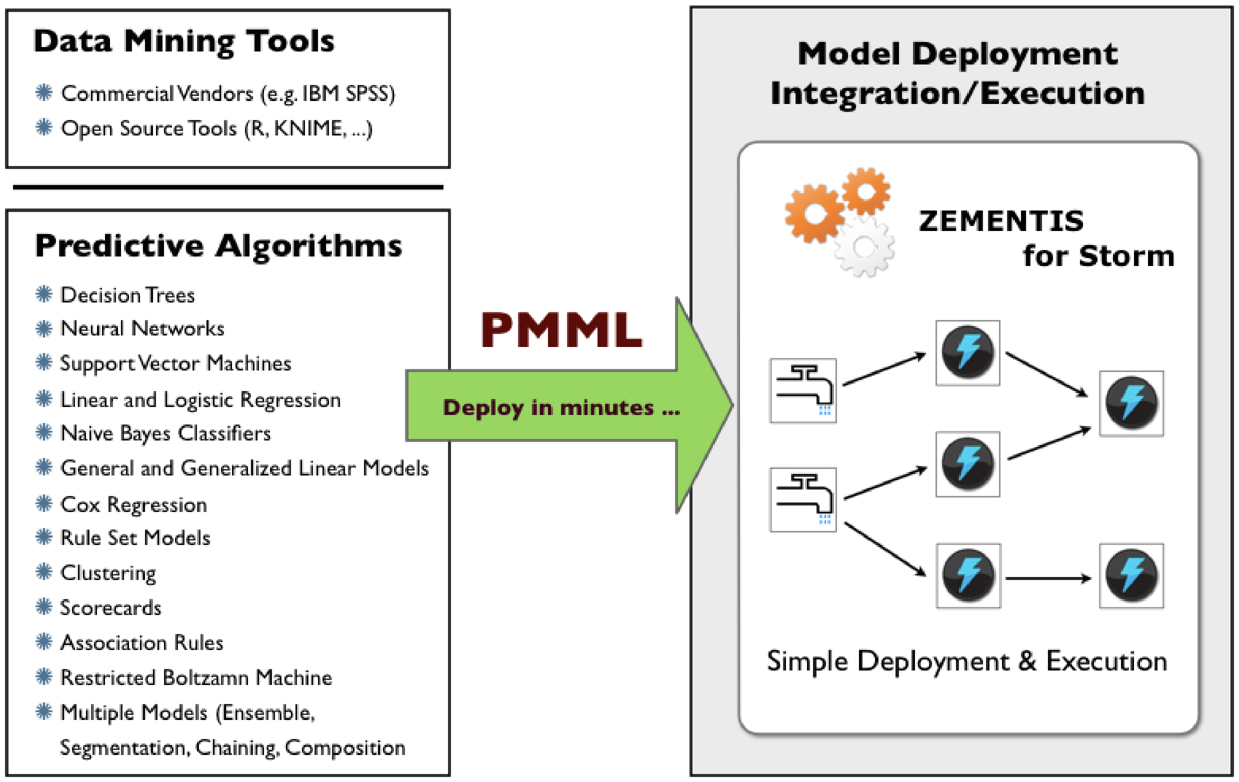
A variety of sample PMML models are included with the ZEMENTIS distribution package. In addition, a wealth of resources on PMML can be found from the PMML in Action.

2.2. ZEMENTIS Predictive Analytics (ZEMENTIS)

ZEMENTIS for Storm enables execution of standards-based predictive analytics directly within a Storm cluster. It shares the PMML execution core with the ZEMENTIS server offered by Software AG. ZEMENTIS for Storm, however, is optimized to seamlessly integrate PMML models into Storm Bolts and Trident Functions to enable predictive analytics on a Storm cluster while ensuring high data throughput with low latency semantics (see Figure 2.1).

ZEMENTIS for Storm converts each loaded PMML model into a Storm Bolt and Trident Function. The name, input tuples, and output tuples of each Storm Bolt/Trident Function match the name, input fields, and output fields of the corresponding model defined in the PMML file. This way, scoring a data set against one or more models in a Storm cluster requires nothing more than creating a corresponding Storm Bolt/Trident Function and integrating them in an existing Storm topology. The output of those models or "predictions" such as scores, probabilities, categories, and cluster identifiers can then be passed to other Storm Bolts/Trident Functions down-stream for further processing.

