



ZEMENTIS Predictive Analytics Solutions Guide

10.1.0.0

ZEMENTIS Predictive Analytics

Solutions Guide

Software AG

Copyright © 2004 - 2016 Zementis Inc.

Copyright © 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.

This document applies to ZEMENTIS 10.1.0.0 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.

The name Software AG and all Software AG product names are either trademarks or registered trademarks of Software AG and/or Software AG USA Inc. and/or its subsidiaries and/or its affiliates and/or their licensors. Other company and product names mentioned herein may be trademarks of their respective owners.

Detailed information on trademarks and patents owned by Software AG and/or its subsidiaries is located at <http://softwareag.com/licenses/>.

Use of this software is subject to adherence to Software AG's licensing conditions and terms. These terms are part of the product documentation, located at <http://softwareag.com/licenses/> and/or in the root installation directory of the licensed product(s).

This software may include portions of third-party products. For third-party copyright notices, license terms, additional rights or restrictions, please refer to "License Texts, Copyright Notices and Disclaimers of Third Party Products". For certain specific third-party license restrictions, please refer to section E of the Legal Notices available under "License Terms and Conditions for Use of Software AG Products / Copyright and Trademark Notices of Software AG Products". These documents are part of the product documentation, located at <http://softwareag.com/licenses/> and/or in the root installation directory of the licensed product(s).

Table of Contents

1. Introduction	1
1.1. Decision Solutions Overview	1
2. Predictive Models	3
2.1. Predictive Model Building Process	6
2.2. Deploy and Test Predictive Models	9
2.2.1. Deploying Models	9
2.2.2. Testing Models	10
2.3. Data Scoring and Classification	11
2.4. Other Data Sources	11
3. Custom Resources	12
3.1. Custom PMML Functions	12
3.1.1. Create Custom PMML Functions	12
3.1.2. Use Custom PMML Functions	14
3.1.3. Non-Deterministic Functions	15
3.2. External Lookup Tables	15
3.2.1. Create Lookup Tables in Excel	16
3.2.2. Use Lookup Tables in PMML	18
3.3. External Training Data Tables	19
3.4. Binary Sources	19
3.4.1. Using Default <code>binary</code> Type	20
3.4.2. Using Buffered <code>binary</code> Type	22
3.5. Deploy Resources	23
3.5.1. Deleting Resources	24
4. Extensions API	25
4.1. Using the ZEMENTIS Extensions API	25
4.1.1. Custom Function	25
4.1.2. Lookup Table	26
4.1.3. Asset Repository	27
4.1.4. Logging Store	28
4.2. Overview of code examples	29
4.3. Deployment of ZEMENTIS Extensions	30
5. SOAP/XML Web Services	31
5.1. Web Service Address (URL)	31
5.2. Models Web Service	32

5.2.1. SOAP Request and Response	36
5.3. RPC Web Service	43
5.4. Using ZEMENTIS Web Services from Java	45
5.4.1. Models Web Service	46
5.4.2. RPC Web Service	50
6. REST API	53
6.1. General Notes	53
6.1.1. URI	53
6.1.2. Request	55
6.1.3. Response	55
6.1.4. Errors	55
6.1.5. Authorization	57
6.2. API	58
6.2.1. JSON Objects	58
6.2.2. Operations on Models	64
6.2.2.1. List Available Models	64
6.2.2.2. Get Model Information	65
6.2.2.3. Get Model Source	66
6.2.2.4. Get Model Serialized Source	67
6.2.2.5. Upload New Model with POST	67
6.2.2.6. Upload New Model with PUT	69
6.2.2.7. Activate an existing Model with PUT	70
6.2.2.8. Deactivate an existing Model with PUT	71
6.2.2.9. Remove Model	72
6.2.2.10. Remove All Models	73
6.2.3. Apply model	73
6.2.3.1. Apply Model to Single Record	73
6.2.3.2. Apply Model to Single Record and Explain Result	74
6.2.3.3. Apply Model to Multiple Records or Apply Model to Single Binary Data with POST.....	76
6.2.3.4. Apply Model to Multiple Records or Apply Model to Single Binary Data with PUT	78
6.2.3.5. Asynchronously Apply Model to Multiple Records with POST	80
6.2.3.6. Asynchronously Apply Model to Multiple Records with PUT	81
6.2.4. Operations on Resources	82
6.2.4.1. List Available Resources	82
6.2.4.2. Get Resource Information	83
6.2.4.3. Get Resource File	84
6.2.4.4. Upload New Resource File with POST	85

6.2.4.5. Upload New Resource File with PUT	86
6.2.4.6. Remove Resource File	87
6.2.4.7. Remove All Resource Files	88

List of Figures

1.1. Decision Solution	2
2.1. Model Ensemble	5
2.2. Predictive Modeling Process	6
2.3. Predictive Models in the ZEMENTIS Console	10
3.1. Custom PMML Function Example	13
3.2. Example Using a Custom Function in PMML	14
3.3. Custom PMML Function Example	15
3.4. Lookup Table Example	16
3.5. Sample Excel Lookup Table	17
3.6. A LookupTable with two inputs and one output	18
3.7. Binary DataType Example	20
3.8. Custom Function of Binary Data Example	21
3.9. Example Using Custom Function of Binary Data in PMML	22
3.10. Binary (Buffered) DataType Example	22
3.11. Custom Function of Buffered Binary Data Example	23
3.12. Example Using Custom Function of Buffered Binary Data in PMML	23
3.13. Resource Files in the ZEMENTIS Console	24
3.14. Resource dependency exception in the ZEMENTIS Console	24
4.1. Dependencies for Custom Functions	26
4.2. Dependencies for Lookup Table	27
4.3. Dependencies for Asset Repository	28
4.4. Dependencies for Logging Repository	29
6.1. Interactive REST API Documentation	54

List of Tables

2.1. Sample Predictive Models	7
2.2. Directory Structure of Sample Models	8
3.1. PMML and Java types in ZEMENTIS	13
4.1. Directory structure of code examples	29
5.1. Overview of ZEMENTIS Web Services	31
5.2. ZEMENTIS Web Service Addresses (URLs)	32
5.3. Operations of ZEMENTIS Web Services	32
6.1. Typical ZEMENTIS REST Error Responses	56
6.2. ZEMENTIS REST Permissions	57

List of Examples

5.1. ZEMENTIS Models Web Service Import Model	36
5.2. ZEMENTIS Models Web Service Describe Model	37
5.3. ZEMENTIS Models Web Service Apply Model	38
5.4. ZEMENTIS Models Web Service Apply Model with Settings	39
5.5. ZEMENTIS Models Web Service Apply Model to Binary Source	41
5.6. ZEMENTIS Models Web Service Apply Model to CSV	41
5.7. ZEMENTIS Models Web Service Import Resource	42
5.8. ZEMENTIS RPC Web Service WSDL	44
5.9. ZEMENTIS RPC Web Service SOAP Request Body	44
5.10. ZEMENTIS RPC Web Service SOAP Response Body	45
5.11. ZEMENTIS Models Web Service Java client	46
5.12. ZEMENTIS RPC Web Service Java client	51
6.1. ZEMENTIS REST Error Response	56
6.2. ZEMENTIS REST Errors Object	58
6.3. ZEMENTIS REST Models Object	59
6.4. ZEMENTIS REST ModelInfo Object	60
6.5. ZEMENTIS REST Field Object	61
6.6. ZEMENTIS REST Record Object	61
6.7. ZEMENTIS REST Record Object	62
6.8. ZEMENTIS REST Result Object	63
6.9. ZEMENTIS REST ResourceInfo Object	63
6.10. ZEMENTIS REST Resources Object	64
6.11. ZEMENTIS REST List Models	64
6.12. ZEMENTIS REST Get Model Information	65
6.13. ZEMENTIS REST Get Model Source	66
6.14. ZEMENTIS REST Get Model Serialized	67
6.15. ZEMENTIS REST Upload New Model with POST	68
6.16. ZEMENTIS REST Upload New Model with PUT	69
6.17. ZEMENTIS REST Activate an existing Model	70
6.18. ZEMENTIS REST Deactivate an existing Model	71
6.19. ZEMENTIS REST Remove Model	72
6.20. ZEMENTIS REST Remove All Models	73
6.21. ZEMENTIS REST Apply Model to Single Record	74
6.22. ZEMENTIS REST Apply Model to Single Record and Explain Result	75

6.23. ZEMENTIS REST Apply Model to Multiple Records with POST	77
6.24. ZEMENTIS REST Apply Model to Single Binary Record with POST	77
6.25. ZEMENTIS REST Apply Model to Multiple Records with PUT	79
6.26. ZEMENTIS REST Apply Model to Single Binary Record with PUT	79
6.27. ZEMENTIS REST Asynchronously Apply Model to Multiple Records with POST	80
6.28. ZEMENTIS REST Asynchronously Apply Model to Multiple Records with PUT	82
6.29. ZEMENTIS REST List Resources	83
6.30. ZEMENTIS REST Get Resource Information	84
6.31. ZEMENTIS REST Get Resource File	84
6.32. ZEMENTIS REST Upload New Resource File with POST	85
6.33. ZEMENTIS REST Upload New Resource File with PUT	86
6.34. ZEMENTIS REST Remove Resource File	87
6.35. ZEMENTIS REST Remove All Resource Files	88

Chapter 1. Introduction

ZEMENTIS (ZEMENTIS Predictive Analytics) enables the agile deployment and integration of predictive decision services. It allows organizations to convert predictive models into operational services without requiring any additional custom coding by the information technology (IT) organization. ZEMENTIS ensures model integrity, optimizes performance and powers scaling as necessary.

This document serves as a guide for creating decision solutions using ZEMENTIS. It describes how ZEMENTIS components are used to verify and execute your advanced analytics either in real-time (against in-flight data) or batch mode (against data at rest). This guide also explains how the different ZEMENTIS components are combined to offer a powerful scoring framework.

It is important to note that ZEMENTIS leverages the Predictive Model Markup Language (PMML) standard. PMML handles data pre-processing and post-processing as well as the predictive model itself. In this way, the entire predictive workflow can be implemented in PMML.

1.1. Decision Solutions Overview

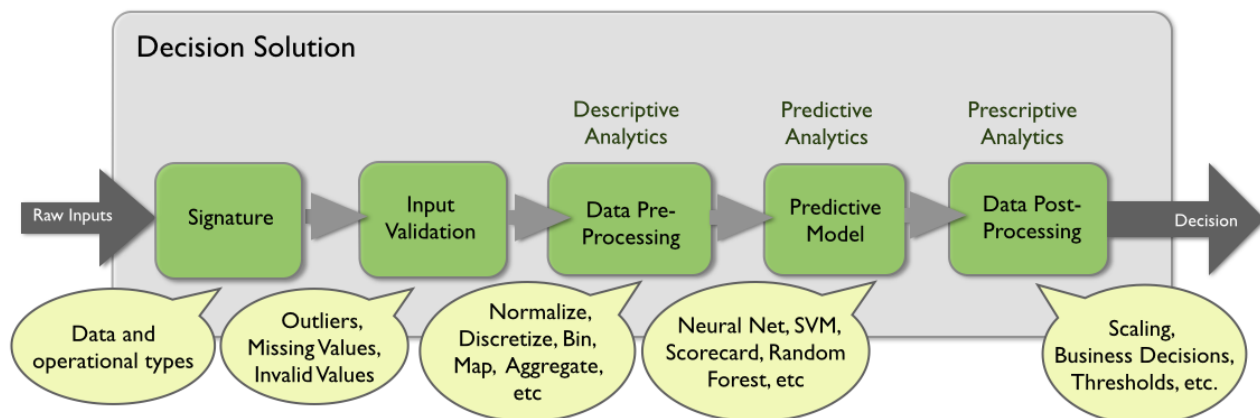
ZEMENTIS allows data-driven insight and expert knowledge to be combined into a single and powerful decision strategy through the use of PMML. Whereas *expert knowledge* encapsulates the logic used by experts to solve problems, *data-driven knowledge* is based on the ability to automatically recognize patterns in data not obvious to the expert eye. These two different types of knowledge are represented by two powerful technologies: Business Rules and Predictive Analytics. By bringing together both technologies, ZEMENTIS offers the best combination of control and flexibility for orchestrating critical day-to-day business decisions.

During the process of building a predictive model, there is usually quite a bit of data analysis and data pre-processing that need to take place. This is done to prepare the raw historical data so that it is suitable for model building and/or to combine and transform different data fields so that they create feature detectors that offer a richer predictive power than the input fields they were derived from. More often than not, such features entail looking at the data from a descriptive point of view as to explain it. For example, a feature detector may be defined as an aggregate value of an input field containing the transaction amount as part of a credit card transaction. If this feature detector is the average transaction amount for the last month or week, the predictive model can use this information to generate a prediction that takes into account the delta between the current amount and the average past amount. The average amount for the last month is a typical case of descriptive analytics which tries to answer what happened in the past. Descriptive features are extremely important since when fed into a predictive model, they transform the nature of the information itself, allowing a model to answer what will happen next. Through PMML, ZEMENTIS is capable of

capturing this process entirely. It also takes it a step further, by including prescriptive analytics into the mix, which is implemented by a series of post-processing steps expressed via the use of business rules.

Prescriptive analytics takes advantage of the outputs generated by a predictive technique by transforming them into business decisions. As depicted in [Figure 1.1](#) the process of integrating descriptive, predictive and prescriptive analytics into a single solution is easy and straightforward with PMML and ZEMENTIS.

Figure 1.1. Decision Solution



With ZEMENTIS, the power of predictive analytics is made available to any other application in your enterprise via web services. Without further configuration or customization, a predictive model is exposed as a web service and seamlessly participates in the overall business process flow.

ZEMENTIS is the first technology solution that enables an enterprise to score data from any source in batch or real-time while combining the power of descriptive, predictive and prescriptive analytics by leveraging a well-supported open industry standard. By using ZEMENTIS and PMML both the human resource and technology requirement to deploy these powerful analytics is drastically reduced. At the same time, pace of deployment is increased and model integrity and quality is improved.

In this guide, we start by describing the process of building and testing a predictive model in [Chapter 2](#). This is followed by a description of custom resources in [Chapter 3](#). [Chapter 4](#) gives an overview of the ZEMENTIS Java Extensions API. Finally, an extensive description of the web service capabilities is provided in [Chapter 5](#) and [Chapter 6](#).

Chapter 2. Predictive Models

The conversation around Big Data for both technologists and businesses has become pervasive. The challenge many enterprises and teams face is how to deliver measurable value from Big Data initiatives. By enabling rapid deployment from the Data Scientist's desktop to the operational IT environment, ZEMENTIS and PMML provide a standards-based methodology and process through which value from Big Data initiatives can be gained, quantified and demonstrated.

The predictive model building process begins by working with and developing a deep understanding of historical data which is mined for feature detectors. These are in turn used to build the predictive models. While a time consuming and laborious process, this provides the foundation for creating value from Big Data.

Building models is only the first step to realizing the benefits of predictive analytics. The second and final step is to actually use them within the overall business flow and processes. In other words, the models need to move from the data scientist's desktop into the enterprise operational IT environment where they can be used for scoring new data and drive business decisions.

Deployment of predictive models into the IT operational environment is all but straightforward. It can take as long as the data analysis phase itself or even longer and consume a significant amount of resources. It is not uncommon that by the time models are finally deployed, they are already stale and require to be refreshed with newer (historical) data reflecting a changing market.

ZEMENTIS makes deployment and use of complex predictive models trivial. ZEMENTIS has been designed from the ground up to consume, execute, optimize and scale Predictive Models that have been saved in PMML. PMML is the standard for moving predictive models between applications and, as a consequence, is supported by the leading technology companies including IBM, Microsoft, Oracle, SAP, SAS and Software AG to name a few. PMML is developed by the [Data Mining Group \(DMG\)](#), an independent vendor led consortium that develops data mining standards.

PMML is a very mature standard. Its latest version, PMML 4.3, was released in August 2016. Given that different data mining tools may support different versions of PMML, ZEMENTIS incorporates proprietary IP developed by Software AG that converts any older version of PMML (versions 2.0, 2.1, 3.0, 3.1, 3.2, 4.0, 4.1, and 4.2) into version 4.3. This converter also checks the code for any syntactic and semantic problems and corrects known issues found in the PMML code of certain model building tools automatically.

PMML 4.2 incorporates many new elements into the standard, including elements for text-mining as well as built-in functions for string manipulation. These are concat, matches and replace. PMML 4.2 also simplified the way outputs are implemented and established a way for the outputs of a segment in a multiple model scenario to be output by the top mining model. PMML 4.2 also enhanced the Naive Bayes model element to include continues input fields

and the scorecard element to implement partial scores based on expressions which may include input or derived fields. ZEMENTIS is compliant with PMML 4.2 and so it can consume PMML code that incorporates all the new PMML 4.2 features.

Note

PMML 4.2 changed the way the target field is referred to in the mining schema element. In PMML 4.2, the target field is simply referred to as "target" while in previous versions of PMML, it was referred to as "predicted". This change avoids any confusion related to the target field which is used to train a model and the true predicted field which is output by a model after scoring. As a consequence, ZEMENTIS also changed the way it treats predicted fields. If a PMML file is missing the output element, ZEMENTIS will add it to the file and will name the predicted output field "predictedValue" if no target field name is specified in the model's mining schema. If however, the target field is given, ZEMENTIS will name the predicted output field "predictedValue_X" where X is the name of the target field as specified in the mining schema. ZEMENTIS will not add any output fields to a PMML file if it already has an output element.

If you would like to learn more about PMML, we highly recommend that you visit the [Software AG](#) web site for a list of resources. We also recommend the book [PMML in Action \(2nd Edition\): Unleashing the Power of Open Standards for Data Mining and Predictive Analytics](#) by Alex Guazzelli, Wen-Ching Lin, and Tridivesh Jena, which is available for purchase on Amazon.com. "PMML in Action" gives an introduction to PMML as well as a PMML-based description of all the predictive modeling techniques supported by ZEMENTIS.

Software AG also offers a two-day on-site training course in PMML which is usually enough training for data scientists to become highly productive in using PMML. No pre-requisites for this course are required to be effective.

ZEMENTIS supports an extensive collection of statistical and data mining algorithms. These are:

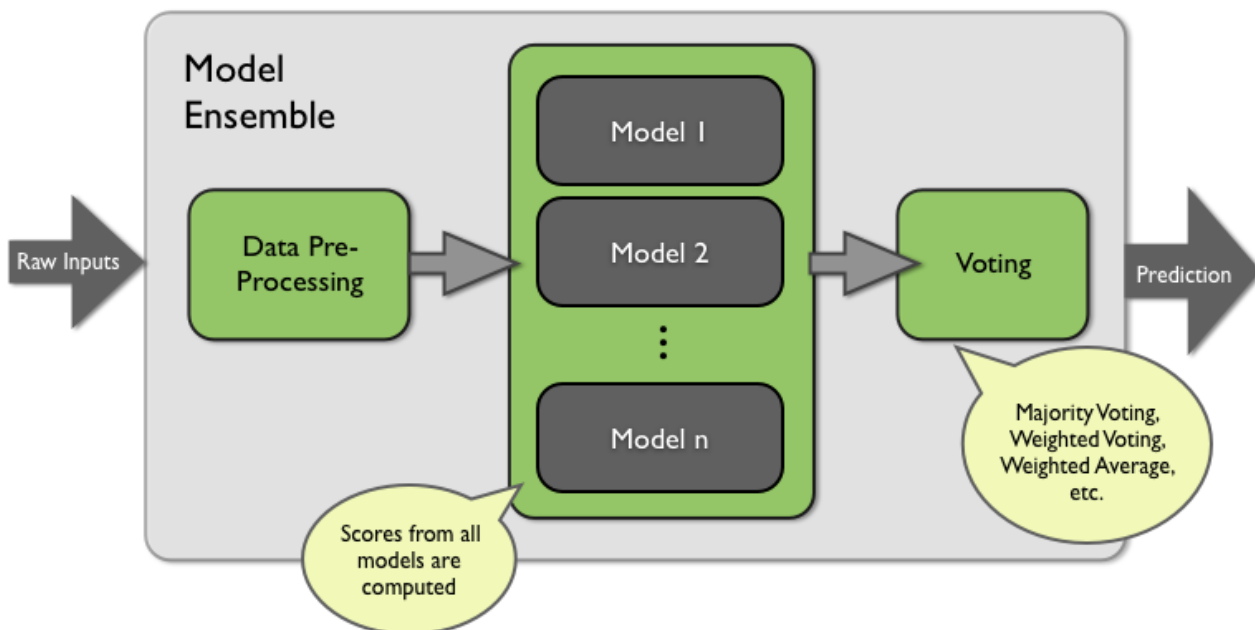
- Association Rules Models (Rectangular or Transactional format)
- Clustering Models (Distribution-Based, Center-Based, and 2-Step Clustering)
- Decision Trees (for classification and regression) together with multiple missing value handling strategies (Default Child, Last Prediction, Null Prediction, Weighted Confidence, Aggregate Nodes)
- K-Nearest Neighbors (for regression, classification and clustering)
- Naive Bayes Classifiers (with continuous or categorical inputs)
- Neural Networks (Back-Propagation, Radial-Basis Function, and Neural-Gas)
- Regression Models (Linear, Polynomial, and Logistic) and General Regression Models (General Linear, Ordinal Multinomial, Generalized Linear, Cox)

- Ruleset Models (Each rule contains a predicate and a predicted class value)
- Support Vector Machines (for regression and multi-class and binary classification)
- Scorecards (point allocation for categorical, continuous, and complex attributes as well as support for reason codes)
- Multiple models (model ensemble, segmentation, chaining, composition and cascade), including Random Forest Models and Stochastic Boosting Models

ZEMENTIS also implements the definition of a data dictionary, missing and invalid values handling, outlier treatment, as well as a myriad of functions for data pre- and post-processing, including: text mining (introduced in PMML 4.2), value mapping, discretization, normalization, scaling, logical and arithmetic operators, conditional logic, built-in functions, business decisions and thresholds.