Figure 2.1. Overview of ZEMENTIS for Storm


Data Mining Tools
Commercial Vendors (e.g. IBM SPSS)
Open Source Tools (R, KNIME, ...)
Predictive Algorithms
Decision Trees
Neural Networks
Support Vector Machines
Linear and Logistic Regression
Naive Bayes Classifiers
General and Generalized Linear Models
Cox Regression
Rule Set Models
Clustering
Scorecards
Association Rules
Restricted Boltzamn Machine
Multiple Models (Ensemble, Segmentation, Chaining, Composition
PMML
Deploy in minutes ...
Model Deployment Integration/Execution
ZEMENTIS for Storm
Simple Deployment & Execution

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

1. *Preparation Step*: Validates the PMML files and prepares a `pmml.jar` file which is an implementation of a Storm Bolt and Trident Function corresponding to each input PMML file.
2. *Deployment Step*: Creates a new Storm/Trident topology using the Storm Bolt/Trident Function in the `pmml.jar` and deploys them into the Storm cluster.

These steps are described in more detail in Chapter 5 and illustrated with actual examples in Chapter 6.

After preparation and deployment steps are completed, it is then time to run the appropriate data process on the Storm cluster.

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 Storm does not support Association Rules models.

Chapter 3. Using PMML in Storm

This chapter describes in more detail how PMML models are used directly in Storm and Trident topologies by translating them into Storm Bolts and Trident Functions.

3.1. PMML Model as a Storm Bolt

A Storm Bolt [1] consumes one or more input streams, does some data processing, and possibly emits new streams. A stream is an unbounded sequence of tuples. In Storm topologies, all processing is done in Storm Bolts. Bolts can do anything from filtering, functions, aggregations, joins, connecting to databases, and more.

With ZEMENTIS for Storm, a predictive model is converted into a Java implementation of a Storm Bolt that can be packaged into Storm topologies. The name of the Storm Bolt is derived from the name of the model specified in the PMML file. Each value in the active and supplementary mining fields of the model becomes part of the input tuple for the generated Storm Bolt. Each value in the output fields of the model becomes part of the output tuple that is emitted by the Storm Bolt. This way a model can be easily deployed on a Storm cluster to perform scoring/classification process on incoming stream of data.

3.2. PMML Model as a Trident Function

A Trident Function[2] takes in a set of input tuples and emits zero or more tuples as output. The fields of the output tuple are appended to the original input tuple in the stream. In a Trident topology, streams are processed as a series of batches of tuples.

With ZEMENTIS for Storm, a predictive model is turned into a Java implementation of a Trident Function that can be used inside Trident topologies. The name of the Trident Function is derived from the name of the model specified in the PMML file. Each value in the active and supplementary mining fields of the model becomes part of the input tuple for the generated Trident Function. Each value in the output fields of the model becomes the output tuple that is emitted by the Trident Function.

In Trident topology, a stream is partitioned among the nodes in a cluster and operations are applied to a stream in parallel across each partition. By representing the PMML model as a Trident Function, we ensure that each batch partition is processed independently and involves no network transfer. Trident has consistent, exactly-once processing semantics, ensuring that a PMML model is applied exactly-once to the input tuple.

[1]For more information on bolt, please refer to: Storm documentation tutorial.

[2]For more information on function, please refer to: Trident documentation API overview.

3.3. 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`
`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[]`

3.4. 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 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 Storm and Trident topologies, the option `returnInvalid` may have a significant (unintended) impact. Consider the case where a tuple 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 will fail the input tuple causing the spout to handle the replay strategy for the failed tuple. If the tuple replay is not handled properly at the spout, the tuple may get unnecessarily replayed multiple times or not get replayed at all.

In some cases, this may be the desired behavior so that invalid values can be detected. However, it is often the case that an alternative approach where invalid values do not cause tuples to fail 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 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 record cannot be processed.