Due to the highly publicized [Netflix prize](#) and the many tools that now make it easier for data scientists to develop a solution containing multiple models, model ensembles are now being used to build many predictive solutions. As depicted in [Figure 2.1](#), in a model ensemble, every model is executed and the overall result or output is a combination of the partial results obtained from each model.

Figure 2.1. Model Ensemble

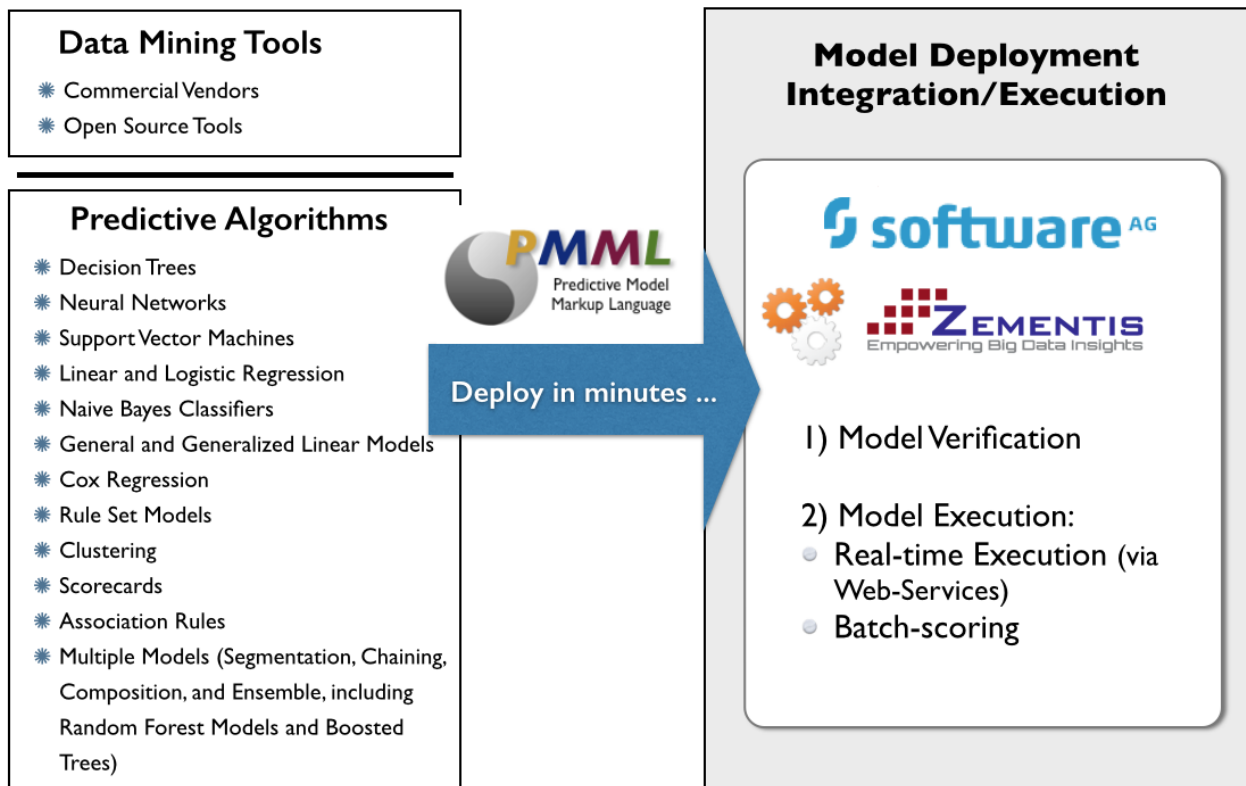


PMML is capable of representing not only a model ensemble but also model composition, segmentation, chaining and cascade. The same is true for ZEMENTIS, which consumes and executes PMML files containing multiple models. With ZEMENTIS and PMML, after a model is built, export it as (or convert it to) PMML, upload it to ZEMENTIS, and start scoring right away.

ZEMENTIS makes the task of verifying a model extremely easy. After a model gets uploaded in ZEMENTIS, a test data file containing the expected results can be uploaded so that the necessary validation can be performed, before the model is actually used to score new data. When presented with a scored data file, ZEMENTIS will automatically operate in score-matching test mode. In this mode, ZEMENTIS will compare expected scores against computed scores for each data record and warn the user if any mismatches are found.

The overall process of model building, using a commercial or open-source data mining tool as well as model deployment, verification, and execution is depicted in [Figure 2.2](#). In the next sections, we elaborate on each phase of this process in more detail.

Figure 2.2. Predictive Modeling Process



2.1. Predictive Model Building Process

The process of creating predictive models starts by defining a clear business goal that needs to be achieved. This is followed by the data analysis phase in which the data scientist mines historical data looking for all the pieces deemed necessary for model building. Data is usually processed and feature detectors are created before a predictive algorithm such as a neural network is trained. Data analysis, model building and model validation is usually performed within the scientist's desktop through the use of an array of tools and scripts. Today, the leading statistical packages

are able to export models in PMML, the language recognized by ZEMENTIS. Examples of such statistical packages are [IBM SPSS](#), [SAS](#), [R](#), and [KNIME](#). For a more comprehensive list of tools that support the PMML standard, check the [Powered by PMML](#) on the [Data Mining Group \(DMG\)](#) web site.

Besides this guide and as part of the overall documentation for ZEMENTIS, a number of sample models represented in PMML format are also available for inspection and use. Our sample models provide the PMML files listed in [Table 2.1](#). These models were obtained from a variety of datasets, including the Iris, Heart, Audit and Diabetes datasets. We use three of the sample models built with the Iris dataset to showcase the power of web services through a series of examples. These are featured in the code shown in [Chapter 5](#) and in [Chapter 6](#).

The Iris classification problem is one of the most famous data mining problems and datasets. It involves determining the class of an Iris plant given the length and width of its sepal and petal. Possible classes are: *setosa*, *virginica*, and *versicolor*. The models built with the Iris dataset not only predict the class with the highest probability, but also output the probabilities for each of the three classes. For more on the Iris dataset and for further information on the Heart Disease dataset, please refer to Bache, K. and Lichman, M. (2013). UCI Machine Learning Repository [<http://archive.ics.uci.edu/ml>]. Irvine, CA: University of California, School of Information and Computer Science.

The Audit dataset is supplied as part of the R Rattle package - <http://rattle.togaware.com> (it is also available for download as a CSV file from <http://rattle.togaware.com/audit.csv>). The Audit data set is an artificial dataset consisting of fictional clients who have been audited, perhaps for tax refund compliance. For each case an outcome is recorded (whether the taxpayer's claims had to be adjusted or not) and any amount of adjustment that resulted is also recorded.

The Diabetes dataset consists of ten physiological variables (age, sex, weight, blood pressure ...) measure on 442 patients, and an indication of disease progression after one year. The goal is to predict disease progression from the given physiological variables. For more information on the Diabetes dataset, please refer to Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.

Table 2.1. Sample Predictive Models

File Name	Description
HeartDisease_MS_Classification.pmm1	A multiple model implementing model segmentation and used to predict the likelihood of a person developing a heart disease. It contains three different models: a linear regression model, a decision tree and a neural network model. Each predictive model is executed depending on the value of an input field.
TaxAudit_SVM.pmm1	A predictive model composed of a support vector machine used to predict compliance issues with tax returns and the consequent need for them to be audited.

File Name	Description
CustomerChurn_NN.pmml	A predictive model composed of a neural network model used to predict the likelihood of churn, based on attrition, for a company's customer base. This model also defines thresholds and business rules as part of the model's post-processing for implementing a business strategy to mitigate the risk of churn.
Diabetes_RF.pmml	A predictive model composed of a random forest model used to predict diabetes progression for a group of patients. This predictive model is an example of multiple models being used to implement a random forest model (model ensemble).
ECommerceFraud_NN.pmml	A predictive model composed of a neural network model used to predict the likelihood of fraud for e-commerce transactions. This model requires the use of custom functions for some of its data pre-processing, which are made available through the file "custom.jar". It also requires a lookup table, which can be found in the "customerStateMappingTable.xls" file. Both files are available as custom resource files (see Table 2.2 for information on how to locate these files).
Transformations.pmml	This file contains a series of data pre-processing steps. It illustrates how PMML, in conjunction with ZEMENTIS, can be used solely for data manipulation. The results obtained from a PMML file containing transformations can then be used for training a predictive model.
Iris_NN.pmml	A neural network model used to predict the class of Iris flower. This model is used to illustrate the use of web services.
Iris_MLR.pmml	A multinomial logistic regression used to predict the class of Iris flower. This model is used to illustrate the use of web services.
Iris_CT.pmml	A CART decision tree used to predict the class of Iris flower. This model is used to illustrate the use of web services.

All sample files described here are available to download from the ZEMENTIS Console Help page. In there you will find a link to a compressed file in ZIP format. When uncompressed, this file reveals a number of directories which contain the sample files. [Table 2.2](#) describes how the sample files are organized.

Table 2.2. Directory Structure of Sample Models

Directory	Contents
models	Predictive models (PMML) files: contains the PMML files for all the sample solutions.

Directory	Contents
resources	Custom resource files: contains custom functions (JAR file) and a lookup table for model "ECommerceFraud_NN.pmml". Upload these resource files in ZEMENTIS before uploading the PMML model file.
data	Scored data files: contains the scored data files in CSV format for model execution for all the sample predictive models. Score a data file in ZEMENTIS against its respective model in order to perform the score matching test. Each data file is named according to its respective PMML file. In this case, if the PMML file is "Diabetes_RF.pmml", the data file is "Diabetes_RF.csv".
ws-client	Source and build files for sample Java clients to ZEMENTIS Web Services.
rest-client	Source and build files for sample Java client to ZEMENTIS REST API.

2.2. Deploy and Test Predictive Models

Once your models are built and expressed in PMML, it is extremely easy to deploy them in ZEMENTIS. Managing and deploying models can be accomplished through the use of the ZEMENTIS Console.

2.2.1. Deploying Models

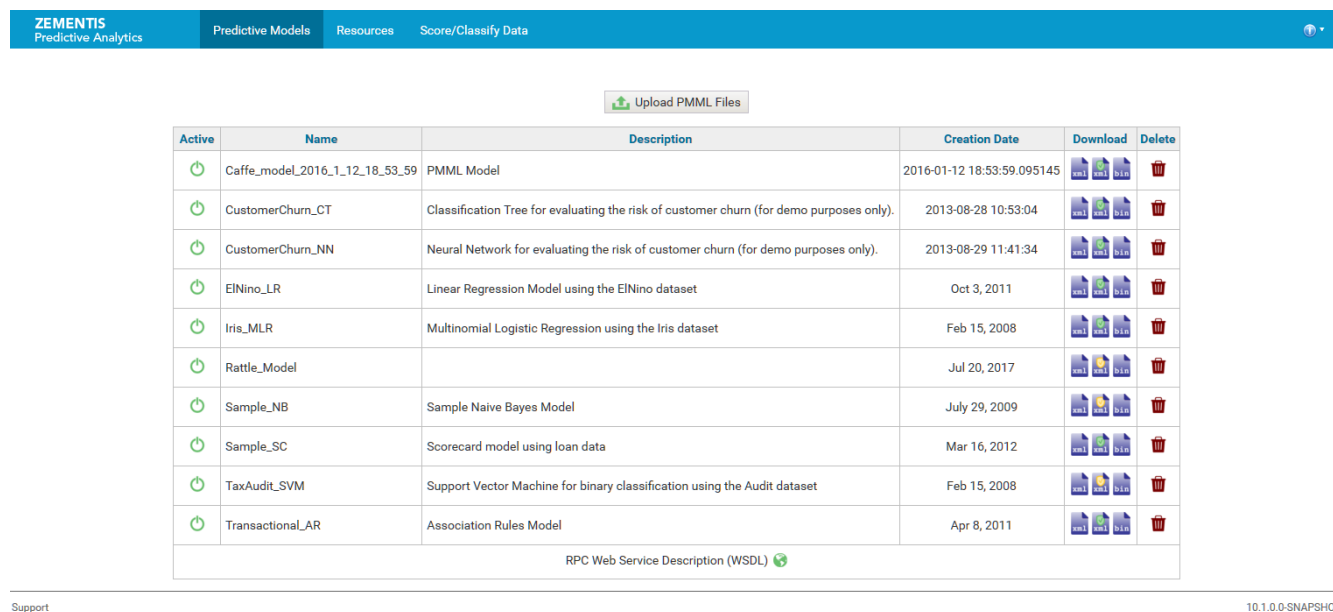
Models are deployed in ZEMENTIS by uploading them directly in the ZEMENTIS Console. Although a data mining tool may export an older version of PMML, ZEMENTIS will automatically perform comprehensive syntactic and semantic checks, correct known issues and convert your PMML file to version 4.3 when the `Enable validation and correction on PMML file(s)` checkbox is checked. By default, the `Enable validation and correction on PMML file(s)` checkbox is checked. Unchecking the checkbox will improve upload time, but this is only recommended for annotated PMML files that are generated after being processed by ZEMENTIS. The annotated PMML file for a model can be downloaded by clicking the middle icon in the "Download" column of the corresponding model name. The yellow shield indicates potential issues with a PMML file that may need to be reviewed. The detailed warning messages are available in the annotated PMML file as comments at appropriate locations. 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 as, in some cases, they may have an impact on the scoring process. The green shield indicates that the PMML file was uploaded without any warnings or errors.

Tip

If the PMML file is large, such as the Random Forest model, we recommend compressing the file using ZIP/GZIP before uploading. This will reduce the upload time dramatically.

If you had previously uploaded models into ZEMENTIS, those models would be listed in the ZEMENTIS Console Predictive Models page. [Figure 2.3](#) shows the ZEMENTIS Console after uploading the sample predictive models described in [Table 2.1](#).

Figure 2.3. Predictive Models in the ZEMENTIS Console



Active	Name	Description	Creation Date	Download	Delete
	Caffe_model_2016_1_12_18_53_59	PMML Model	2016-01-12 18:53:59.095145		
	CustomerChurn_CT	Classification Tree for evaluating the risk of customer churn (for demo purposes only).	2013-08-28 10:53:04		
	CustomerChurn_NN	Neural Network for evaluating the risk of customer churn (for demo purposes only).	2013-08-29 11:41:34		
	ElNino_LR	Linear Regression Model using the ElNino dataset	Oct 3, 2011		
	Iris_MLR	Multinomial Logistic Regression using the Iris dataset	Feb 15, 2008		
	Rattle_Model		Jul 20, 2017		
	Sample_NB	Sample Naive Bayes Model	July 29, 2009		
	Sample_SC	Scorecard model using loan data	Mar 16, 2012		
	TaxAudit_SVM	Support Vector Machine for binary classification using the Audit dataset	Feb 15, 2008		
	Transactional_LAR	Association Rules Model	Apr 8, 2011		

[RPC Web Service Description \(WSDL\)](#)

Support

10.1.0.0-SNAPSHOT

For more information on how to upload your models through the ZEMENTIS Console, see the [Help page](#).

2.2.2. Testing Models

Given that models are built with different tools, you need to make sure that both ZEMENTIS and the model development environment produce exactly the same results during scoring.

ZEMENTIS provides an integrated testing process to make sure your model was represented accurately, uploaded correctly, and works as expected. This is also done through the ZEMENTIS Console which allows for a model verification data file to be uploaded for score matching. This file should be in Comma Separated Values (CSV) format containing one record per line (for more information on how to format your CSV file for scoring, please refer to the [Zementis support forum](#)). Each record should have values for all the input variables along with at least one of the output variables. The values for the output variables serve as the expected predicted values. ZEMENTIS will compute new predicted values and compare them to the expected ones. If all the values match, the model is considered production-ready, i.e. ready for scoring. If not, ZEMENTIS offers execution trace details to facilitate trouble shooting.

The sample predictive models ([Table 2.1](#)) provide CSV files that can be used for testing their respective PMML files. For more information on how to test models, see the ZEMENTIS Console Help page or the [Zementis support forum on model verification](#).

PMML also offers a "ModelVerification" element for similar testing purposes. In this way, verification records are part of the PMML file itself. Given that ZEMENTIS supports this element, there is more than one way to test models. For more information on this specific PMML element, please refer to the [Data Mining Group \(DMG\)](#) web site or to the book [PMML in Action \(2nd Edition\): Unleashing the Power of Open Standards for Data Mining and Predictive Analytics](#) by Alex Guazzelli, Wen-Ching Lin, and Tridivesh Jena, which is available for purchase on Amazon.com.

2.3. Data Scoring and Classification

Bulk scoring in batch mode can be easily performed through the ZEMENTIS Console, using the same process as for model testing. First, select the target model and then upload a data file in CSV format. The only difference between this process and the score-matching test is that in the present case, the predicted field and its expected scores are not part of the data file. ZEMENTIS will process the uploaded file and return a new file with your original data expanded with an extra column containing the predicted variable and the scores/results for each row. For more details on how to format your data file for batch scoring in ZEMENTIS, please refer to the [Zementis support forum on data formatting](#)

Tip

If the data file is large, Software AG suggests compressing the file in ZIP format before uploading. This reduces the upload time dramatically. In this case, ZEMENTIS also returns a compressed file containing the results.

Real-time scoring allows other applications to get and use predictions on demand from anywhere in your enterprise. With ZEMENTIS this can be achieved through standard web service calls. Details on using web services can be found in [Chapter 5](#).

2.4. Other Data Sources

ZEMENTIS also supports applying predictive models to a wide variety of data sources, such as images, audio files, videos, binary feeds or even text files as input data. In ZEMENTIS, with binary input definition and proper custom functions which convert unstructured data into structured data, the data type of the input source in the deployed model could be in any format for analytics process. Details on how to apply models to binary data source can be found in [Section 3.4](#).

Chapter 3. Custom Resources

Predictive models may require external resources such as custom functions, look-up tables or training data tables. Files containing such resources can be uploaded in ZEMENTIS using the ZEMENTIS Console. Note, such resources should be uploaded before any models that depend on them. Also, deleting a resource file will remove all the resources contained in it from ZEMENTIS. In this case, first the model that is dependent on the resource should be deleted.

3.1. Custom PMML 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 which 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. The predictive model `ECommerceFraud_NN.pmml`, provided as part of the sample models, uses several custom functions to compute the average transaction amount as well as the transaction velocity for a period of time. This model is described in [Table 2.1](#)

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.

3.1.1. Create Custom PMML Functions

Custom functions are implemented as 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. In addition, the types of the method parameters as well as its return type must be compatible with the PMML data types. An example of such a function is shown in [Figure 3.1](#).

Figure 3.1. Custom PMML Function Example

```

package com.company.udf;

import com.zementis.stereotype.PMMLFunctions;

@PMMLFunctions(namespace = "company")
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, the class `RecursiveFunctions` has been annotated with the `@PMMLFunctions` annotation. This annotation informs ZEMENTIS that the class contains methods which may be used as PMML functions. The parameter `namespace` defines a namespace for the functions defined in this class. Namespaces prevent conflicts between function names. Within PMML, the namespace is used as a prefix for the name of the custom function. For example, the PMML name of the function implemented by the Java method `factorial` in [Figure 3.1](#) would be `company:factorial`.

The namespace does not have to be unique for each class. Multiple classes may specify the same namespace. This would allow, for example, creating the notion of a function library where functions spread across multiple class files are grouped under one namespace. In this scenario, extra care needs to be taken so that there are no ambiguities between function names located in different classes.

Within each annotated class, only methods that are declared as `public` and `static` can be used as PMML functions. In addition, a method should accept parameters and return values compatible with the PMML data types. [Table 3.1](#) provides the Java primitive types and classes that correspond to the different PMML data types.

Table 3.1. PMML and Java types in ZEMENTIS

PMML Data Type	Java Primitive Type	Java Class
boolean	boolean	java.lang.Boolean
date		org.joda.time.LocalDate
dateTime		org.joda.time.DateTime
double	double	java.lang.Double

PMML Data Type	Java Primitive Type	Java Class
float	float	java.lang.Float
integer	long	java.lang.Long
string		java.lang.String
time		org.joda.time.LocalDateTime
binary		java.io.InputStream
binary (buffered)	byte[]	byte[]

The method return type must be one of the Java types listed in the table. Note that methods declared as `void` cannot be used as PMML functions. 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`). Finally, methods can also declare variable number of parameters (`varargs`).

Caution

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

3.1.2. Use Custom PMML Functions

To make custom functions available to ZEMENTIS, compile the corresponding classes into a JAR file and upload it using the ZEMENTIS Console. To compile a class using the `@PMMLFunctions` annotation, include the `ada-pa-api-10.1.0.0.jar` file in the classpath. This file is included in the ZEMENTIS distribution package as well as the provided package of sample files.

Once deployed, custom functions can be used exactly like the built-in functions within `Apply` transformations. Please make sure you use the fully qualified name of the custom function, i.e. prefix the function name with the appropriate namespace. The PMML fragment in [Figure 3.2](#) contains a simple example that uses the function defined in [Figure 3.1](#).

Figure 3.2. Example Using a Custom Function in PMML

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

3.1.3. 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 3.3. 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 = "company")
class CustomFunctions {

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

3.2. External Lookup Tables

Predictive models can sometimes require the use of lookup tables. If relatively small and static, these tables can be easily embedded within the PMML file itself. However, if they are fairly large and/or they are modified frequently, it is more practical to create and manage them separately. ZEMENTIS supports external lookup tables and their seamless integration with predictive models.

As an example of a simple lookup table, suppose a model makes use of a country's GDP (Gross Domestic Product). That requires the ability to look up the GDP by country name. Such a simple lookup table is shown in [Figure 3.4](#).

Given an input country, say Taiwan, the row in the lookup table which has `Taiwan` in its first column maps it to a GDP of `576.20`. Being an example, we show only a few mappings; in reality, we can imagine similar cases with hundreds and even thousands of mappings.

Figure 3.4. Lookup Table Example

Find Country	Set GDP
Afghanistan	21.50
Brazil	1,492.00
Canada	1,023.00
China	7,262.00
Egypt	316.30
Germany	2,362.00
Greece	226.40
India	3,319.00
Iraq	54.40
Morocco	134.60
Switzerland	251.90
Taiwan	576.20
US	11,750.00

The predictive model `ECommerceFraud_NN.pmml`, provided as part of the sample models, uses a lookup table to retrieve the number of points for each US state. This model is described in [Table 2.1](#)

3.2.1. Create Lookup Tables in Excel

ZEMENTIS supports lookup tables implemented in Excel files. In this section, we describe the structure of such tables. In general, a lookup table has one or more input variables and an output variable. The intended functionality is that any set input values can be looked up to retrieve the corresponding output value, if one is found. [Figure 3.5](#) shows a slightly expanded version of the previous example. Here, we have two input variables, `Country` and `State`. The output variable is `GDP`.

Figure 3.5. Sample Excel Lookup Table

LookupTable	GDPTable	
input	Country	String
input	State	String
output	GDP	Double
Country	State	GDP
Afghanistan		21.50
Brazil		1492.00
Canada		1023.00
China		7262.00
Egypt		316.30
Germany		2362.00
Greece		226.40
India		3319.00
US	California	557.37
US		11750.00

A single Excel file may contain one or more lookup tables. However, only one lookup table is allowed per worksheet. Multiple tables should be arranged in separate worksheets. Within a worksheet, the beginning of a lookup table is identified by the keyword `LookupTable`. The name of the table should appear in the cell right next to this keyword (`GDPTable` in this example). The definitions of the input and output variables start in the cell right below the `LookupTable` keyword. Variables must be listed one per row, with the output variable listed last. For each variable, provide the usage (`input` or `output`), the name and the data type. The variable names must be unique. The allowed types of data are `Integer`, `Long`, `Double`, `Float`, `Boolean` and `String`, corresponding to the Java primitive types. In this example, the first row defines an input variable called `Country` which is of type `String`. The next row defines an input variable `State`, again of type `String`. Finally, the output variable is called `GDP` which is of type `Double`.

The data area of the lookup table starts right below the output variable definition. In the simple form shown here, this area consists of one column per variable. The first is the header row, where the name of the corresponding variable is listed. All the following rows contain combinations of input and output values. Each row represents a mapping from the input values to the output value. Note that empty cells are allowed. For an input variable, an empty cell represents any value. For an output variable, an empty cell represents no value (or a `null` value). A fully empty row, i.e., a row with empty cells for all the variables marks the end of the table. Anything below a fully empty row is ignored.

Duplicate mappings are not allowed. However, with empty cells representing any value, overlapping mappings are possible (and allowed). To illustrate this, please consider the overlapping mapping in last two rows of the example in Figure 3.5. The second to last row implies that if the country is USA and the state is CA then the GDP is 557.37. However, the last row implies that if the country is USA, the GDP is 11750.00 no matter what the state is. In the presence of overlapping mappings, the tighter mapping, i.e. the mapping with more matching input values, prevails. In the current example, this means that the a GDP lookup for CA will result in 557.37 and a GDP lookup for any other state will be 11,750.00

In some cases, it is desirable to arrange some mapping as a cross tab. Such an example is shown in Figure 3.6 where the probability of child obesity can be looked up by child age and group. The probabilities for all the combinations of four child groups (Rural Girls, Urban Girls, Rural Boys, and Urban Boys) and six different ages (10 through 15) are presented.

Figure 3.6. A LookupTable with two inputs and one output

LookupTable	ChildObesity					
input	Group	String				
input	Age	Integer				
output	ObesityProbability	Double				
Age	10	11	12	13	14	15
Group	ObesityProbability					
Rural Girls	0.0058	0.0116	0.0566	0.0309	0.0174	0.0000
Urban Girls	0.0550	0.0570	0.0467	0.0650	0.0420	0.0526
Rural Boys	0.0222	0.0333	0.0294	0.0411	0.0118	0.0384
Urban Boys	0.0730	0.0730	0.0745	0.0627	0.0668	0.0117

The structure of a cross tab lookup table is similar to the previous one. The only difference is that the values for one or more of the input variables are listed horizontally above the header of the data area, as opposed to vertically. Note that not all input variables can be listed horizontally. At least one must be listed vertically. In addition, the header cell containing the name of the output variable must span all the data columns. Similarly to the previous case, the boundaries of the lookup table are identified by the first fully empty row and the first fully empty column.

Our sample solution provides a lookup table in the Excel file `borrowerStateMappingTable.xls`. This table is used by the demo PMML model for fixed rate loans.

3.2.2. Use Lookup Tables in PMML

In PMML, lookup tables can be used within `MapValues` transformations and the `TableLocator` mechanism. In the following sample PMML snippet, the lookup table `ChildObesity` is used to retrieve the appropriate child obesity probability.

```
<LocalTransformations>
  <DerivedField name="obesityProbability" dataType="double" optype="continuous">
    <MapValues outputColumn="Probability" defaultValue="0.5" mapMissingTo="0">
      <FieldColumnPair column="Age" field="childAge" />
      <FieldColumnPair column="Group" field="childGroup" />
      <TableLocator>
        <Extension extender="ADAPA" name="TABLE_NAME" value="ChildObesity" />
      </TableLocator>
    </MapValues>
  </DerivedField>
</LocalTransformations>
```

The table used in the mapping is identified in the `Extension` element. The value attribute of this element contains the name of the lookup table to use. The rest of the structure details what fields of the model (`childAge`, `childGroup`, and `childObesity`) correspond to what columns (`Age`, `Group`, and `Probability`) of the lookup table.

3.3. External Training Data Tables

Some algorithms (e.g. K Nearest Neighbor) expect a table of training data as part of the model. This table can be included in the PMML document, or loaded as an external resource in CSV format. The format of the external table is identical to the one of the test data offered in the samples directory. This file should be in CSV format containing one record per line (for more information on how to format your CSV file, please refer to the [Zementis support forum](#)). Each record should have values for all the input variables along with the predicted values.

```
<TrainingInstances>
  <InstanceFields>
    <InstanceField field="Sepal.Length" column="Sepal.Length"/>
    <InstanceField field="Sepal.Width" column="Sepal.Width"/>
    <InstanceField field="Petal.Length" column="Petal.Length"/>
    <InstanceField field="Petal.Width" column="Petal.Width"/>
    <InstanceField field="Species" column="Species"/>
  </InstanceFields>
  <TableLocator>
    <Extension extender="ADAPA" name="TRAINING_INSTANCES_NAME" value="Iris_KNN.csv" />
  </TableLocator>
</TrainingInstances>
```

The table is identified in the `Extension` element. The value attribute of this element contains the name of the training data table to use including the file ending. The `InstanceFields` element details one to one correspondence between the field of the model and the column of the table.

3.4. Binary Sources

Some predictive models use binary data as input for scoring or classifying results. ZEMENTIS supports applying models to binary data that utilizes an external custom function to transform unstructured data into readable data. Given proper binary input definition and custom function deployed in ZEMENTIS, the input binary data can be

seamlessly integrated to the scoring/classifying process. This section provides examples of how to define a binary source in PMML and how to create a corresponding custom function that converts the provided binary data.

3.4.1. Using Default binary Type

An example shown in [Figure 3.7](#) represents the definition of `binary` type input. Set the data type as `binary` in `<DataField>` element, ZEMENTIS can process the provided file as a single binary record named `field1`. It is recommended to provide MIME type in `<Extension>` element, for example `image/jpeg`, ZEMENTIS will help on verifying the data format before starting the scoring/classifying process in order to avoid the data type mismatch problem.

Figure 3.7. Binary DataType Example

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

Note

ZEMENTIS supports single binary file upload. Please make sure the `numberOfFields` is 1 and there is only one `<DataField>` element in `<DataDictionary>`.

Here are the steps to create a corresponding custom function:

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

The custom function can be compatible with the PMML data type of `field1` defined in PMML fragment [Figure 3.7](#). An example of custom function that converts a 28 x 28 pixel image into readable data is shown in [Figure 3.8](#).

Figure 3.8. Custom Function of Binary Data Example

```

package com.company.udf;

import java.io.InputStream;
import javax.imageio.ImageIO;
import org.canova.image.loader.ImageLoader;
import org.nd4j.linalg.api.ndarray.INDArray;
import com.zementis.stereotype.PMMLFunctions;

@PMMLFunctions(namespace = "company")
class CustomFunctions {
    private static char comma = ',';
    private static double factor = 1.0 / 255;

    public static String getRowVector(InputStream inputStream) {
        StringBuilder stringBuilder = new StringBuilder();
        try {
            ImageLoader imageLoader = new ImageLoader(28, 28);
            INDArray array = imageLoader.asRowVector(ImageIO.read(inputStream));
            for (int i = 0; i < array.rows(); i++) {
                for (int j = 0; j < array.columns(); j++) {
                    stringBuilder.append(array.getDouble(i, j));
                    if (i != array.rows() - 1 || j != array.columns() - 1) {
                        stringBuilder.append(comma);
                    }
                }
            }
        } catch (Exception e) {
            return null;
        }
        return stringBuilder.toString();
    }

    public static double getValue(String imageRowVectorStringValue, Long index) {
        return factor * parseDoubleFromString(imageRowVectorStringValue, index.intValue());
    }

    private static double parseDoubleFromString(String s, int index) {
        int from = 0;
        int to = 0;
        int current = 0;
        if (index == 0) {
            to = s.indexOf(comma);
            return Double.parseDouble(s.substring(from, to));
        } else {
            while (current < index) {
                from = (current == 0) ? s.indexOf(comma) : to;
                to = s.indexOf(comma, from + 1);
                current++;
            }
            to = (to == -1) ? s.length() : to;
            return Double.parseDouble(s.substring(from + 1, to));
        }
    }
}

```

Once the custom function in [Figure 3.8](#) being compiled and deployed, `getRowVector` and `getValue` can be used exactly like a built-in function within `Apply` transformations. The PMML fragment in [Figure 3.9](#) contains a simple example that uses the function defined in [Figure 3.8](#).

Figure 3.9. Example Using Custom Function of Binary Data in PMML

```

<LocalTransformations>
  <DerivedField name="field2" optype="categorical" dataType="string">
    <Apply function="company:getRowVector">
      <FieldRef field="field1"/>
    </Apply>
  </DerivedField>
</LocalTransformations>
<NeuralInputs numberOfInputs="784">
  <NeuralInput id="1">
    <DerivedField dataType="double" name="V1" optype="continuous">
      <Apply function="company:getValue">
        <FieldRef field="field2"/>
        <Constant>0</Constant>
      </Apply>
    </DerivedField>
  </NeuralInput>
  <NeuralInput id="2">
    <DerivedField dataType="double" name="V2" optype="continuous">
      <Apply function="company:getValue">
        <FieldRef field="field2"/>
        <Constant>1</Constant>
      </Apply>
    </DerivedField>
  </NeuralInput>
  ...
</NeuralInputs>

```

3.4.2. Using Buffered binary Type

ZEMENTIS provides two ways to manage binary data. The provided binary data can be retrieved as either a `java.io.InputStream` object or a `byte[]`. The types of data are listed in [Table 3.1](#). ZEMENTIS processes binary data using `java.io.InputStream` as default java type. Since a non-markSupported `java.io.InputStream` object cannot be read more than once, there is an additional option to enable binary data stored in a `byte[]` for the case that models use binary data multiple times during process. Set `BINARY_BUFFERED` as `true` in `<Extension>` element like the PMML fragment in [Figure 3.10](#) to guarantee the binary data will not be `null` after being consumed.

Figure 3.10. 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 static method of Java class.
- Annotate it with 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 3.10](#). An example of custom function is shown in [Figure 3.11](#). It is similar to the example in [Figure 3.8](#), but the difference is the method parameter which is changed to Java primitive type `byte[]`.

Figure 3.11. Custom Function of Buffered Binary Data Example

```
package com.company.udf;

import com.zementis.stereotype.PMMLFunctions;

@PMMLFunctions(namespace = "company")
class CustomFunctions {

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

Once the custom function in [Figure 3.11](#) being compiled and deployed , `convert` can be used exactly like a built-in function within `Apply` transformations. The PMML fragment in [Figure 3.12](#) contains a simple example that uses the function defined in [Figure 3.11](#).

Figure 3.12. Example Using Custom Function of Buffered Binary Data in PMML

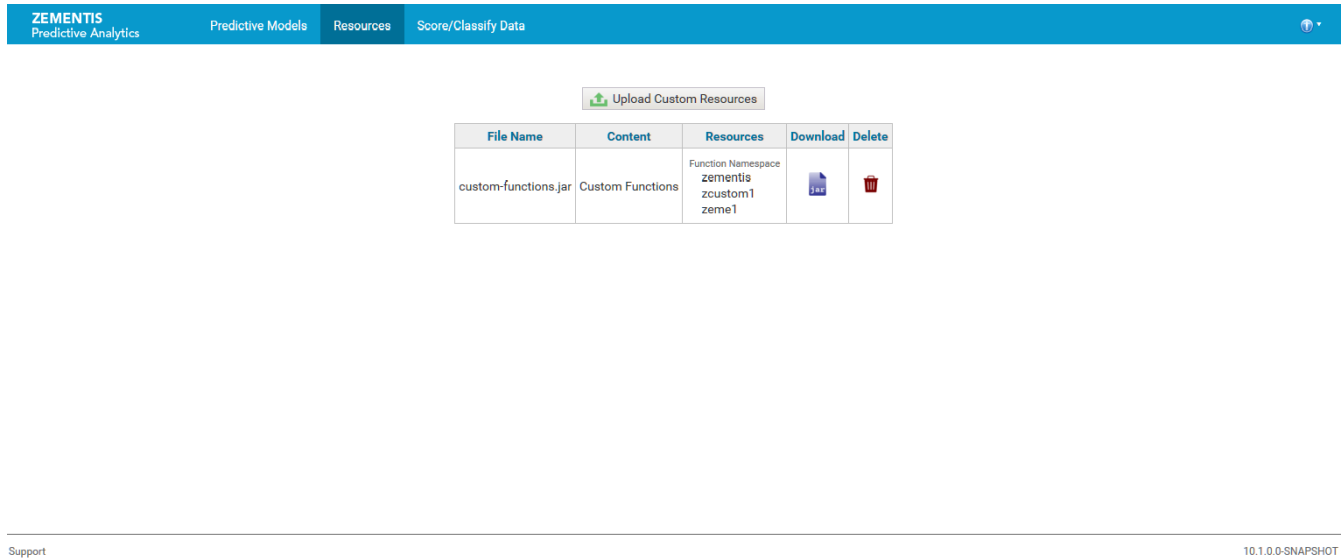
```
<DerivedField name="field2" optype="categorical" dataType="string">
  <Apply function="company:convert">
    <FieldRef field="field1"/>
  </Apply>
</DerivedField>
```

3.5. Deploy Resources

Custom PMML functions or lookup tables are deployed in ZEMENTIS by simply uploading them directly in the ZEMENTIS Console.

If you have previously uploaded any resource files into ZEMENTIS, these are shown in the ZEMENTIS Console as a list. [Figure 3.13](#) shows the ZEMENTIS Console after the uploading of the lookup table and custom functions (JAR file) used by predictive model `ECommerceFraud_NN.pmml` (for more details on this sample mode, see [Table 2.1](#)).

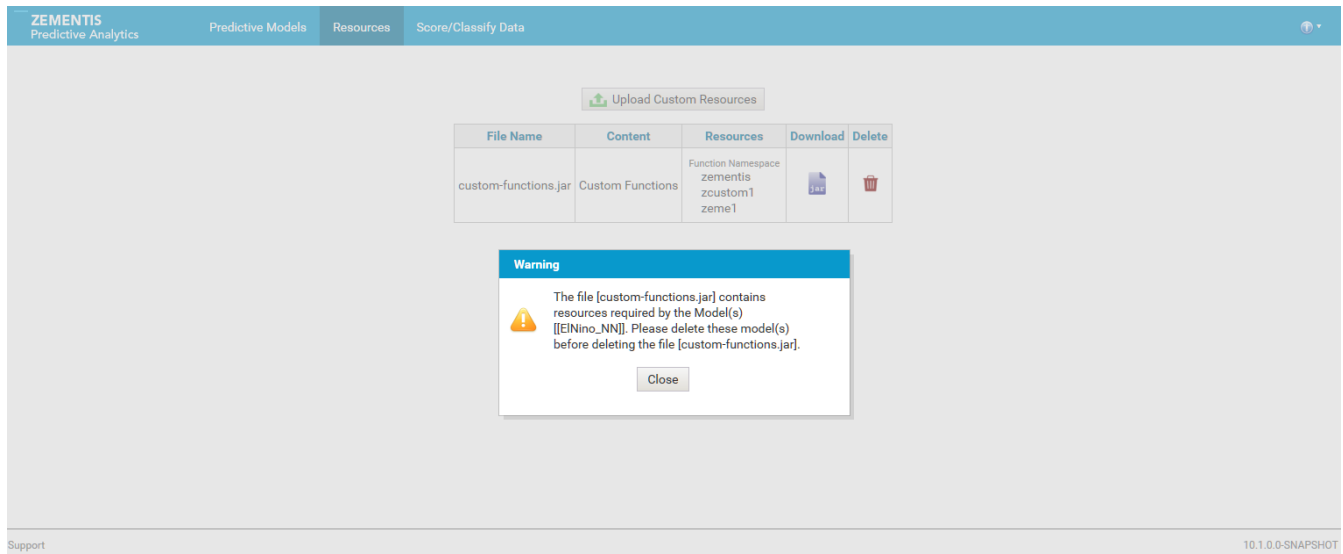
Figure 3.13. Resource Files in the ZEMENTIS Console



3.5.1. Deleting Resources

When deleting a resource file which is a downstream dependency of one of the models from the models list, you must first delete the model and then delete the resource. [Figure 3.14](#) shows the ZEMENTIS Console when an exception is thrown.