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

4.1. Requirements

Following are the requirements for installing ZEMENTIS for Storm on your system:

Table 4.1. The ZEMENTIS Installation Requirements

Requirement	**Version**	**Notes**
Storm	1.0.2	The rest of this documentation assumes that Storm is already installed. Please see the Storm documentation for details.
Java Platform, Standard Edition (Java SE)	7 or above	Please make sure you use the Java Development Kit (JDK) and not the Java Runtime Environment (JRE).
ZEMENTIS for Storm License Key	10.1.0.0	Installation of new PMML models with ZEMENTIS for Storm 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` script is executed. More information about `prepare-pmml.sh` can be found in Table 5.2. Please note that execution of existing models will not be interrupted when the license expires.

4.2. Packaging

ZEMENTIS for Storm is distributed as a compressed archive file: `uppi-storm-10.1.0.0.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 Storm package

Directory	**Contents**
`bin`	Contains the `prepare-pmml.sh` scripts needed for generating the Storm Bolt and Trident Function from PMML files (see Section 5.1.1).
`docs`	Documentation in HTML and PDF format.
`lib`	The library (JAR) files required for installing ZEMENTIS for Storm (see Section 4.3).

Directory	Contents
`pmml`	A number of sample PMML files along with data files in CSV format. These include the examples described in Chapter 6.
`standalone`	Contains the artifacts for using ZEMENTIS as a stand-alone Java library within Storm and Trident topologies (described in Section 5.2). For details on contents of the sub-folders, please refer to the `README.txt` file located in this folder. The `standalone/apidocs` sub-folder contains detailed Javadoc on how to use ZEMENTIS as a stand-alone Java library.

4.3. Installation

To install ZEMENTIS for Storm, simply uncompress the provided file (`uppi-storm-10.1.0.0.zip`) to a directory on your system. This will create a ZEMENTIS sub-directory with contents as described in Table 4.2. The Java library file `uppi-storm-10.1.0.0.jar` contained in the `lib` folder needs to be available in the Storm cluster at runtime. Typically, the `uppi-storm-10.1.0.0.jar` will be installed along with the topology jars.

Note

The provided package (`uppi-storm-10.1.0.0.zip`) does not have to be extracted on the same host as Storm cluster nimbus node. It can be extracted on a separate host which can act as a client to the Storm cluster nimbus node. The host must have the following software installed:

1. Java Platform, Standard Edition 7 or above (JDK) available in the classpath
2. Storm jar which can be used as client to submit topologies to a Storm cluster

Chapter 5. Using ZEMENTIS

There are two ways in which ZEMENTIS can be used within a Storm cluster. With the first approach, described in Section 5.1, a Storm Bolt and Trident Function is generated for each of the provided PMML files. These Storm Bolts and Trident Functions can be programatically incorporated into topologies which can then be deployed on a Storm cluster. With the second approach, described in Section 5.2, `uppi-library-10.1.0.0.jar` can be used within a Storm/Trident topology as a stand-alone Java library. Using this approach, PMML files can be dynamically uploaded and made available for processing with-in existing Storm Bolts or Trident Functions using a convenient Java API.

5.1. Generating Storm Bolt and Trident Function from PMML

This approach involves two steps which assume that:

- A Storm cluster is running.
- 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.

Note

As shown in Table 5.2, ZEMENTIS is distributed with several sample PMML files. These can be used to validate that ZEMENTIS for Storm has been installed correctly and that you have mastered the preparation and installation steps described below.

5.1.1. Step 1: Prepare Storm Bolt and Trident Function

The preparation step involves creation of a Storm Bolt and Trident Function from the available PMML models. The ZEMENTIS for Storm distribution package contains a `prepare-pmml.sh` script (for Unix/Linux environment) that is used for this purpose. This script will first validate the PMML models and then generate the necessary implementations for Storm Bolt and Trident Function for the corresponding models.

Important

The `prepare-pmml.sh` script requires that the environment variable *`JAVA_HOME`* is set to the installation directory of the Java Development Kit 7 or above (JDK).

From a Storm distribution package or installation (version 1.0.2), obtain a copy of the file `storm-core-1.0.2.jar` and place it in the `lib` directory of the ZEMENTIS distribution. This file is usually located in the `lib` directory of the Storm distribution.

Run the preparation script by using the following command on a Linux/Windows environment:

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

where *`UPPI_DIR`* refers to the directory where the ZEMENTIS distribution package has been uncompressed and *`PMML_DIR`* refers to the (top) directory where the PMML files are located.

Note

To use your own PMML files, you can either copy them into the existing PMML directory, or set *`PMML_DIR`* to point to the directory where your PMML files are located.