Figure 3.14. Resource dependency exception in the ZEMENTIS Console



Chapter 4. Extensions API

ZEMENTIS has been designed to easily support customizations and/or extensions needed to meet the requirements imposed by the target environment. Using the popular [Spring Framework](#), it allows injecting external resources either as configuration modifications or as extensions. This means that ZEMENTIS can be customized by providing an appropriate Spring context file along with the necessary custom implementations and required libraries. In the following sections, the ZEMENTIS Java Extensions API is described, which can be implemented to provide custom resources (Custom Functions and Lookup Tables), custom asset repository and a custom logging store for ZEMENTIS server.

4.1. Using the ZEMENTIS Extensions API

Using ZEMENTIS Extensions API, you can provide a custom implementation for the following:

- Custom Function
- Lookup Table
- Asset Repository
- Logging Store

The following sections will describe each of these items in detail. [Section 4.2](#) will provide details about how the Extensions API and sample implementations are packaged with the `adapa-app-10.1.0.0.zip` distribution.

4.1.1. Custom Function

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.

The `Function<T>` interface represents a custom function which can be called from PMML. This function can be referenced by the name returned by the `getName()` method and it operates on the arguments provided in the `evaluate(Object...)` method. It returns a value of the specified type `T`. A sample implementation of this interface is contained in `CalcSomething.java` which demonstrates a custom function that can operate on several (at least 2) numeric arguments and returns a value of type `Double`.

The `Function.Factory` interface provides a factory method for creating `Function` instances with the method `createFunction(String functionName, Class<?> ... argumentTypes)`. The `Function.getName()`

method must match parameter functionName and Function.evaluate(Object...) must be able to operate on parameter argumentTypes. A sample implementation of this interface is contained in CalcSomethingFactory.java which creates functions that can operate over a variable number (but at least two) of numeric arguments.

Please add the following dependencies as listed under [Figure 4.1](#) when packaging the project as a JAR. Make sure \${project.version} resolves to 10.1.0.0.

Figure 4.1. Dependencies for Custom Functions

```
<dependencies>
  <dependency>
    <groupId>com.zementis.adapa</groupId>
    <artifactId>adapa-extensions</artifactId>
    <version>${project.version}</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>com.zementis.adapa</groupId>
    <artifactId>adapa-api</artifactId>
    <version>${project.version}</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>com.zementis.adapa</groupId>
    <artifactId>adapa-bundle</artifactId>
    <version>${project.version}</version>
  </dependency>
</dependencies>
```

4.1.2. Lookup Table

Predictive models can sometimes require the use of lookup tables. If relatively small and static, these tables can be easily embedded within the PMML file itself. However, if they are fairly large and/or they are modified frequently, it is more practical to create and manage them separately.

The LookupTable interface represents a lookup table that can be called from PMML. This lookup table can be referenced by the name returned by the getName() method. The lookup table implementation can be used to retrieve an output value identified by column name with getOutputColumnName(). This can be done by looking up provided input values which are identified by column names with getInputColumnNames(). The order of input values for the lookup(Object...) method must match the order of column names returned by the getInputColumnNames() method. A sample implementation of this interface is contained in GDPLookupTable.java which returns a GDP number corresponding to two inputs, Country and State by querying a database table.

Please add the following dependencies as listed under [Figure 4.2](#) when packaging the project as a JAR. Make sure \${project.version} resolves to 10.1.0.0.

Figure 4.2. Dependencies for Lookup Table

```
<dependencies>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.6</version>
  </dependency>
  <dependency>
    <groupId>com.zementis.adapa</groupId>
    <artifactId>adapa-extensions</artifactId>
    <version>${project.version}</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>com.zementis.adapa</groupId>
    <artifactId>adapa-api</artifactId>
    <version>${project.version}</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>com.zementis.adapa</groupId>
    <artifactId>adapa-bundle</artifactId>
    <version>${project.version}</version>
  </dependency>
</dependencies>
```

4.1.3. Asset Repository

The `AssetRepository` interface provides methods for managing ZEMENTIS assets on a back-end storage. By default, ZEMENTIS uses a file-based repository to store the uploaded artifacts (models and resources). ZEMENTIS also provides support for a database-based repository by using the [Java Persistence API \(JPA\)](#) in conjunction with using [Hibernate](#) as the JPA provider. A traditional Database can be plugged-in as a repository store for ZEMENTIS by providing an appropriate configuration file.

On top of this, ZEMENTIS also allows users to provide a custom back-end store (e.g. [MongoDB](#)) by implementing this interface. A sample implementation is contained in `MongoAssetRepository.java`. As shown in the sample implementation, the `addAsset(Serializable, InputStream)` method requires assignment of a unique identifier to the provided ZEMENTIS asset. The choice of unique identifier is left to the implementor. The implementation of this interface needs to be in the classpath of ZEMENTIS library along with any required JDBC drivers.

Please add the following dependencies as listed under [Figure 4.3](#) when packaging the project as a JAR. Make sure `${project.version}` resolves to 10.1.0.0.

Figure 4.3. Dependencies for Asset Repository

```
<dependencies>
<dependency>
  <groupId>org.mongodb</groupId>
  <artifactId>mongodb-driver</artifactId>
  <version>3.3.0</version>
</dependency>
<dependency>
  <groupId>commons-io</groupId>
  <artifactId>commons-io</artifactId>
  <version>2.5</version>
</dependency>
<dependency>
  <groupId>commons-lang</groupId>
  <artifactId>commons-lang</artifactId>
  <version>2.6</version>
</dependency>
<dependency>
  <groupId>com.zementis.adapa</groupId>
  <artifactId>adapa-extensions</artifactId>
  <version>${project.version}</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>com.zementis.adapa</groupId>
  <artifactId>adapa-api</artifactId>
  <version>${project.version}</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>com.zementis.adapa</groupId>
  <artifactId>adapa-bundle</artifactId>
  <version>${project.version}</version>
</dependency>
</dependencies>
```

4.1.4. Logging Store

Information about records processed by ZEMENTIS can be logged in a file system or database. The captured data includes input and output values as well as information regarding invalid and missing values presented to the model for execution. The logging mechanism can be enabled and configured for file-based or database store by providing an appropriate Spring configuration file as described in ADAPA Deployment Guide.

On top of this, ZEMENTIS also allows users to provide a custom logging store by implementing the `ModelLogHandler` interface. This interface represents a handler for logging records that a model processes. This interface can be implemented to log entire records, invalid values and missing values. A sample implementation of this interface is contained in `FileLogHandler.java`. This implementation logs every record to a file as soon as the record is processed. The implementation also logs a counter for missing and invalid values for a given field. The logging of missing and invalid values is done when method `flush()` is invoked.

Note

The implementor is responsible for the invocation of `flush()` and for ensuring the thread safety of any state which is maintained before `flush()` is invoked. The code samples are for illustration purposes only.

The `ModelLogHandler.Factory` interface provides a factory method for creating `ModelLogHandler` instances. A sample implementation of this interface is contained in `FileLogHandlerFactory.java`.

Please add the following dependencies as listed under [Figure 4.4](#) when packaging the project as a JAR. Make sure `${project.version}` resolves to `10.1.0.0`.

Figure 4.4. Dependencies for Logging Repository

```
<dependencies>
<dependency>
  <groupId>com.zementis.adapa</groupId>
  <artifactId>adapa-extensions</artifactId>
  <version>${project.version}</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>com.zementis.adapa</groupId>
  <artifactId>adapa-api</artifactId>
  <version>${project.version}</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>com.zementis.adapa</groupId>
  <artifactId>adapa-bundle</artifactId>
  <version>${project.version}</version>
</dependency>
</dependencies>
```

4.2. Overview of code examples

The files under directory `adapa-extensions/samples` offer Java code examples for each use case. [Table 4.1](#) describes all the sample files in detail.

Table 4.1. Directory structure of code examples

Directory	Files	Description
customfunction	<code>applicationContext.xml</code>	The application context XML file to be included.
	<code>CalcSomething.java</code>	The <code>CalcSomething</code> function calculates something over several numeric arguments. In order to support <code>Double</code> , <code>Float</code> , and <code>Long</code> arguments, it uses reflection to enable invocation of the appropriate "doubleValue" method at runtime.

Directory	Files	Description
	<code>CalcSomethingFactory.java</code>	Example of a custom function factory which creates functions that can compute something over a variable number (but at least two) of numeric arguments.
loghandler	<code>applicationContext.xml</code>	The application context XML file to be included.
	<code>FileLogHandler.java</code>	Contains methods for custom record logging.
	<code>FileLogHandlerFactory.java</code>	Factory for custom record logging.
lookuptable	<code>applicationContext.xml</code>	The application context XML file to be included.
	<code>GDPLookupTable.java</code>	The lookup table returns a GDP number corresponding to Country and State. Country, State and GDP are columns in the database table <code>GDP_Table</code> .
repository	<code>applicationContext.xml</code>	The application context XML file to be included.
	<code>MongoAssetRepository.java</code>	A sample <code>AssetRepository</code> for MongoDB.

4.3. Deployment of ZEMENTIS Extensions

Once the new ZEMENTIS extension is created, the Java code needs to be packaged as a JAR together with all depending libraries. Once the JAR file is created, copy it in the directory `ADAPA_HOME/adapa-lib`. This directory must also contain `adapa-extensions-10.1.0.0.jar` file. The new code can then be integrated into ZEMENTIS by using a Spring configuration file as described in the respective `applicationContext.xml`. This context file needs to be copied to the working directory of the server. One or more context files may be used. In case there are multiple context files, rename them as per the extension it configures (For example, `adapaContextLogging.xml` or `adapaContextRepository.xml`). For configuration purposes and upon start-up, ZEMENTIS will examine any files in the server's working directory following the name pattern `adapaContext*.xml`. Please note that configuration changes through context files require a server restart before they can take effect.

Chapter 5. SOAP/XML Web Services

ZEMENTIS provides the functionality to turn predictive models into decision services that can be easily integrated with other applications that can consume SOAP/XML web services. All models and custom resources that are uploaded onto an ZEMENTIS instance are automatically available to be invoked via a web services call that can be leveraged to process data in real-time or batch mode. In this chapter, we describe these web services and show examples of how they can be invoked from other applications.

ZEMENTIS offers two separate services. The functionality of these services is described in [Table 5.1](#).

Table 5.1. Overview of ZEMENTIS Web Services

Service	Functionality
Models Web Service	Manage predictive models and custom resources, and process data. This service provides operations to import, export, describe, and remove predictive models and custom resources. It also provides operations for processing data against one or more predictive models in real-time and batch mode.
RPC Web Service	Through this service, each predictive model is exposed as a separate remote procedure call (RPC). Every RPC operation is named after the corresponding model with the input/output parameters reflecting the mining schema of that model.

Note

The code examples shown for Models Web Service and RPC Web Service are based on the Iris models described in [Section 2.1](#).

5.1. Web Service Address (URL)

[Table 5.2](#) lists the address of each of these services along with the corresponding WSDL files. In this table, *ZEMENTIS-BASE-URL* refers to the base address (URL) of the ZEMENTIS instance. This is of the form `http://HOSTNAME:PORT` where *HOSTNAME* is name of the server where ZEMENTIS is deployed and *PORT* is port number of the web service. If your server has been configured to use [SSL](#), you should use `https` instead of `http` as the URL protocol (prefix). Note also that if the service is using the default port number (80 for HTTP or 443 for HTTPS) the port number can be omitted (together with the preceding colon).

Table 5.2. ZEMENTIS Web Service Addresses (URLs)

Service	Service Address	WSDL Address
Models Web Service	<i>ZEMENTIS-BASE-URL/adapaws/models</i>	<i>ZEMENTIS-BASE-URL/adapaws/models?wsdl</i>
RPC Web Service	<i>ZEMENTIS-BASE-URL/adapaws/rpc</i>	<i>ZEMENTIS-BASE-URL/adapaws/rpc?wsdl</i>

Tip

The exact addresses for the web services of your ZEMENTIS instance can also be found at the bottom of the Help page of the ZEMENTIS Console.

5.2. Models Web Service

The ZEMENTIS Models Web Service allows client applications to remotely manage predictive models and custom resources, and process data in real-time and batch mode. The ZEMENTIS Models Web Service uses different a role authorization for each operation. [Table 5.3](#) provides a description of the operations available with Models Web Service along with the role(s) authorized to perform those operations.

Table 5.3. Operations of ZEMENTIS Web Services

Operation Name	Authorized Role(s)	Description
import	adapa-admin	Import a new model into ZEMENTIS. The request is a PMML file provided as a SOAP attachment. The response contains an import status indication (SUCCESS, WARNING, or ERROR), and a description message. On successful import, the response will also list the name(s) of the model(s) imported. If there are errors or warnings, the response will also provide the source file annotated with explanatory messages as an attachment.
activateModel	adapa-admin	Activate one model. The request provides the name of the model. The response contains information about the specific model, which consists of name, description, input fields, output fields, model upload status (SUCCESS/WARNING), and model runtime (active) status (true/false).
deactivate-Model	adapa-admin	Deactivate one model. The request provides the name of the model. The response contains information about the specific model, which consists of name, description, input fields, output fields, model upload status (SUCCESS/WARNING), and model runtime (active) status (true/false).

Operation Name	Authorized Role(s)	Description
export	adapa-admin	Export the source (PMML file) for a model. The request provides the name of the model. The response contains the source (PMML file) as a SOAP attachment.
exportAnnotated	adapa-admin	Export the annotated source (PMML file) for a model. The annotated source typically contains explanatory messages (including warning messages) describing the syntactic and semantic corrections that are performed by ZEMENTIS on the uploaded model. The request provides the name of the model. The response contains the annotated source (PMML file) as a SOAP attachment.
exportModel	adapa-admin	Export the model as a serialized copy of its in-memory representation. The serialized format can be used to distribute the model without disclosing the details of the model. The request provides the name of the model. The response contains the serialized file as a SOAP attachment.
remove	adapa-admin	Remove one or more models. The request lists one or more model names to remove. The response does not contain any information.
removeAll	adapa-admin	Remove all models. The request does not provide any parameters and the response does not contain any information.
describe	adapa-admin, adapa-ws-user	Retrieve information about one or more models. The request lists one or more model names to describe. The response contains information about the specific models, which consists of name, description, input fields, output fields, model upload status (SUCCESS/WARNING), and model runtime (active) status (true/false).
describeAll	adapa-admin, adapa-ws-user	Retrieve information about all models. The request does not provide any parameters. The response contains information about all models, which consists of name, description, input fields, output fields, model upload status (SUCCESS/WARNING), and model runtime (active) status (true/false).
apply	adapa-admin, adapa-ws-user	Apply one or more models to one or more data records. The request lists the model names to be applied and provides the input data records. The web service will apply the specified models to every record and will respond with one output record for each result. The total number of records in the response will be the product of the number of input records and the

Operation Name	Authorized Role(s)	Description
		<p>number of models. The output records are returned in the same order as the input records. In case of multiple models, the response will first provide all the output records for the first input record, then all the output records for the second input record, and so forth. Finally, note that each output record identifies the model that produced it.</p>
explainApply	adapa-admin, adapa-ws-user	<p>Apply one or more models to one data record and return an execution trace for each model. The request lists the model names to be applied and provides the input data record. The web service will apply the specified models to the input record and will respond with an execution trace for each model. The execution trace describes in detail the input to the model, the mining schema for the model (with missing value/outlier value treatment, if any), the derived fields with transformations, the computation done in the model algorithm and the output generated by the model. The output also contains the model name for which the execution trace was generated.</p>
applyToBinary	adapa-admin, adapa-ws-user	<p>Apply one or more models to a binary file. The request lists the model names to be applied and provides the binary file to score as an attachment. The attachment can be in any format which could be specified in PMML. This is to avoid type mismatch problems if the format of the provided binary file does not match the format in PMML. The web service will apply the specified models to the attached binary file and will respond with one output CSV file for each model. The output CSV file contains the output records of the provided models. In case of multiple models, the response will provide output CSV files along with the corresponding model.</p>
applyToCSV	adapa-admin, adapa-ws-user	<p>Apply one or more models to all records in a CSV file. The request lists the model names to be applied and provides the CSV file to score as an attachment. The attachment can be a CSV file or a compressed CSV file in ZIP/GZIP format. Optionally, the request can provide aliases for column names in the CSV file. This is convenient if the column names do not match the field names of the model or if a CSV column needs to be mapped to more than one field. The web service will apply the specified models to the attached CSV file and will respond with one output CSV file for each model. If the CSV file was attached as a ZIP/GZIP file, the output</p>

Operation Name	Authorized Role(s)	Description
		<p>will contain the output CSV file in a corresponding ZIP/GZIP format. The output CSV file contains the output records in the same order as the input records contained in the input CSV file. In case of multiple models, the response will provide output CSV files along with the corresponding model that produced it and its content type, i.e. CSV or ZIP/GZIP.</p>
applyAsync	adapa-admin, adapa-ws-user	<p>Asynchronously apply one model to all records from remote data source, and write back results to remote data target. The request provides model name to be applied and a properties file as an attachment. This properties file describes the remote data source and target locations, connection properties, and access credentials. Optionally, the request can provide aliases for column names in the data file. This is convenient if the field names do not match the field names of the model or if a field name needs to be mapped to more than one field. The web service will asynchronously stream input file from specified source location, apply specified model to the source data, and stream results to remote target. If the data source is compressed ZIP/GZIP file, the output will contain the output ZIP/GZIP format. The output data stream contains the output records in the same order as the input records contained in the input stream. Immediately after processing is started, response is sent to the client with information about asynchronous scoring job such as current state, job id, output handle, start timestamp, and job description.</p> <p>Note</p> <p>This service currently only supports processing data on AWS S3 bucket.</p>
importResource	adapa-admin	<p>Import a new custom resource into ZEMENTIS. The request is a resource file provided as a SOAP attachment. The resource file can be a custom function JAR file or a look-up table represented in Microsoft Excel spreadsheet. On successful import, the response will contain resource file name, type, identifier, and a list of resource names contained in the resource file.</p>
exportResource	adapa-admin	<p>Export the resource file. The request provides resource the file name by which it is identified in ZEMENTIS. The response contains the resource file as a SOAP attachment.</p>

Operation Name	Authorized Role(s)	Description
describeAll-Resources	adapa-admin, adapa-ws-user	Retrieve information about all resource files. The request does not provide any parameters. The response contains information about all available resources, including resource file name, resource type, resource identifier, and list of resource names.
removeResource	adapa-admin	Remove one or more resource files. The request lists one or more resource file names to remove. The response does not contain any information.
removeAllResources	adapa-admin	Remove all the resource files. The request does not provide any parameters and the response does not contain any information.

5.2.1. SOAP Request and Response

This section provides examples of SOAP requests and responses for common Models Web Service operations. For brevity, the SOAP Envelope and Header are not included in the code snippet. A valid username and password has to be provided with each SOAP request as a Basic HTTP Authorization header. If the credentials do not map to the role(s) required for a specific operation (as described in [Table 5.3](#)), a SOAP Fault is returned with the Authorization Failure message.