Table 5.1. `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.
-out	*`path`*	`uppi-output`	No	Specify path to generated output files directory.
-extLib	*`path`*	`""`	No	Specify path to custom functions JAR file(s) directory. See Chapter 7 regarding custom PMML functions.
-excpOnInvalid	`true` or `false`	`true`	No	If `true`, Storm processing throws exception when encountering an invalid value. The exception will result in immediate fail of the input tuple. This will cause the input tuple to be replayed by the depending on Storm topology and settings. If `false` tuple processing continues by returning null values. We recommended that the treatment of invalid values should be handled in the PMML as described in Section 3.4.
-runWithPmml	`true` or `false`	`false`	No	If `true`, include PMML file in generated Storm or Trident JAR instead of binary representation of the PMML model. This option should be used only for debugging or diagnostic purpose.

The script will generate a series of messages marked as `INFO`, `WARNING`, or `SEVERE` during its execution. 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 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 directory. You can also change the default output directory by specifying an output directory of choice as a parameter when running the preparation script. The contents of the output directory are described in Table 5.2.

Table 5.2. Output generated from the `prepare-pmml.sh` script.

File or Directory	Description
`pmml.jar`	A JAR file containing the validated PMML model as well as generated Java code that serves as wrapper for the model. As we will describe in the next sections, this wrapper contains the Storm Bolt and Trident Function implementations that can be incorporated into a Storm topology and Trident topology respectively.
`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 `pmml.jar` file.
`classes`	A directory containing the compiled Java code, along with the original PMML files. This reflects the contents of the `pmml.jar` file.

For diagnostic purposes the `java` directory contains the generated Java code. Please open the `java` folder and make sure that Java files reside inside the folder. Likewise, class files are generated in the `classes` folder which also should not be empty.

Out of all these files, only the `pmml.jar` file is used in the deployment step. The rest are generated during the execution of the script and remain in place for diagnostic purposes only.

5.1.2. Step 2: Deploy on a Storm cluster

To deploy the generated Storm Bolt and Trident Function, simply define a corresponding Storm topology or Trident topology built around them. Once the desired topology is created, it can be deployed on a Storm cluster by using a storm client. The storm client typically requires the topology code and all the dependencies of your code packaged in a single JAR file. This means, the resulting JAR file will also need to package the contents of `pmml.jar` and `uppi-storm-10.1.0.0.jar` files.

One possible way to deploy a Storm topology on a Storm cluster is via Storm command line client, as shown below:

```
storm jar PATH_TO_TOPOLOGY_JAR CLASS_NAME
```

where *PATH_TO_TOPOLOGY_JAR* refers to the Uber/Fat JAR file that contains the topology code and all the dependencies, including contents of `pmml.jar` and `uppi-storm-10.1.0.0.jar` files. The *CLASS_NAME* refers to the class which typically contains code that starts the topology.

5.2. ZEMENTIS Stand-alone Java Library

The `uppi-library-10.1.0.0.jar` can be used as a stand-alone Java library within a Storm topology. The Java API documentation is available under `standalone/apidocs` sub-folder of the `uppi-storm-10.1.0.0.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 Storm 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 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` from it. For more information about using `uppi-library-10.1.0.0.jar`, please refer to the Javadoc contained in the `standalone/apidocs` sub-folder of ZEMENTIS for Storm distribution.

Note

The code snippets listed above can be part of an existing Storm Bolt and Trident Function implementation. 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.

Chapter 6. Examples

6.1. Sample PMML and Data Files

The ZEMENTIS for Storm package contains a number of sample PMML files, each with a CSV (Comma Separate Values) file containing test data. The test data contains both input and output values. The output values are provided to allow validation of the results generated on a Storm cluster. To run each example, the test data needs to be generated from a Storm spout. In the examples presented below, we describe the process of creating the topologies and adding necessary spouts, bolts, and functions into them.