Example 5.1. ZEMENTIS Models Web Service Import Model

Following is a SOAP request for the `import` operation. The request shows the PMML file being sent as an [MTOM](#) attachment. With MTOM, the binary data (contents of the PMML file) is included as a MIME attachment to the SOAP message. The SOAP request itself contains a placeholder for the file and the binary data is placed between delimiters at the end of the SOAP request. Without MTOM the binary data is encoded (BASE64) and is included inline in the SOAP request. ZEMENTIS Models Web Service supports both formats and leaves the choice to the client application. In our examples, we will list SOAP messages with MTOM enabled.

The `applyCleanser` flag indicates if comprehensive semantic checks and corrections are to be performed on the provided PMML file. By default, the value is `true`. Turning `applyCleanser` to `false` will improve upload time, but this is only recommended for annotated PMML files that are generated after being processed by ZEMENTIS.

Tip

If the PMML file is large, such as Random Forest model, we recommend compressing the file using ZIP/GZIP before uploading. This will reduce the upload time dramatically.

```
<soap:Body xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <models:importRequest xmlns:models="http://www.zementis.com/adapa/ws/models">
    <file><inc:Include href="cid:263290870530" xmlns:inc="http://www.w3.org/2004/08/xop/include"/></file>
    <applyCleanser>true</applyCleanser>
  </models:importRequest>
</soap:Body>
```

Following is the SOAP response which indicates that the model `Iris_NN` was successfully uploaded into ZEMENTIS.

```
<soap:Body xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <models:importResponse xmlns:models="http://www.zementis.com/adapa/ws/models">
    <importResult status="SUCCESS">
      <message>Model has been successfully imported.</message>
      <importedModels>Iris_NN</importedModels>
    </importResult>
  </models:importResponse>
</soap:Body>
```

Example 5.2. ZEMENTIS Models Web Service Describe Model

Following is the SOAP request for the `describe` operation, which expects one or more `modelName`s as input. In this example we describe the `Iris_NN` model.

```
<soap:Body xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <models:describeRequest xmlns:models="http://www.zementis.com/adapa/ws/models">
    <modelName>Iris_NN</modelName>
  </models:describeRequest>
</soap:Body>
```

The SOAP response provides the model description and status along with model inputs and outputs. It further indicates whether the model is currently loaded into memory with the field `active`.

```
<soap:Body xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <models:describeResponse xmlns:models="http://www.zementis.com/adapa/ws/models">
    <model>
      <name>Iris_NN</name>
      <description>Neural Network for multi-class classification using the Iris dataset</description>
      <input usage="ACTIVE" type="DOUBLE" name="sepal_length"/>
      <input usage="ACTIVE" type="DOUBLE" name="sepal_width"/>
      <input usage="ACTIVE" type="DOUBLE" name="petal_length"/>
      <input usage="ACTIVE" type="DOUBLE" name="petal_width"/>
      <output usage="OUTPUT" type="STRING" name="class"/>
      <output usage="OUTPUT" type="DOUBLE" name="Probability_setosa"/>
      <output usage="OUTPUT" type="DOUBLE" name="Probability_versicolor"/>
      <output usage="OUTPUT" type="DOUBLE" name="Probability_virginica"/>
    </model>
  </models:describeResponse>
</soap:Body>
```

```

        <status>SUCCESS</status>
        <active>true</active>
    </model>
</models:describeResponse>
</soap:Body>

```

Example 5.3. ZEMENTIS Models Web Service Apply Model

Following is a SOAP request for the `apply` operation. The request lists three models to be applied to two records. The models are: `Iris_NN`, `Iris_MLR`, and `Iris_CT`, all of which are available in our distribution samples.

```

<soap:Body xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <models:applyRequest xmlns:models="http://www.zementis.com/adapa/ws/models">
    <modelName>Iris_NN</modelName>
    <modelName>Iris_MLR</modelName>
    <modelName>Iris_CT</modelName>
    <record>
      <field name="sepal_length" value="5.1" />
      <field name="sepal_width" value="3.5" />
      <field name="petal_length" value="1.4" />
      <field name="petal_width" value="0.2" />
    </record>
    <record>
      <field name="sepal_length" value="4.9" />
      <field name="sepal_width" value="3.0" />
      <field name="petal_length" value="1.4" />
      <field name="petal_width" value="0.2" />
    </record>
  </models:applyRequest>
</soap:Body>

```

The corresponding SOAP response is shown below. Notice that the response contains six records. The first three records are the results of applying the three models to the first record. The last three records are the result of applying the three models to the second record. Each record contains fields with the predicted values as well as the name of the model which produced it.

```

<soap:Body xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <models:applyResponse xmlns:models="http://www.zementis.com/adapa/ws/models">
    <record>
      <field name="class" value="Iris-setosa" />
      <field name="Probability_setosa" value="0.9995535104664939" />
      <field name="Probability_versicolor" value="4.464895332525394E-4" />
      <field name="Probability_virginica" value="2.536692637033174E-13" />
      <modelName>Iris_NN</modelName>
    </record>
    <record>
      <field name="class" value="Iris-setosa" />
      <field name="Probability_setosa" value="0.9999999999999996" />
      <field name="Probability_versicolor" value="4.0732051602909886E-15" />
      <field name="Probability_virginica" value="6.290640809163842E-42" />
      <modelName>Iris_MLR</modelName>
    </record>
    <record>
      <field name="class" value="Iris-setosa" />

```

```

    <field name="Probability_setosa" value="1.0" />
    <field name="Probability_versicolor" value="0.0" />
    <field name="Probability_virginica" value="0.0" />
    <modelName>Iris_CT</modelName>
  </record>
</record>
<record>
  <field name="class" value="Iris-setosa" />
  <field name="Probability_setosa" value="0.9985890830740689" />
  <field name="Probability_versicolor" value="0.0014109169248845783" />
  <field name="Probability_virginica" value="1.0465677336558715E-12" />
  <modelName>Iris_NN</modelName>
</record>
<record>
  <field name="class" value="Iris-setosa" />
  <field name="Probability_setosa" value="0.9999999999992712" />
  <field name="Probability_versicolor" value="7.287039008004534E-13" />
  <field name="Probability_virginica" value="5.20202200281695E-38" />
  <modelName>Iris_MLR</modelName>
</record>
<record>
  <field name="class" value="Iris-setosa" />
  <field name="Probability_setosa" value="1.0" />
  <field name="Probability_versicolor" value="0.0" />
  <field name="Probability_virginica" value="0.0" />
  <modelName>Iris_CT</modelName>
</record>
</models:applyResponse>
</soap:Body>

```

Example 5.4. ZEMENTIS Models Web Service Apply Model with Settings

Following is a SOAP request for the `apply` operation with settings enabled. The `settings` element is optional when applying one or more models to data as shown in the previous example [Example 5.3](#). However, the `settings` element can be used to customize data processing. The `returnedFieldName` element is used to specify the field names that need to be returned as an output. One `returnedFieldName` element is required for each field that needs to be returned. The `returnedFieldUsage` element is used to specify the usage type of fields that needs to be returned as an output. Again, one `returnedFieldUsage` element is required for each usage type that needs to be returned. The `maxThreads` and `maxRecordsPerThread` options are only applicable for batch scoring using a CSV file. The default values for these two settings are sufficient for most use cases.

The following request lists the same three models to be applied to two records. But now we add a `returned-FieldName` and `returnedFieldUsage` options in the `settings` element.

```

<soap:Body>
  <models:applyRequest xmlns:models="http://www.zementis.com/adapa/ws/models">
    <modelName>Iris_NN</modelName>
    <modelName>Iris_MLR</modelName>
    <modelName>Iris_CT</modelName>
    <record>
      <field name="sepal_length" value="5.1" />
      <field name="sepal_width" value="3.5" />
      <field name="petal_length" value="1.4" />
      <field name="petal_width" value="0.2" />
    </record>
    <record>

```



```

    <field name="sepal_length" value="4.9" />
    <field name="sepal_width" value="3.0" />
    <field name="petal_length" value="1.4" />
    <field name="petal_width" value="0.2" />
  </record>
  <settings>
    <returnedFieldName>class</returnedFieldName>
    <returnedFieldUsage>ACTIVE</returnedFieldUsage>
  </settings>
</models:applyRequest>
</soap:Body>

```

The corresponding SOAP response is shown below. The response contains six records. The first three records are the results of applying the three models to the first record. The last three records are the result of applying the three models to the second record. Each record contains fields which are of type `ACTIVE` along with field named `class` as was indicated in the request.

```

<soap:Body xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <models:applyResponse xmlns:models="http://www.zementis.com/adapa/ws/models">
    <record>
      <field value="5.1" name="sepal_length"/>
      <field value="0.2" name="petal_width"/>
      <field value="Iris-setosa" name="class"/>
      <field value="3.5" name="sepal_width"/>
      <field value="1.4" name="petal_length"/>
      <modelName>Iris_MLR</modelName>
    </record>
    <record>
      <field value="1.4" name="petal_length"/>
      <field value="Iris-setosa" name="class"/>
      <field value="5.1" name="sepal_length"/>
      <field value="3.5" name="sepal_width"/>
      <field value="0.2" name="petal_width"/>
      <modelName>Iris_CT</modelName>
    </record>
    <record>
      <field value="3.5" name="sepal_width"/>
      <field value="5.1" name="sepal_length"/>
      <field value="1.4" name="petal_length"/>
      <field value="Iris-setosa" name="class"/>
      <field value="0.2" name="petal_width"/>
      <modelName>Iris_NN</modelName>
    </record>
    <record>
      <field value="3.0" name="sepal_width"/>
      <field value="1.4" name="petal_length"/>
      <field value="4.9" name="sepal_length"/>
      <field value="0.2" name="petal_width"/>
      <field value="Iris-setosa" name="class"/>
      <modelName>Iris_MLR</modelName>
    </record>
    <record>
      <field value="Iris-setosa" name="class"/>
      <field value="0.2" name="petal_width"/>
      <field value="1.4" name="petal_length"/>
      <field value="4.9" name="sepal_length"/>
      <field value="3.0" name="sepal_width"/>
      <modelName>Iris_NN</modelName>
    </record>
    <record>
      <field value="Iris-setosa" name="class"/>

```

```

    <field value="1.4" name="petal_length"/>
    <field value="4.9" name="sepal_length"/>
    <field value="0.2" name="petal_width"/>
    <field value="3.0" name="sepal_width"/>
    <modelName>Iris_CT</modelName>
  </record>
</models:applyResponse>
</soap:Body>

```

Example 5.5. ZEMENTIS Models Web Service Apply Model to Binary Source

Following is a SOAP request for the `applyToBinary` operation. The request lists the model to be applied to a binary file which is sent as an MTOM attachment. Note that more than one `modelName` elements can be specified in the request. In this case, the attached binary file will be processed against each of the specified models.

```

<soap:Body xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <models:applyToBinaryRequest xmlns:models="http://www.zementis.com/adapa/ws/models">
    <modelName>Caffe_NN</modelName>
    <file><inc:Include href="cid:1368237645855" xmlns:inc="http://www.w3.org/2004/08/xop/include"/></file>
  </models:applyToBinaryRequest>
</soap:Body>

```

The corresponding SOAP response contains one `csvOutput` element for each model that was applied. The following response shows one `csvOutput` element for the model `Caffe_NN`. The output CSV file is sent back as an MTOM attachment. The `contentType` element in the `csvOutput` will indicate the file type returned as an output.

```

<soap:Body xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <models:applyToBinaryResponse xmlns:models="http://www.zementis.com/adapa/ws/models">
    <csvOutput>
      <modelName>Caffe_NN</modelName>
      <contentType>application/csv; charset=UTF-8</contentType>
      <file>
        <xop:Include href="cid:c89749b2-861d-41db-9920-0f3d2434ccb8-1@cxf.apache.org"
          xmlns:xop="http://www.w3.org/2004/08/xop/include"/>
      </file>
    </csvOutput>
  </models:applyToBinaryResponse>
</soap:Body>

```

Example 5.6. ZEMENTIS Models Web Service Apply Model to CSV

Following is a SOAP request for the `applyToCSV` operation. The request lists the model to be applied to a CSV file which is sent as an MTOM attachment. Note that more than one `modelName` elements can be specified in the request. In this case, the attached CSV file will be processed against each of the specified models.

```
<soap:Body xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <models:applyToCSVRequest xmlns:models="http://www.zementis.com/adapa/ws/models">
    <modelName>Iris_NN</modelName>
    <file><inc:Include href="cid:1368237645855" xmlns:inc="http://www.w3.org/2004/08/xop/include"/></file>
  </models:applyToCSVRequest>
</soap:Body>
```

The corresponding SOAP response contains one `csvOutput` element for each model that was applied. The following response shows one `csvOutput` element for the model `Iris_NN`. The output CSV file is sent back as an MTOM attachment. If the input CSV file was sent as a compressed file in either ZIP or GZip format, the output CSV file will also be in the same format. The `contentType` element in the `csvOutput` will indicate the file type returned as an output.

```
<soap:Body xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <models:applyToCSVResponse xmlns:models="http://www.zementis.com/adapa/ws/models">
    <csvOutput>
      <modelName>Iris_NN</modelName>
      <contentType>application/csv; charset=UTF-8</contentType>
      <file>
        <xop:Include href="cid:70139587-7b1a-4800-93d2-77dd17b70aa0-2@cxf.apache.org"
          xmlns:xop="http://www.w3.org/2004/08/xop/include"/>
      </file>
    </csvOutput>
  </models:applyToCSVResponse>
</soap:Body>
```

Example 5.7. ZEMENTIS Models Web Service Import Resource

The `importResource` operation can be used to upload custom resources like Custom Functions and Lookup Tables in ZEMENTIS. More information about custom resources can be found in [Chapter 3](#). Following is a SOAP request for the `importResource` operation. A `custom-functions.jar` file is sent as an MTOM attachment. If the upload is successful, the value in the `fileName` element will be used to identify the resource file on ZEMENTIS.

```
<soap:Body xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <models:importResourceRequest xmlns:models="http://www.zementis.com/adapa/ws/models">
    <fileName>custom-functions.jar</fileName>
    <file><inc:Include href="cid:709658965540" xmlns:inc="http://www.w3.org/2004/08/xop/include"/></file>
  </models:importResourceRequest>
</soap:Body>
```

Following is the SOAP response which indicates that the resource file `custom-functions.jar` was successfully uploaded into ZEMENTIS. The response also enumerates the resources found in the resource file using the `resourceName` elements.

```
<soap:Body xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <models:importResourceResponse xmlns:models="http://www.zementis.com/adapa/ws/models">
    <importResourceResult resourceIdentifier="Function Namespace" resourceType="CUSTOM_FUNCTIONS"
      fileName="custom-functions.jar">
      <resourceName>zementis</resourceName>
      <resourceName>zcustom1</resourceName>
      <resourceName>zem1</resourceName>
    </importResourceResult>
  </models:importResourceResponse>
</soap:Body>
```

5.3. RPC Web Service

With the ZEMENTIS RPC Web Service, each predictive model is exposed as a separate operation. Operations are added or removed from the service as models get added or removed from ZEMENTIS. The name of each model becomes the name of the corresponding operation. The input fields of the model become the operation's input parameters and the output fields of the model provide the output values. Executing an operation on this web service will result in scoring or classifying a single data record against the predictive model with the same name.

The WSDL of the service is dynamically generated based on the models that have been uploaded into ZEMENTIS. As an example, [Example 5.8](#) presents the generated WSDL after the model `Iris_NN` is uploaded, with no other model being available.

Compared to the generic Models Web Service, the RPC Web Service offers no model management capabilities and allows applying only one model to one record at a time. However, this is particularly useful for scenarios where individual predictions are integrated into a business process, using tools that rely on WSDL introspection to facilitate graphical mapping of data flows.

Example 5.8. ZEMENTIS RPC Web Service WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="AdapaRPCService" targetNamespace="http://www.zementis.com/adapa/ws/rpc"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:tns="http://www.zementis.com/adapa/ws/rpc"
  xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
  <wsdl:message name="Iris_NNRequest">
    <wsdl:part name="sepal_length" type="xs:double"/>
    <wsdl:part name="sepal_width" type="xs:double"/>
    <wsdl:part name="petal_length" type="xs:double"/>
    <wsdl:part name="petal_width" type="xs:double"/>
  </wsdl:message>
  <wsdl:message name="Iris_NNResponse">
    <wsdl:part name="class" type="xs:string"/>
    <wsdl:part name="Probability_setosa" type="xs:double"/>
    <wsdl:part name="Probability_versicolor" type="xs:double"/>
    <wsdl:part name="Probability_virginica" type="xs:double"/>
  </wsdl:message>
  <wsdl:portType name="RPCService">
    <wsdl:operation name="Iris_NN">
      <wsdl:input message="tns:Iris_NNRequest"/>
      <wsdl:output message="tns:Iris_NNResponse"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="AdapaRPCServiceSoapBinding" type="tns:RPCService">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="Iris_NN">
      <soap:operation soapAction="" style="rpc"/>
      <wsdl:input>
        <soap:body use="literal" namespace="http://www.zementis.com/adapa/ws/rpc"/>
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal" namespace="http://www.zementis.com/adapa/ws/rpc"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="AdapaRPCService">
    <wsdl:port name="AdapaRPCServicePort" binding="tns:AdapaRPCServiceSoapBinding">
      <soap:address location="http://localhost:8080/adapaws/rpc"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Following the above WSDL, the body of a SOAP request to apply the `Iris_NN` model would look like the one in [Example 5.9](#).

Example 5.9. ZEMENTIS RPC Web Service SOAP Request Body

```
<soap:Body xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <rpc:Iris_NN xmlns:rpc="http://www.zementis.com/adapa/ws/rpc">
    <sepal_length>5.1</sepal_length>
    <sepal_width>3.5</sepal_width>
    <petal_length>1.4</petal_length>
    <petal_width>0.2</petal_width>
  </rpc:Iris_NN>
</soap:Body>
```

The corresponding body of the SOAP response is shown in [Example 5.10](#). Notice that the response contains the value for the predicted value (`class`) and the probabilities for three possible classes.

Example 5.10. ZEMENTIS RPC Web Service SOAP Response Body

```
<soap:Body>
  <rpc:Iris_NNResponse xmlns:rpc="http://www.zementis.com/adapa/ns/rpc">
    <class>Iris-setosa</class>
    <Probability_setosa>0.9995535104664939</Probability_setosa>
    <Probability_versicolor>4.464895332525399E-4</Probability_versicolor>
    <Probability_virginica>2.536692637033174E-13</Probability_virginica>
  </rpc:Iris_NNResponse>
</soap:Body>
```

5.4. Using ZEMENTIS Web Services from Java

This section presents sample Java applications invoking the ZEMENTIS Web Services. The source code and other files required to run these examples are included in the provided sample package. To execute the Java applications, you need to have Java and [Apache Ant](#) available in the system path. Java Development Kit version 7 can be downloaded from the [Java Download Page](#) and Apache Ant version 1.7.1 can be downloaded from the [Apache Ant Download Page](#).

Our sample relies on the [Apache CXF](#) framework to create proxy clients for the web services and the [Spring](#) framework for configuration. With Apache CXF, the client application can be programmed against a client proxy library, ignoring all the underlying details of actually invoking the web service. Finally, a Spring configuration file is used to connect the proxy to the actual web service.

In our sample application, we include an Apache Ant build script (`build.xml`) that contains tasks for the steps necessary to compile and execute applications against the ZEMENTIS Web Services. The Apache Ant script requires values for the base URL of your ZEMENTIS instance and the user credentials. These must be set in the accompanying `adapa.properties` file. For the credentials, use the user name and password you use to log into the ZEMENTIS Console. For the base URL, use the `ZEMENTIS-BASE-URL` as described in [Section 5.1](#). Finally, the file `adapaWSContext.xml` contains the necessary Spring configuration information. As in the case of the Apache Ant script, the Spring configuration file also relies on the parameters specified in the `adapa.properties` file.

Important

If SSL is enabled, the Java runtime environment needs to be configured to trust the server certificate. This can be achieved by importing the server certificate into the `cacerts` trusted certificate key store using the `keytool` utility included in the Java installation package. Detailed instructions are provided in the article [Secure Internet Programming with the Java 2 Platform](#) and the documentation about `keytool`. As an example, for ZEMENTIS deployments on [Amazon EC2](#), you have to download the [Zementis CA Root Certificate](#) and import it using the command `keytool -import -file ZementisCA.cer -keystore JAVA_HOME/jre/lib/security/cacerts` where `JAVA_HOME` is the directory where Java is installed. Note that the default password for the key store is `changeit`.