6.2. Example Model

This section provides an example of scoring data against a model on Storm topology. 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>
    ...
```

To test this sample file, execute the preparation and deployment steps described in Section 5.1.1 and Section 5.1.2. Since 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 sub-directory containing only the tree models:

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

Note

The above command will prepare all models in the `TreeModel` sub-directory, including `Iris_CT.pmml`. Consequently, in the deployment step (described in Section 5.1.2), more than one Storm Bolts and Trident Functions are generated, including the one from model file `Iris_CT.pmml`, which could be used to build a Storm topology.

Running the `prepare-pmml.sh` script for `Iris_CT.pmml` will generate the `Iris_CT.java` file in the `uppi-output` directory. As seen in the listing below, the name of the Java class is derived from the model name. The `Iris_CT.java` contains two sub-classes, `Iris_CT$Bolt` which is a Storm Bolt implementation for `Iris_CT.pmml` and `Iris_CT$Function` which is a Trident Function implementation for `Iris_CT.pmml`.

The Storm Bolt operates on an input `Tuple` by getting values corresponding to input field names (defined in the `MiningSchema` element) from the `Tuple`. The Storm Bolt then emits named value(s) corresponding to output field names (defined in the `MiningSchema` or `Output` elements) as an output. Similarly, the Trident Function operates on an input `TridentTuple` by getting values corresponding to input field names from the `TridentTuple` and emits named value(s) corresponding to output field names as an output.

```
/*
 * 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.
 */
package com.zementis.uppi.storm;

import java.util.UUID;

import com.zementis.uppi.storm.StormModelWrapper.Factory;

public final class Iris_CT {
    private static final StormModelWrapper.Factory FACTORY = new Factory(Iris_CT.class, "/Iris_CT.pmml",
    false);

    private static final StormModelWrapper MODEL_WRAPPER = FACTORY.create("Iris_CT");

    public static class Bolt extends AbstractPMMLBolt {
        private static final long serialVersionUID = UUID.randomUUID().getLeastSignificantBits();

        @Override
        public StormModelWrapper getStormModelWrapper(){
            return MODEL_WRAPPER;
        }
    }

    public static class Function extends AbstractPMMLFunction {
        private static final long serialVersionUID = UUID.randomUUID().getLeastSignificantBits();

        @Override
        public StormModelWrapper getStormModelWrapper(){
            return MODEL_WRAPPER;
        }
    }
}
```

6.2.1. Storm Topology

Once the relevant Storm Bolts are generated, they can then be incorporated into a Storm topology. The resulting Storm topology can be submitted on a Storm cluster using the following code snippet:

```
// define the topology
TopologyBuilder topologyBuilder = new TopologyBuilder();
topologyBuilder.setSpout("Spout_ID", new Spout());
topologyBuilder.setBolt("Bolt_ID", new Iris_CT$Bolt()).shuffleGrouping("Spout_ID");

// submit the topology
StormSubmitter.submitTopology("mytopology", new HashMap(), topologyBuilder.createTopology());
```

The `Spout_ID` above is referenced by other components that want to consume this spout's outputs. For the above example the spout generates tuples based on the example CSV file for the Iris dataset. The `Bolt_ID` above is referenced by other components that want to consume this bolt's outputs.

6.2.2. Trident Topology

Like in the previous example, once relevant Trident Functions are generated, they can then be incorporated into a Trident topology. The resulting Trident topology can be submitted on a Storm cluster using the following code snippet:

```
// define the topology
TridentTopology topology = new TridentTopology();
topology
 .newStream("Spout_ID", new Spout())
 .each(inputField, new Iris_CT$Function, outputField);

// submit the topology
```

```
StormSubmitter.submitTopology("mytopology", new HashMap(), topology.createTopology());
```

The `Spout_ID` must be unique across all Trident topologies running on the cluster. Trident keeps track of a small amount of state for each input source (metadata about what it has consumed) in Zookeeper, and the `Spout_ID` above specifies the node in Zookeeper where Trident should keep that metadata.

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 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 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

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 is used as a 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 the prefix corresponds to the `namespace` parameter of annotation `@PMMLFunctions`, and the 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 topology JAR file. To compile a class containing the `@PMMLFunctions` annotation, include the `uppi-storm-10.1.0.0.jar` file in Java classpath. This file is included with the ZEMENTIS distribution package.

To prepare Storm Bolt and Trident Function for PMML files which references custom 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 5.1.

Then, simply include custom functions JAR in addition to `pmml.jar` in your Storm deployment artifact.

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 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 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 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 , `convert` can be used exactly like a built-in function within `Apply` transformations. 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>
```