5.4.1. Models Web Service

This section presents a sample Java client application that invokes the Models Web Service to apply models `Iris_NN`, `Iris_MLR`, and `Iris_CT` to two records.

The listing in [Example 5.11](#) contains the sample source code and shows the key aspects of preparing, invoking, and handling the results of the models web service call. First, the input records are prepared. They are created as a `List` of `XmlRecords`, one for each record that needs to be scored. Then, the set of model names to be applied is also prepared. The actual web service call comes next. This happens through the `apply` method of the web service proxy. This method takes as input the `Set` of model names to apply and the `List` of `XmlRecords` to score. Assuming there are no errors executing the operation, the result is returned as a `List` of `XmlOutputRecords`. Finally, we iterate over the returned list and print the computed values. Similarly, we execute the web service for a model that references custom functions and lookup table. First, we upload resource files, followed by model source upload, and then we apply data from a CSV file.

Example 5.11. ZEMENTIS Models Web Service Java client

```
/*
 * 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.sample.models;

import java.io.File;
import java.io.FileOutputStream;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Properties;

import javax.activation.DataHandler;
import javax.activation.DataSource;
import javax.activation.FileDataSource;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.Marshaller;
import javax.xml.ws.BindingProvider;

import com.zementis.adapa.ws.models.AdapaModelsService;
import com.zementis.adapa.ws.models.ApplyRequest;
import com.zementis.adapa.ws.models.ApplyResponse;
import com.zementis.adapa.ws.models.ApplySettingsType;
import com.zementis.adapa.ws.models.ApplyToCSVRequest;
import com.zementis.adapa.ws.models.ApplyToCSVResponse;
import com.zementis.adapa.ws.models.CSVOutputType;
import com.zementis.adapa.ws.models.FieldType;
import com.zementis.adapa.ws.models.ImportRequest;
import com.zementis.adapa.ws.models.ImportResourceRequest;
import com.zementis.adapa.ws.models.ImportResourceResponse;
import com.zementis.adapa.ws.models.ImportResponse;
```

```
import com.zementis.adapa.ws.models.ImportResultInformationType;
import com.zementis.adapa.ws.models.ModelsService;
import com.zementis.adapa.ws.models.ObjectFactory;
import com.zementis.adapa.ws.models.OutputRecordType;
import com.zementis.adapa.ws.models.RecordType;
import com.zementis.adapa.ws.models.ResourceInformationType;
import com.zementis.adapa.ws.models.Usage;

public final class SampleWSInvocation {

    private static final ModelsService service = new AdapaModelsService().getAdapaModelsServicePort();

    private static final String PMML = "pmml";

    private static final String DATA = "data";

    private static final String RESOURCES = "resources";

    public static void main(String[] args) throws Exception {

        setAuthenticationParameters();

        // Upload models
        uploadModel(new File(PMML, "Iris_NN.pmml"));
        uploadModel(new File(PMML, "Iris_CT.pmml"));
        uploadModel(new File(PMML, "Iris_MLR.pmml"));

        // Apply the models to XML Records
        List<String> modelNames = new ArrayList<String>();
        modelNames.add("Iris_NN");
        modelNames.add("Iris_CT");
        modelNames.add("Iris_MLR");

        List<RecordType> inputRecords = new ArrayList<RecordType>();
        inputRecords.add(createInputXmlRecord(5.1, 3.5, 1.4, 0.2));
        inputRecords.add(createInputXmlRecord(4.9, 3, 1.4, 0.2));

        ApplyRequest applyRequest = new ApplyRequest();
        applyRequest.getModelName().addAll(modelNames);
        applyRequest.getRecord().addAll(inputRecords);

        JAXBContext jaxbContext = JAXBContext.newInstance("com.zementis.adapa.ws.models");
        Marshaller marshaller = jaxbContext.createMarshaller();
        System.out.println("Apply Request : \n");
        marshaller.marshal(
            new ObjectFactory().createApplyRequest(applyRequest),
            System.out);

        System.out.println("\n\nExecuting Models WS...\n");
        // Execute the web service
        ApplyResponse applyResponse = service.apply(applyRequest);

        // Print out the output values
        for (OutputRecordType outputRecord : applyResponse.getRecord()) {
            System.out
                .print("Model [" + outputRecord.getModelName() + "] : \n");
            for (FieldType field : outputRecord.getField()) {
                System.out.println(field.getName() + " = " + field.getValue());
            }
        }

        // Execute the web service with apply settings
        ApplySettingsType applySettingsType = new ApplySettingsType();
        // Lets return fields with Usage type ACTIVE, TARGET and OUTPUT
        applySettingsType.getReturnedFieldUsage().add(Usage.ACTIVE);
        applySettingsType.getReturnedFieldUsage().add(Usage.TARGET);
        applySettingsType.getReturnedFieldUsage().add(Usage.OUTPUT);
        applyRequest.setSettings(applySettingsType);
    }
}
```



```

System.out.println("\nExecuting Models WS with ApplySettings...\n");
// Execute the web service
applyResponse = service.apply(applyRequest);

// Print out the output values
for (OutputRecordType outputRecord : applyResponse.getRecord()) {
    System.out
        .print("Model [" + outputRecord.getModelName() + "] : \n");
    for (FieldType field : outputRecord.getField()) {
        System.out.println(field.getName() + " = " + field.getValue());
    }
}

// Apply models to CSV file
File csvFile = new File(DATA, "Iris_NN.csv");
applyModelToCSV(modelNames, csvFile);

System.out.println("\nExecuting Models WS for model with resources...\n");

uploadResource(new File(RESOURCES, "ECommerceFraud_NN.jar"));
uploadResource(new File(RESOURCES, "ECommerceFraud_NN.xls"));

modelNames.clear();
modelNames.add("ECommerceFraud_NN");
uploadModel(new File(PMML, "ECommerceFraud_NN.pmml"));

applyModelToCSV(modelNames, new File(DATA, "ECommerceFraud_NN.csv"));
}

/**
 * @param pmmlFile
 *         pmml File
 */
private static void uploadModel(File pmmlFile) {
    DataSource dataSource = new FileDataSource(pmmlFile);
    DataHandler dataHandler = new DataHandler(dataSource);

    ImportRequest importRequest = new ImportRequest();
    importRequest.setFile(dataHandler);

    ImportResponse importResponse = service._import(importRequest);
    ImportResultInformationType importResult = importResponse.getImportResult();
    System.out.println(importResult.getMessage());
}

/**
 * @param resourceFile
 *         resource File
 */
private static void uploadResource(File resourceFile) {
    DataSource dataSource = new FileDataSource(resourceFile);
    DataHandler dataHandler = new DataHandler(dataSource);

    ImportResourceRequest importResourceRequest = new ImportResourceRequest();
    importResourceRequest.setFileName(resourceFile.getName());
    importResourceRequest.setFile(dataHandler);
    try {
        ImportResourceResponse importResultResponse = service.importResource(importResourceRequest);
        ResourceInformationType importResult = importResultResponse.getImportResourceResult();
        System.out.println("Resource " + importResult.getFileName() + " has been successfully
imported.");
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}

/**
 * @param sepal_length

```

```

*           sepal length
* @param sepal_width
*           sepal width
* @param petal_length
*           petal length
* @param petal_width
*           petal width
* @return {@link RecordType}
*/
private static RecordType createInputXmlRecord(double sepal_length,
        double sepal_width, double petal_length, double petal_width) {
    RecordType record = new RecordType();

    List<FieldType> fields = new ArrayList<FieldType>();
    fields.add(createInputField("sepal_length", sepal_length));
    fields.add(createInputField("sepal_width", sepal_width));
    fields.add(createInputField("petal_length", petal_length));
    fields.add(createInputField("petal_width", petal_width));
    record.getField().addAll(fields);

    return record;
}

/**
 * @param fieldName
 *           name of the field
 * @param fieldValue
 *           value of the field
 * @return {@link FieldType}
 */
private static FieldType createInputField(String fieldName, double fieldValue) {
    FieldType field = new FieldType();
    field.setName(fieldName);
    field.setValue(String.valueOf(fieldValue));
    return field;
}

/**
 * @param modelName
 *           model name to apply
 * @param csvFile
 *           csv file
 * @throws Exception
 *           if something goes wrong
 */
private static void applyModelToCSV(List<String> modelNames, File csvFile)
    throws Exception {
    System.out.println("Applying models " + modelNames + " to CSV file " + csvFile.getAbsolutePath());
    // Create the request object for 'applyToCSV' ws operation
    ApplyToCSVRequest applyToCSVRequest = new ApplyToCSVRequest();

    // Set the name of the models to score
    applyToCSVRequest.getModelName().addAll(modelNames);

    // Set the CSV file as the input file
    DataHandler inputDataHandler = new DataHandler(new FileDataSource(
        csvFile));
    applyToCSVRequest.setFile(inputDataHandler);

    // Execute the web service operation
    ApplyToCSVResponse applyToCSVResponse = service
        .applyToCSV(applyToCSVRequest);

    // Get the CSV output from the response (we scored using 3 models, we
    // get 3 CSVs back).
    for (CSVOutputType csvOutput : applyToCSVResponse.getCsvOutput()) {
        // Read the CSV file and store the result
        System.out.println("Generating output CSV " + csvOutput.getModelName() + "_Out.csv");
        File outputFile = new File(csvOutput.getModelName() + "_Out.csv");
    }
}

```

```
        csvOutput.getFile().writeTo(new FileOutputStream(outputFile));
    }
}

/**
 *
 */
private static void setAuthenticationParameters() {
    final Properties props = new Properties();
    try {
        props.load(new FileReader("adapa.properties"));
    } catch (IOException ioe) {
        System.err.println("Could not read the adapa.properties file");
        System.exit(0);
    }
    BindingProvider bindingProvider = (BindingProvider) service;
    bindingProvider.getRequestContext().put(
        BindingProvider.USERNAME_PROPERTY,
        props.getProperty("adapa.username"));
    bindingProvider.getRequestContext().put(
        BindingProvider.PASSWORD_PROPERTY,
        props.getProperty("adapa.password"));
}
}
```

The above sample application can be executed with the help of the provided [Apache Ant](#) script and the command **ant execute_models_ws**.

5.4.2. RPC Web Service

This section presents a sample Java client application that invokes the RPC Web Service to classify data using the `Iris_NN` model.

As mentioned before, the RPC Web Service is dynamic. The published WSDL is modified every time models are added to or removed from ZEMENTIS. This requires that the client proxy library gets generated from the WSDL after the appropriate model has been uploaded. In order to test the sample code, please upload the `Iris_NN` model first. Then, to generate the proxy library, use the provided Apache Ant script and the command **ant -f build.xml generate_proxy**.

Once a client proxy library has been created, programming against the web service becomes fairly simple. An example is shown in [Example 5.12](#). The client proxy is fetched from the Spring configuration. The client proxy contains the method `irisNN` with parameters matching the input mining fields of the model. Note that this particular model computes and returns more than one value (top class and class probabilities). In order to accommodate multiple return values in Java, special holder variables are declared and provided as parameters to the web service, along with the actual input data. Assuming there are no errors, the execution of the web service populates the holder variables with the computed values.

Note

Holder variables are not needed for models that compute a single value. In such a case, the actual service call has a return value which can be assigned directly to a variable.

Example 5.12. ZEMENTIS RPC Web Service Java client

```
/*
 * 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.sample.rpc;

import java.io.FileReader;
import java.io.IOException;
import java.util.Properties;

import javax.xml.ws.BindingProvider;
import javax.xml.ws.Holder;

import com.zementis.adapa.ws.rpc.AdapaRPCService;
import com.zementis.adapa.ws.rpc.RPCService;

public final class SampleWSInvocation {

    private static final RPCService service = new AdapaRPCService().getAdapaRPCServicePort();

    public static void main(String[] args) {

        setAuthenticationParameters();

        // Lets create holders for outputs
        Holder<String> classHolder = new Holder<String>();
        Holder<Double> probabilitySetosaHolder = new Holder<Double>();
        Holder<Double> probabilityVersicolorHolder = new Holder<Double>();
        Holder<Double> probabilityVirginicaHolder = new Holder<Double>();

        // Execute the web service (Iris MLR model)
        service.irisMLR(5.1, 3.5, 1.4, 0.2, classHolder, probabilitySetosaHolder,
            probabilityVersicolorHolder, probabilityVirginicaHolder);

        // Print out output values
        System.out.println("-----Iris MLR-----");
        System.out.println("class = " + classHolder.value);
        System.out.println("Probability_setosa = " + probabilitySetosaHolder.value);
        System.out.println("Probability_versicolor = " + probabilityVersicolorHolder.value);
        System.out.println("Probability_virginica = " + probabilityVirginicaHolder.value);

        // Execute the web service (Iris NN model)
        service.irisNN(5.1, 3.5, 1.4, 0.2, classHolder, probabilitySetosaHolder,
            probabilityVersicolorHolder,
            probabilityVirginicaHolder);

        // Print out output values
        System.out.println("-----Iris NN-----");
        System.out.println("class = " + classHolder.value);
        System.out.println("Probability_setosa = " + probabilitySetosaHolder.value);
        System.out.println("Probability_versicolor = " + probabilityVersicolorHolder.value);
        System.out.println("Probability_virginica = " + probabilityVirginicaHolder.value);
    }
}
```

```
// Execute the web service (Iris CT model)
service.irisCT(5.1, 3.5, 1.4, 0.2, classHolder, probabilitySetosaHolder,
probabilityVersicolorHolder,
    probabilityVirginicaHolder);
// Print out output values
System.out.println("-----Iris CT-----");
System.out.println("class = " + classHolder.value);
System.out.println("Probability_setosa = " + probabilitySetosaHolder.value);
System.out.println("Probability_versicolor = " + probabilityVersicolorHolder.value);
System.out.println("Probability_virginica = " + probabilityVirginicaHolder.value);
}

/**
 *
 */
private static void setAuthenticationParameters() {
    final Properties props = new Properties();
    try {
        props.load(new FileReader("adapa.properties"));
    } catch (IOException ioe) {
        System.err.println("Could not read the adapa.properties file");
        System.exit(0);
    }
    BindingProvider bindingProvider = (BindingProvider) service;
    bindingProvider.getRequestContext().put(BindingProvider.USERNAME_PROPERTY,
props.getProperty("adapa.username"));
    bindingProvider.getRequestContext().put(BindingProvider.PASSWORD_PROPERTY, props.getProperty(
        "adapa.password"));
}
}
```

The above sample application can be executed with the help of the provided Apache Ant script and the command **ant execute_rpc_ws**.

Chapter 6. REST API

This Application Programming Interface (API) provides users with a comprehensive set of defined interfaces to interact with ZEMENTIS using Representational State Transfer (REST) over Hypertext Transfer Protocol (HTTP). ZEMENTIS REST API allows users to perform operations on models and custom resources, and process data by issuing a simple request using any HTTP client such as a web browser.

6.1. General Notes

6.1.1. URI

A full path to the ZEMENTIS REST API resource consists of a base path and a resource path. The base path Uniform Resource Identifier (URI) for the ZEMENTIS REST API is `http://domain:port/adapars`, where `http` or `https` is the protocol name, `domain` is the internet domain or network address, `port` is a non-negative integer representing the port number, and `adapars` represents the application context path. The base path is static and does not change between requests; it merely identifies the server with an application on the network. Connecting with your favorite web browser to the base path URI will load ZEMENTIS REST interactive API documentation that describes all available resources, enables request execution and displays received responses from the ZEMENTIS REST service. See [Figure 6.1](#).

Figure 6.1. Interactive REST API Documentation

models : Model operations			Show/Hide	List Operations	Expand Operations	Raw
GET	/models	List available models				
GET	/model/{model_name}	Get model properties				
GET	/model/{model_name}/source	Get PMML source				
GET	/model/{model_name}/serialized	Download serialized model				
POST	/model	Upload new PMML model				
PUT	/model	Upload new PMML model				
PUT	/model/{model_name}/activate	Activate a model				
PUT	/model/{model_name}/deactivate	Deactivate a model				
DELETE	/model/{model_name}	Remove model				
DELETE	/models	Remove all models				

apply : Apply model to data			Show/Hide	List Operations	Expand Operations	Raw
GET	/apply/{model_name}	Apply model to single input record				
GET	/apply/{model_name}/explain	Apply model to single input record and explain result				
POST	/apply/{model_name}	Apply model to multiple input records or to single binary record				
PUT	/apply/{model_name}	Apply model to multiple input records or to single binary record				
POST	/apply/{model_name}/async	Asynchronously apply model to multiple input records				
PUT	/apply/{model_name}/async	Asynchronously apply model to multiple input records				

resources : Resource operations			Show/Hide	List Operations	Expand Operations	Raw
GET	/resources	List all available resources				
GET	/resource/{file_name}	Get Resource Info				
GET	/resource/{file_name}/source	Download resource file				
POST	/resource	Upload new resource file				
PUT	/resource	Upload new resource file				
DELETE	/resource/{file_name}	Remove resource file				
DELETE	/resources	Remove all resource files				

Following the base path is the resource path. It may contain path or query parameters depending on the type of the request and available resources on the server. For example, a resource path `/model/Iris_NN/source?annotated=true` contains static path definitions such as `model` or `source`, path parameter `Iris_NN` for a dynamically allocated resource, and a query parameter `annotated=true`.

6.1.2. Request

The HTTP request is a combination of a simple Uniform Resource Identifier (URI), HTTP verb GET, POST, PUT, or DELETE, request parameters, which can be in the form of a path variable, query, body, or header parameters, and message body (content). The path variable is a variable part of otherwise static URI that denotes a set of possible resource names on the server and is denoted with curly braces. For example, our `/model/{model_name}/source` resource path specifies the PMML file for an arbitrary model denoted as `{model_name}`. Thus, the request path for the PMML file of model `Iris_NN` should be constructed as `/model/Iris_NN/source`. Query parameters are appended to the URI with a question mark followed by a list of key/value pairs. A query variable annotated with the value `true` in the `/model/Iris_NN?annotated=true` resource path specifies that the returned PMML file should contain annotations as placed by ZEMENTIS, in case of errors or warnings. Header parameters are HTTP message metadata in the form of key/value pairs containing information about the message such as content type, message encoding type, authorization, etc. Body parameters appear only in POST or PUT requests and need to be encoded by the HTTP client.

Please, refer to [HTTP 1.1 specification](#) for details.

6.1.3. Response

The HTTP response message is composed of a message header and a message body. All ZEMENTIS REST response content types implement standard UTF-8 character set encoding. The header contains response status code and header fields represented as list of key/value pairs, i.e. `Content-Type:application/json`.

Every response from ZEMENTIS REST contains a `Content-Type` header entry with one of following internet media types (aka MIME) as value.

- `application/json`
- `application/xml`
- `text/plain`
- `application/zip`

6.1.4. Errors

ZEMENTIS REST maps error responses to appropriate HTTP status codes and returns a Javascript Object Notation (JSON) [Errors](#) object in the response body containing an array of error messages. For example, if the requested model, e.g. `Iris_NN`, has not been uploaded into ZEMENTIS yet, a response header with status code 404 and its following response body with [Errors](#) are returned.

Example 6.1. ZEMENTIS REST Error Response

Request

```
curl -u adapa:adapa -k https://localhost/adapars/model/Iris_NN
```

Request Header

```
GET /adapars/model/Iris_NN HTTP/1.1
Authorization: Basic YWRhcGE6YWRhcGE=
User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r zlib/1.2.5
Host: localhost
Accept: */*
```

Response Header

```
HTTP/1.1 404 Not Found
Server: Apache-Coyote/1.1
Pragma: No-cache
Cache-Control: no-cache
Expires: Wed, 31 Dec 1969 16:00:00 PST
X-Powered-By: Servlet 2.5; JBoss-5.0/JBossWeb-2.1
Content-Type: application/json
Content-Length: 49
Date: Thu, 27 Mar 2014 20:22:14 GMT
```

Response Body

```
{
  "errors" : [ "Model 'Iris_NN' not found." ]
}
```

Table 6.1. Typical ZEMENTIS REST Error Responses

Code	Error Message
400	Empty input stream.
400	File name missing.
400	Invalid XML format.
400	Failed to parse JSON input.
400	Invalid CSV File.
401	This request requires HTTP authentication.
403	You are not authorized to access this resource.
404	Model 'model_name' not found.
404	Resource 'file_name' not found.
409	A model with the name 'model_name' already exists.
409	Resource file 'file_name' already exists.
500	Invalid License.

Code	Error Message
500	Internal server error.

6.1.5. Authorization

All requests are authorized by the basic access authentication method. For example, HTTP header entry `Authorization: Basic YWRhcGE6YWRhcGE=` is created for credentials with user name and password `adapa`. If the provided credentials fail to authenticate, the HTTP 401 response code is returned, and 403 if the user is not authorized to perform the requested operation. The below table lists the authorized role(s) for each operation, and the detailed description of each operation can be found in the following sections.

Table 6.2. ZEMENTIS REST Permissions

Operation	Definition	Authorized Role(s)
List Available Models	GET /models	adapa-admin, adapa-ws-user
Get Model Information	GET /model/{model_name}	adapa-admin, adapa-ws-user
Get Model Source	GET /model/{model_name}/source	adapa-admin
Get Model Serialized Source	GET /model/{model_name}/serialized	adapa-admin
Upload New Model with POST	POST /model	adapa-admin
Upload New Model with PUT	PUT /model	adapa-admin
Activate an existing Model with PUT	PUT /model/{model_name}/activate	adapa-admin
Deactivate an existing Model with PUT	PUT /model/{model_name}/deactivate	adapa-admin
Remove Model	DELETE /model/{model_name}	adapa-admin
Remove All Models	DELETE /models	adapa-admin
Apply Model to Single Record	GET /apply/{model_name}	adapa-admin, adapa-ws-user
Apply Model to Single Record and Explain Result	GET /apply/{model_name}/explain	adapa-admin, adapa-ws-user
Apply Model to Multiple Records or Apply Model to Single Binary Data with POST	POST /apply/{model_name}	adapa-admin, adapa-ws-user
Apply Model to Multiple Records or	PUT /apply/{model_name}	adapa-admin, adapa-ws-user

Operation	Definition	Authorized Role(s)
Apply Model to Single Binary Data with PUT		
Asynchronously Apply Model to Multiple Records with POST	POST /apply/{model_name}/async	adapa-admin, adapa-ws-user
Asynchronously Apply Model to Multiple Records with PUT	PUT /apply/{model_name}/async	adapa-admin, adapa-ws-user
List Available Resources	GET /resources	adapa-admin, adapa-ws-user
Get Resource Information	GET /resource/{file_name}	adapa-admin, adapa-ws-user
Get Resource File	GET /resource/{file_name}/source	adapa-admin
Upload New Resource File with POST	POST /resource	adapa-admin
Upload New Resource File with PUT	PUT /resource	adapa-admin
Remove Resource File	DELETE /resource/{file_name}	adapa-admin
Remove All Resource Files	DELETE /resources	adapa-admin

6.2. API

ZEMENTIS REST has three APIs denoted by static path identifiers: `models`, `apply`, and `resources`. Requests in the following examples employ syntax for `cURL`, a popular command line data transfer tool for Unix-like systems, and use username/password credentials `adapa/adapa` with user permissions to execute all REST API operations. All examples also include `Iris_NN` PMML model which can be found in the executable samples package.

6.2.1. JSON Objects

Errors

Error messages container

Properties

errors (`array[string]`): array of strings containing error messages

Example 6.2. ZEMENTIS REST Errors Object

```
{
  "errors": [
    "Model 'Iris_NN' not found."
  ]
}
```

Models Model names container

Properties

models (array[string]): array of strings containing model names

Example 6.3. ZEMENTIS REST Models Object

```
{
  "models": [
    "Iris_NN",
    "Iris_CT",
    "Iris_MLR"
  ]
}
```

ModelInfo Model information

Properties

modelName (string): model name

description (string): model description

isActive (boolean): model currently loaded into memory

inputFields (array[Field]): array of input [Field](#) objects

outputFields (array[Field]): array of output [Field](#) objects

Example 6.4. ZEMENTIS REST ModelInfo Object

```
{
  "modelName": "Iris_NN",
  "description": "Neural Network for multi-class classification using the Iris
dataset",
  "isActive": true,
  "inputFields": [
    {
      "name": "sepal_length",
      "type": "DOUBLE",
      "usage": "ACTIVE"
    },
    {
      "name": "sepal_width",
      "type": "DOUBLE",
      "usage": "ACTIVE"
    },
    {
      "name": "petal_length",
      "type": "DOUBLE",
      "usage": "ACTIVE"
    },
    {
      "name": "petal_width",
      "type": "DOUBLE",
      "usage": "ACTIVE"
    }
  ],
  "outputFields": [
    {
      "name": "class",
      "type": "STRING",
      "usage": "OUTPUT"
    },
    {
      "name": "Probability_setosa",
      "type": "DOUBLE",
      "usage": "OUTPUT"
    },
    {
      "name": "Probability_versicolor",
      "type": "DOUBLE",
      "usage": "OUTPUT"
    },
    {
      "name": "Probability_virginica",
      "type": "DOUBLE",
      "usage": "OUTPUT"
    }
  ]
}
```

Field

Field information

Properties

name (string): field name

type (string): field data type with one of string values: BOOLEAN, INTEGER, FLOAT, DOUBLE, DATE, DATETIME, TIME, or STRING

usage (**string**): field usage type with one of string values: ACTIVE, SUPPLEMENTARY, TARGET, GROUP, DERIVED, or OUTPUT

Example 6.5. ZEMENTIS REST Field Object

```
{
  "name": "petal_width",
  "type": "DOUBLE",
  "usage": "ACTIVE"
}
```

Record

Object used to represent input or output data record as a set of field/value pairs.

Properties

field_name_1 (**string**): optional field/value pair

field_name_2 (**number**): optional field/value pair

field_name_3 (**boolean**): optional field/value pair

field_name... (**date-time**): optional field/value pair

field_name_n (**array[string]**): optional field/value pair

Example 6.6. ZEMENTIS REST Record Object

```
{
  "probability": 0.99995417336,
  "days": 47,
  "class": "shirt",
  "time": "2010-07-14 09:00:02",
  "colors": [ "white", "red", "yellow" ]
}
```

Records

Anonymous array of [Record](#) objects used to represent multiple input or output records.

Example 6.7. ZEMENTIS REST Record Object

```
[
  {
    "Probability_virginica": 2.536692637033178E-13,
    "class": "Iris-setosa",
    "Probability_setosa": 0.9995535104664939,
    "Probability_versicolor": 4.464895332525406E-4
  },
  {
    "Probability_virginica": 1.0465677336558733E-12,
    "class": "Iris-setosa",
    "Probability_setosa": 0.9985890830740689,
    "Probability_versicolor": 0.0014109169248845744
  },
  {
    "Probability_virginica": 4.111504068226951E-13,
    "class": "Iris-setosa",
    "Probability_setosa": 0.9993451737365701,
    "Probability_versicolor": 6.54826263018726E-4
  }
]
```

Result Object used to return the result of applying a model to data.

Properties

modelName (string): model name

outputs (array[Record]): array of output [Record](#) objects

Example 6.8. ZEMENTIS REST Result Object

```
{
  "model": "Iris_NN",
  "outputs": [
    {
      "Probability_virginica": 2.536692637033178E-13,
      "class": "Iris-setosa",
      "Probability_setosa": 0.9995535104664939,
      "Probability_versicolor": 4.464895332525406E-4
    },
    {
      "Probability_virginica": 1.0465677336558733E-12,
      "class": "Iris-setosa",
      "Probability_setosa": 0.9985890830740689,
      "Probability_versicolor": 0.0014109169248845744
    },
    {
      "Probability_virginica": 4.111504068226951E-13,
      "class": "Iris-setosa",
      "Probability_setosa": 0.9993451737365701,
      "Probability_versicolor": 6.54826263018726E-4
    },
    {
      "Probability_virginica": 6.620361333170605E-13,
      "class": "Iris-setosa",
      "Probability_setosa": 0.9990465573403722,
      "Probability_versicolor": 9.534426589658814E-4
    }
  ]
}
```

ResourceInfo Resource file information

Properties

fileName (string): file name

resourceType (string): resource type

resourceIdentifier (string): resource identifier

resourceNames (array[string]): array of resource names

Example 6.9. ZEMENTIS REST ResourceInfo Object

```
{
  "fileName": "ECommerceFraud_NN.xls",
  "resourceType": "Lookup Tables",
  "resourceIdentifier": "Table Name",
  "resourceNames": [
    "StatePoints"
  ]
}
```

Resources Anonymous array of [ResourceInfo](#) objects.

Example 6.10. ZEMENTIS REST Resources Object

```
{
  "resources": [
    {
      "fileName": "ECommerceFraud_NN.xls",
      "resourceType": "Lookup Tables",
      "resourceIdentifier": "Table Name",
      "resourceNames": [
        "StatePoints"
      ]
    },
    {
      "fileName": "ECommerceFraud_NN.jar",
      "resourceType": "Custom Functions",
      "resourceIdentifier": "Function Namespace",
      "resourceNames": [
        "fraud"
      ]
    }
  ]
}
```

6.2.2. Operations on Models

6.2.2.1. List Available Models

Definition

GET /models

This operation retrieves the model names of all the available PMML models in ZEMENTIS. Use these model names as identifiers for all operations requiring the `model_name` path variable.

Request Parameters

None

Returns

Returns [Models](#) object if successful, [Errors](#) otherwise.

Example 6.11. ZEMENTIS REST List Models

Request

```
curl -u adapa:adapa -k https://localhost/adapars/models
```

Request Header

```
GET /adapars/models HTTP/1.1
Authorization: Basic YWRhcGE6YWRhcGE=
User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r zlib/1.2.5
Host: localhost
Accept: */*
```

Response Header

```
HTTP/1.1 200 OK
```

```
Server: Apache-Coyote/1.1
Pragma: No-cache
Cache-Control: no-cache
Expires: Wed, 31 Dec 1969 16:00:00 PST
X-Powered-By: Servlet 2.5; JBoss-5.0/JBossWeb-2.1
Content-Type: application/json
Content-Length: 53
Date: Wed, 26 Mar 2014 18:20:09 GMT
```

Response Body

```
{
  "models" : [ "Iris_NN", "Iris_CT", "Iris_MLR" ]
}
```

6.2.2.2. Get Model Information

Definition	GET /model/{model_name} Get model name, description, and information about input, output, or derived fields.
Request Parameters	model_name (string): required path variable for existing model name
Returns	Returns ModellInfo object if successful, Errors otherwise.

Example 6.12. ZEMENTIS REST Get Model Information

Request

```
curl -u adapa:adapa -k https://localhost/adapars/model/Iris_NN
```

Request Header

```
GET /adapars/model/Iris_NN HTTP/1.1
Authorization: Basic YWRhcGE6YWRhcGE=
User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r zlib/1.2.5
Host: localhost
Accept: */*
```

Response Header

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Pragma: No-cache
Cache-Control: no-cache
Expires: Wed, 31 Dec 1969 16:00:00 PST
X-Powered-By: Servlet 2.5; JBoss-5.0/JBossWeb-2.1
Content-Type: application/json
Content-Length: 1969
Date: Wed, 26 Mar 2014 18:39:57 GMT
```

Response Body

```
{
  "modelName" : "Iris_NN",
  "description" : "Neural Network for multi-class classification using the Iris dataset",
  "isActive": true,
  "inputFields" : [ {
    "name" : "sepal_length",
    "type" : "DOUBLE",
```

```

    "usage" : "ACTIVE"
  }, {
    "name" : "sepal_width",
    "type" : "DOUBLE",
    "usage" : "ACTIVE"
  }, {
    ...

```

6.2.2.3. Get Model Source

Definition	GET /model/{model_name}/source
	Get annotated or original PMML file. Annotated source may contain warning or error messages embedded in XML comments that are useful for verifying that the PMML code is correct.
Request Parameters	model_name (string): required path variable for existing model name annotated (boolean): optional query parameter used to request the annotated version of the PMML file.
Returns	Returns the PMML source code if successful, Errors otherwise.

Example 6.13. ZEMENTIS REST Get Model Source

Request

```
curl -u adapa:adapa -k https://localhost/adapars/model/Iris_NN/source?annotated=true
```

Request Header

```

GET /adapars/model/Iris_NN/source?annotated=true HTTP/1.1
Authorization: Basic YWRhcGE6YWRhcGE=
User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r zlib/1.2.5
Host: localhost
Accept: */*

```

Response Header

```

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Pragma: No-cache
Cache-Control: no-cache
Expires: Wed, 31 Dec 1969 19:00:00 EST
X-Powered-By: Servlet 2.5; JBoss-5.0/JBossWeb-2.1
Content-Type: application/xml
Content-Length: 7983
Date: Wed, 26 Mar 2014 20:44:04 GMT

```

Response Body

```

<?xml version="1.0" encoding="UTF-8"?>
<!--(Comment generated by ADAPA) PMML processed by ADAPA (Version : 4.2)-->
<PMML version="4.2"
  xsi:schemaLocation="http://www.dmg.org/PMML-4_2 http://www.dmg.org/v4-2/pmml-4-2.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://www.dmg.org/PMML-4_2">
  <Header copyright="Copyright (c) 2008-2014 Zementis, Inc. (www.zementis.com)"

```

```

        description="Neural Network for multi-class classification using the Iris dataset">
    <Timestamp>Feb 15, 2008</Timestamp>
</Header>
<DataDictionary numberOfFields="5">
  <DataField dataType="string" name="target_class" optype="categorical">
    <Value value="Iris-setosa"/>
    <Value value="Iris-versicolor"/>
    <Value value="Iris-virginica"/>
  </DataField>
  <DataField dataType="double" name="sepal_length" optype="continuous"/>
  <DataField dataType="double" name="sepal_width" optype="continuous"/>
  ...

```

6.2.2.4. Get Model Serialized Source

Definition	GET /model/{model_name}/serialized Get binary file containing serialized representation of the model.
Request Parameters	model_name (string) : required path variable for existing model name
Returns	Returns the binary file if successful, Errors otherwise.

Example 6.14. ZEMENTIS REST Get Model Serialized

Request

```
curl -u adapa:adapa -k https://localhost/adapars/model/Iris_NN/serialized
```

Request Header

```

GET /adapars/model/Iris_NN/serialized HTTP/1.1
Authorization: Basic YWRhcGE6YWRhcGE=
User-Agent: curl/7.54.0
Host: localhost
Accept: */*

```

Response Header

```

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Pragma: No-cache
Cache-Control: private
Expires: Wed, 31 Dec 1969 19:00:00 EST
X-Powered-By: Servlet 2.5; JBoss-5.0/JBossWeb-2.1
Content-Type: application/octet-stream
Transfer-Encoding: chunked
Date: Wed, 09 Aug 2017 22:44:48 GMT

```

Response Body

```

BINARY DATA
...

```

6.2.2.5. Upload New Model with POST

Definition	POST /model
------------	--------------------

Upload new PMML model. Resulting identifier for this model is extracted from optional PMML attribute `modelName` if specified or `file` body parameter name otherwise. If the PMML file is large, such as Random Forest model, we recommend compressing the file using ZIP/GZIP before uploading. This will reduce the upload time dramatically.

Request Parameters

Content-Type (string): required header parameter with two accepted values: `application/octet-stream` or `multipart/form-data`

file (string): required query parameter for PMML file name, if Content-Type is `application/octet-stream`, or a body parameter in `multipart/form-data` content encoding

applyCleanser (boolean): optional parameter used to automatically perform comprehensive syntactic and semantic checks, correct known issues and convert your PMML file to version 4.3 (default is `true`)

Returns

Returns a [ModelInfo](#) object, 201 HTTP status code, and a response header entry `Location` with the URI of the created resource if the upload was successful. If the uploaded model was a valid XML but an invalid PMML, 200 HTTP status code and error annotated PMML source is returned, [Errors](#) otherwise.

Example 6.15. ZEMENTIS REST Upload New Model with POST

Request

```
curl -u adapa:adapa -k https://localhost/adapars/model?file=Iris_NN.pmml -X POST -T Iris_NN.pmml \
  -H "Content-Type:application/octet-stream"
curl -u adapa:adapa -k https://localhost/adapars/model -X POST -F file=@Iris_NN.pmml
curl -u adapa:adapa -k https://localhost/adapars/model?applyCleanser=true -X POST -F
file=@Iris_NN.pmml
```

Request Header

```
POST /adapars/model HTTP/1.1
Authorization: Basic YWRhcGE6YWRhcGE=
User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r zlib/1.2.5
Host: localhost
Accept: */*
Content-Length: 9265
Expect: 100-continue
Content-Type: multipart/form-data; boundary=-----1ff14caee8ae
```

Response Header

```
HTTP/1.1 201 Created
Server: Apache-Coyote/1.1
X-Powered-By: Servlet 2.5; JBoss-5.0/JBossWeb-2.1
Location: https://localhost/adapars/model/Iris_NN
Content-Type: application/json
Content-Length: 836
Date: Wed, 26 Mar 2014 19:45:18 GMT
```

Response Body

```
{
  "modelName" : "Iris_NN",
  "description" : "Neural Network for multi-class classification using the Iris dataset",
  "inputFields" : [ {
    "name" : "sepal_length",
    "type" : "DOUBLE",
    "usage" : "ACTIVE"
  }, {
    "name" : "sepal_width",
    "type" : "DOUBLE",
    "usage" : "ACTIVE"
  }, {
    ...
  }
}
```

6.2.2.6. Upload New Model with PUT

Definition

PUT /model

Upload new PMML model. Resulting identifier for this model is extracted from optional PMML attribute `modelName` if specified or `file` query parameter name otherwise. If the PMML file is large, such as the Random Forest model, we recommend compressing the file using ZIP/GZIP before uploading. This will reduce the upload time dramatically.

Request Parameters

file (string): required query parameter for PMML file name
applyCleanser (boolean): optional parameter used to automatically perform comprehensive syntactic and semantic checks, correct known issues and convert your PMML file to version 4.3 (default is `true`)

Returns

Returns a [ModelInfo](#) object, 201 HTTP status code, and a response header entry `Location` with the URI of the created resource if the upload was successful. If the uploaded model was a valid XML but the PMML was invalid, a 200 HTTP status code and with errors annotated PMML file is returned, [Errors](#) otherwise.

Example 6.16. ZEMENTIS REST Upload New Model with PUT

Request

```
curl -u adapa:adapa -k 'https://localhost/adapars/model?file=Iris_NN.pmml' -X PUT -T Iris_NN.pmml
curl -u adapa:adapa -k 'https://localhost/adapars/model?file=Iris_NN.pmml&applyCleanser=true' -X PUT -T Iris_NN.pmml
```

Request Header

```
PUT /adapars/model?file=Iris_NN.pmml HTTP/1.1
Authorization: Basic YWRhcGE6YWRhcGE=
User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r zlib/1.2.5
```

```
Host: localhost
Accept: */*
Content-Length: 9061
Expect: 100-continue
```

Response Header

```
HTTP/1.1 201 Created
Server: Apache-Coyote/1.1
Pragma: No-cache
Cache-Control: no-cache
Expires: Wed, 31 Dec 1969 19:00:00 EST
X-Powered-By: Servlet 2.5; JBoss-5.0/JBossWeb-2.1
Location: https://localhost/adapars/model/Iris_NN
Content-Type: application/json
Content-Length: 836
Date: Wed, 26 Mar 2014 19:56:26 GMT
```

Response Body

```
{
  "modelName" : "Iris_NN",
  "description" : "Neural Network for multi-class classification using the Iris dataset",
  "inputFields" : [ {
    "name" : "sepal_length",
    "type" : "DOUBLE",
    "usage" : "ACTIVE"
  }, {
    "name" : "sepal_width",
    "type" : "DOUBLE",
    "usage" : "ACTIVE"
  }, {
    ...
  }
}
```

6.2.2.7. Activate an existing Model with PUT

Definition **PUT /model/{model_name}/activate**

Activates the model with name `modelName` if it was inactive. Activating an active model has no effect. After activation, the model is immediately available for handling data processing requests. Please note an active model consumes runtime resources, especially Heap.

Request Parameters **model_name (string)** : required path variable for existing model name

Returns Returns a [ModelInfo](#) object and 200 HTTP status code.

Example 6.17. ZEMENTIS REST Activate an existing Model

Request

```
curl -u adapa:adapa -k https://localhost/adapars/model/Iris_NN/activate -X PUT
```

Request Header

```
PUT /adapars/model/Iris_NN/activate HTTP/1.1
Authorization: Basic YWRhcGE6YWRhcGE=
User-Agent: curl/7.54.0
Host: localhost
Accept: */*
```

Response Header

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Cache-Control: private
Expires: Wed, 31 Dec 1969 19:00:00 EST
Content-Type: application/json
Content-Length: 7166
Date: Wed, 09 Aug 2017 22:44:48 GMT
```

Response Body

```
{
  "modelName" : "Iris_NN",
  "description" : "Neural Network for multi-class classification using the Iris dataset",
  "isActive" : true,
  ...
}
```

6.2.2.8. Deactivate an existing Model with PUT

Definition

PUT /model/{model_name}/deactivate

De-activates the model with name `modelName` by making it inactive. After de-activation, the model is still available, but it no longer consumes runtime resources, especially Heap. An inactive model can be made active by this rest request. Alternatively, if a data processing request comes in for an inactive model, it is automatically made active. Deactivating an inactive model has no effect.

Request Parameters

model_name (string) : required path variable for existing model name

Returns

Returns a [ModelInfo](#) object and 200 HTTP status code.

Example 6.18. ZEMENTIS REST Deactivate an existing Model

Request

```
curl -u adapa:adapa -k https://localhost/adapars/model/Iris_NN/deactivate -X PUT
```

Request Header

```
PUT /adapars/model/Iris_NN/deactivate HTTP/1.1
Authorization: Basic YWRhcGE6YWRhcGE=
User-Agent: curl/7.54.0
Host: localhost
Accept: */*
```

Response Header


```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Cache-Control: private
Expires: Wed, 31 Dec 1969 19:00:00 EST
Content-Type: application/json
Content-Length: 7166
Date: Wed, 09 Aug 2017 22:44:48 GMT
```

Response Body

```
{
  "modelName" : "Iris_NN",
  "description" : "Neural Network for multi-class classification using the Iris dataset",
  "isActive" : false,
  ...
}
```

6.2.2.9. Remove Model

Definition `DELETE /model/{model_name}`

Remove the specified model and list the remaining models.

Request Parameters `model_name (string)`: required path variable for existing model name

Returns Returns a [Models](#) object with a list of remaining model names if successful, [Errors](#) object otherwise.

Example 6.19. ZEMENTIS REST Remove Model

Request

```
curl -u adapa:adapa -k https://localhost/adapars/model/Iris_NN -X DELETE
```

Request Header

```
DELETE /adapars/model/Iris_NN HTTP/1.1
Authorization: Basic YWRhcGE6YWRhcGE=
User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r zlib/1.2.5
Host: localhost
Accept: */*
```

Response Header

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Pragma: No-cache
Cache-Control: no-cache
Expires: Wed, 31 Dec 1969 19:00:00 EST
X-Powered-By: Servlet 2.5; JBoss-5.0/JBossWeb-2.1
Content-Type: application/json
Content-Length: 42
Date: Wed, 26 Mar 2014 19:53:50 GMT
```

Response Body

```
{
  "models" : [ "Iris_CT", "Iris_MLR" ]
}
```

6.2.2.10. Remove All Models

Definition	DELETE /models Remove all available models and list the remaining models.
Request Parameters	None
Returns	Returns a Models object with an empty <code>models</code> array if successful, an Errors object otherwise.

Example 6.20. ZEMENTIS REST Remove All Models

Request

```
curl -u adapa:adapa -k https://localhost/adapars/models -X DELETE
```

Request Header

```
DELETE /adapars/models HTTP/1.1
Authorization: Basic YWRhcGE6YWRhcGE=
User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r zlib/1.2.5
Host: localhost
Accept: */*
```

Response Header

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Pragma: No-cache
Cache-Control: no-cache
Expires: Wed, 31 Dec 1969 19:00:00 EST
X-Powered-By: Servlet 2.5; JBoss-5.0/JBossWeb-2.1
Content-Type: application/json
Content-Length: 20
Date: Wed, 26 Mar 2014 20:01:42 GMT
```

Response Body

```
{
  "models" : [ ]
}
```

6.2.3. Apply model

6.2.3.1. Apply Model to Single Record

Definition	GET /apply/{model_name}
------------	--------------------------------

Apply a model to a single JSON input record.

Request Parameters **model_name (string):** required path variable for name of the model to be applied
record (Record): optional query parameter for input [Record](#)

Returns Returns [Result](#) object if successful, [Errors](#) otherwise.

Example 6.21. ZEMENTIS REST Apply Model to Single Record

Request

```
curl -u adapa:adapa -k https://localhost/adapars/apply/Iris_NN -G --data-urlencode \
  'record={"sepal_length":5.1,"sepal_width":3.5,"petal_length":1.4,"petal_width":0.2}'
```

Request Header

```
GET /adapars/apply/Iris_NN?record=%7B%22sepal_length%22%3A5.1%2C%22sepal_width%22%3A3.5%2C%22
  petal_length%22%3A1.4%2C%22petal_width%22%3A0.2%7D HTTP/1.1
Authorization: Basic YWRhcGE6YWRhcGE=
User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r zlib/1.2.5
Host: localhost
Accept: */*
```

Response Header

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Pragma: No-cache
Cache-Control: no-cache
Expires: Wed, 31 Dec 1969 19:00:00 EST
X-Powered-By: Servlet 2.5; JBoss-5.0/JBossWeb-2.1
Content-Type: application/json
Content-Length: 231
Date: Wed, 26 Mar 2014 20:10:30 GMT
```

Response Body

```
{
  "model" : "Iris_NN",
  "outputs" : [ {
    "Probability_virginica" : 2.536692637033178E-13,
    "class" : "Iris-setosa",
    "Probability_setosa" : 0.9995535104664939,
    "Probability_versicolor" : 4.464895332525406E-4
  } ]
}
```

6.2.3.2. Apply Model to Single Record and Explain Result

Definition **GET /apply/{model_name}/explain**

Apply model to a single JSON input record and get the result with details of the performed computation in plain text. Useful for debugging PMML code.

Request Parameters **model_name (string):** required path variable for name of the model to be applied

record (**Record**): optional query parameter for input **Record**

Returns Returns a result in plain text if successful, **Errors** otherwise.

Example 6.22. ZEMENTIS REST Apply Model to Single Record and Explain Result

Request

```
curl -u adapa:adapa -k https://localhost/adapars/apply/Iris_NN/explain -G --data-urlencode \
  'record={"sepal_length":5.1,"sepal_width":3.5,"petal_length":1.4,"petal_width":0.2}'
```

Request Header

```
GET /adapars/apply/Iris_NN/explain?record=%7B%22sepal_length%22%3A5.1%2C%22sepal_width%22%3A3.5%2C%22
  petal_length%22%3A1.4%2C%22petal_width%22%3A0.2%7D HTTP/1.1
Authorization: Basic YWRhcGE6YWRhcGE=
User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r zlib/1.2.5
Host: localhost
Accept: */*
```

Response Header

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Pragma: No-cache
Cache-Control: no-cache
Expires: Wed, 31 Dec 1969 19:00:00 EST
X-Powered-By: Servlet 2.5; JBoss-5.0/JBossWeb-2.1
Content-Type: text/plain
Content-Length: 1361
Date: Wed, 26 Mar 2014 20:13:34 GMT
```

Response Body

```
[sepal_length] := 5.1 (DOUBLE)
[sepal_width] := 3.5 (DOUBLE)
[petal_length] := 1.4 (DOUBLE)
[petal_width] := 0.2 (DOUBLE)

[MiningSchema]
[sepal_length] := 5.1 (DOUBLE)
[sepal_width] := 3.5 (DOUBLE)
[petal_length] := 1.4 (DOUBLE)
[petal_width] := 0.2 (DOUBLE)

[LocalTransformations]
[derived_sepal_length] := 0.22222222222222213 (DOUBLE)
[derived_sepal_width] := 0.6818181818181818 (DOUBLE)
[derived_petal_length] := 0.07017543859649121 (DOUBLE)
[derived_petal_width] := 0.04166666666666667 (DOUBLE)

[BackPropagationNetwork]
Value of neural input [3] is [0.042].
Value of neural input [2] is [0.07].
Value of neural input [1] is [0.682].
Value of neural input [0] is [0.222].
Value of output neuron [11] in the last neural layer is [1].
Value of output neuron [12] in the last neural layer is [0].
Value of output neuron [13] in the last neural layer is [0].

[Output]
The [predictedValue] is [Iris-setosa (STRING)]
[class] := Iris-setosa (STRING)
The [probability] of [Iris-setosa (STRING)] is [0.9995535104664939 (DOUBLE)]
[Probability_setosa] := 0.9995535104664939 (DOUBLE)
```

```
The [probability] of [Iris-versicolor (STRING)] is [4.464895332525406E-4 (DOUBLE)]
[Probability_versicolor] := 4.464895332525406E-4 (DOUBLE)
The [probability] of [Iris-virginica (STRING)] is [2.536692637033178E-13 (DOUBLE)]
[Probability_virginica] := 2.536692637033178E-13 (DOUBLE)
```

6.2.3.3. Apply Model to Multiple Records or Apply Model to Single Binary Data with POST

Definition

POST /apply/{model_name}

This provides two kinds of operations. Generally, if a predictive model without `binary` type input is applied, this will be a batch 'apply' operation that streams multiple input records to ZEMENTIS. ZEMENTIS will automatically detect `Comma Separated Value (CSV)` or `JSON Records` formatted input and stream results back in the same format unless otherwise specified in the `Accept` request header parameter with `text/csv` or `application/json` values. Compressing input data with `zip` or `gzip` will result in the same compression method for the returned output stream.

If a predictive model with a `binary` type input is applied, this will be a single 'apply' operation that processes a single binary source as input to ZEMENTIS.

Request Parameters

Content-Type (string): required header parameter with two accepted values: `application/octet-stream` or `multipart/form-data`

model_name (string): required path variable for the name of the model to be applied

maxThreads: optional query parameter for specifying the maximum number of concurrent threads (default value is twice the number of processor cores). No impact if a predictive model with a `binary` type input was applied.

maxRecordsPerThread: optional query parameter for specifying the maximum number of records processed by a thread in batch (default value is 5000). No impact if a predictive model with a `binary` type input was applied.

Accept: optional header parameter for explicitly specifying `text/csv` or `application/json` output format

User-Agent: optional header parameter for full duplex HTTP streaming data if set to `AdapaStreaming` followed by any characters or a string containing value `curl`. Default data handling mode is `copy-forward` where response is rendered only after full request has been read by the server.

Returns

Returns results as `CSV` or as `Result` object if successful, `Errors` otherwise.

Example 6.23. ZEMENTIS REST Apply Model to Multiple Records with POST

Request

```
curl -u adapa:adapa -k https://localhost/adapars/apply/Iris_NN -X POST -T Iris_NN.csv \  
-H "Content-Type:application/octet-stream"  
curl -u adapa:adapa -k https://localhost/adapars/apply/Iris_NN?maxThreads=8 -X POST -F  
file=@Iris_NN.csv
```

Request Header

```
POST /adapars/apply/Iris_NN?maxThreads=8 HTTP/1.1  
Authorization: Basic YWRhcGE6YWRhcGE=  
User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r zlib/1.2.5  
Host: localhost  
Accept: */*  
Content-Length: 10148  
Expect: 100-continue  
Content-Type: multipart/form-data; boundary=-----6da946996e0d
```

Response Header

```
HTTP/1.1 200 OK  
Server: Apache-Coyote/1.1  
X-Powered-By: Servlet 2.5; JBoss-5.0/JBossWeb-2.1  
Content-Type: text/csv  
Transfer-Encoding: chunked  
Date: Wed, 26 Mar 2014 20:19:23 GMT
```

Response Body

```
class,Probability_setosa,Probability_versicolor,Probability_virginica  
Iris-setosa,0.9995535104664939,4.464895332525406E-4,2.536692637033178E-13  
Iris-setosa,0.9985890830740689,0.0014109169248845744,1.0465677336558733E-12  
Iris-setosa,0.9993451737365701,6.54826263018726E-4,4.111504068226951E-13  
...
```

Example 6.24. ZEMENTIS REST Apply Model to Single Binary Record with POST

Request

```
curl -u adapa:adapa -k https://localhost/adapars/apply/Caffe_NN -X POST -H 'Accept:application/json' -  
F file=@0.jpg
```

Request Header

```
POST /adapars/apply/Caffe_NN HTTP/1.1  
Authorization: Basic YWRhcGE6YWRhcGE=  
User-Agent: curl/7.43.0  
Host: localhost  
Accept: application/json  
Content-Length: 5319  
Expect: 100-continue  
Content-Type: multipart/form-data; boundary=-----6099e489fd2da819
```

Response Header

```
HTTP/1.1 200 OK  
Server: Apache-Coyote/1.1  
X-Powered-By: Servlet 2.5; JBoss-5.0/JBossWeb-2.1  
Content-Type: application/json  
Content-Length: 403  
Date: Fri, 27 May 2016 21:39:07 GMT
```

Response Body

```
{
  "model" : "Caffe_NN",
  "outputs" : [ {
    "p_7" : 0.009013318755324183,
    "p_8" : 0.011660178735845163,
    "p_9" : 0.040489440800734404,
    "p_0" : 0.7602463077131643,
    "class" : "0",
    "p_1" : 0.006724422031736871,
    "p_2" : 0.052489690530517254,
    "p_3" : 0.004134235496422808,
    "p_4" : 0.027965981244545225,
    "p_5" : 0.014539398304602753,
    "p_6" : 0.07273702638710705
  } ]
}
```

6.2.3.4. Apply Model to Multiple Records or Apply Model to Single Binary Data with PUT

Definition

PUT /apply/{model_name}

This provides two kinds of operations. Generally, if a predictive model without `binary` type input was applied, this is a batch 'apply' operation that streams multiple input records to the ZEMENTIS Server. ZEMENTIS will automatically detect `Comma Separated Value (CSV)` or `JSON Records` formatted input and stream results back in the same format unless otherwise specified in the `Accept` request header parameter with `text/csv` or `application/json` values. Compressing input data with `zip` or `gzip` will result in the same compression method for the returned output stream.

If a predictive model with a `binary` type input is applied, this will be a single 'apply' operation that processes a single binary source as input to ZEMENTIS.

Request Parameters

model_name (string): required path variable for name of the model to be applied

maxThreads: optional query parameter for specifying the maximum number of concurrent threads (default value is twice the number of processor cores). No impact if a predictive model with a `binary` type input was applied.

maxRecordsPerThread: optional query parameter for specifying the maximum number of records processed by a thread in batch (default value is 5000). No impact if a predictive model with a `binary` type input was applied.

Accept: optional header parameter for explicitly specifying `text/csv` or `application/json` output format

Returns

Returns results as `CSV` or as `Result` object if successful, `Errors` otherwise.

Example 6.25. ZEMENTIS REST Apply Model to Multiple Records with PUT

Request

```
curl -u adapa:adapa -k 'https://localhost/adapars/apply/Iris_NN?maxThreads=8&maxRecordsPerThread=1000' \
  -X PUT -T Iris_NN.csv -H 'Accept:application/json'
```

Request Header

```
PUT /adapars/apply/Iris_NN?maxThreads=8&maxRecordsPerThread=1000 HTTP/1.1
Authorization: Basic YWRhcGE6YWRhcGE=
User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r zlib/1.2.5
Host: localhost
Accept:application/json
Content-Length: 9945
Expect: 100-continue
```

Response Header

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Pragma: No-cache
Cache-Control: no-cache
Expires: Wed, 31 Dec 1969 19:00:00 EST
X-Powered-By: Servlet 2.5; JBoss-5.0/JBossWeb-2.1
Content-Type: application/json
Transfer-Encoding: chunked
Date: Wed, 26 Mar 2014 20:25:43 GMT
```

Response Body

```
{
  "model" : "Iris_NN",
  "outputs" : [ {
    "Probability_virginica" : 2.536692637033178E-13,
    "class" : "Iris-setosa",
    "Probability_setosa" : 0.9995535104664939,
    "Probability_versicolor" : 4.464895332525406E-4
  }, {
    "Probability_virginica" : 1.0465677336558733E-12,
    "class" : "Iris-setosa",
    "Probability_setosa" : 0.9985890830740689,
    "Probability_versicolor" : 0.0014109169248845744
  }, {
    ...
  }
}
```

Example 6.26. ZEMENTIS REST Apply Model to Single Binary Record with PUT

Request

```
curl -u adapa:adapa -k 'https://localhost/adapars/apply/Caffe_NN -X PUT -T 0.jpg -H 'Accept:text/csv'
```

Request Header

```
PUT /adapars/apply/Caffe_NN HTTP/1.1
Authorization: Basic YWRhcGE6YWRhcGE=
User-Agent: curl/7.43.0
Host: localhost
Accept:text/csv
Content-Length: 5136
```



```
Expect: 100-continue
```

Response Header

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Pragma: No-cache
Cache-Control: no-cache
Expires: Wed, 31 Dec 1969 16:00:00 PST
X-Powered-By: Servlet 2.5; JBoss-5.0/JBossWeb-2.1
Content-Type: text/csv
Transfer-Encoding: chunked
Date: Fri, 27 May 2016 21:45:27 GMT
```

Response Body

```
class,p_0,p_1,p_2,p_3,p_4,p_5,p_6,p_7,p_8,p_9
0,0.7602463077131643,0.006724422031736871,...
```

6.2.3.5. Asynchronously Apply Model to Multiple Records with POST

Definition

POST /apply/{model_name}/async

This is an asynchronous batch 'apply' operation that streams multiple input records from remote location specified in uploaded properties file and writes the result back to the remote data target. The properties file describes the remote data source and target locations, connection properties, and access credentials. ZEMENTIS will automatically detect Comma Separated Value (CSV) or JSONRecords formatted input and streams the result back in CSV format. Compressing input data with `zip` or `gzip` will result in the same compression method for the result.

Request Parameters

Content-Type (string): required header parameter with two accepted values: `application/octet-stream` or `multipart/form-data`

model_name (string): required path variable for the name of the model to be applied

maxThreads: optional query parameter for specifying the maximum number of concurrent threads (default value is twice the number of processor cores).

maxRecordsPerThread: optional query parameter for specifying the maximum number of records processed by a thread in batch (default value is 5000).

Returns

Returns status information, job ID and description, output handle, and start timestamp of processing job in JSON format.

Example 6.27. ZEMENTIS REST Asynchronously Apply Model to Multiple Records with POST

Request

```
curl -u adapa:adapa -k https://localhost/adapars/apply/Iris_NN/async -X POST -T Iris_NN_CSV.properties \
  -H "Content-Type:application/octet-stream"
curl -u adapa:adapa -k https://localhost/adapars/apply/Iris_NN/async?maxThreads=8 -X POST \
  -F file=@Iris_NN_CSV.properties
```

Request Header

```
POST /adapars/apply/Iris_NN/async?maxThreads=8 HTTP/1.1
Authorization: Basic YWRhcGE6YWRhcGE=
User-Agent: curl/7.54.0
Host: localhost
Accept: */*
Content-Length: 376
Expect: 100-continue
Content-Type: multipart/form-data; boundary=-----c6e69656a61898e9
```

Response Header

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: application/json
Date: Thu, 10 Aug 2017 16:24:42 GMT
```

Response Body

```
{
  "status" : "STARTED",
  "id" : 4,
  "output" : "Iris_NN_output_4_20170810_092441.csv",
  "startTime" : "2017-08-10 09:24:41.595 -0700",
  "description" : "Amazon S3 Connector: bucket='myBucket', input='Iris_NN.csv'"
}
```

6.2.3.6. Asynchronously Apply Model to Multiple Records with PUT

Definition

PUT /apply/{model_name}/async

This is an asynchronous batch 'apply' operation that streams multiple input records from remote location specified in uploaded properties file and writes the result back to the remote data target. The properties file describes the remote data source and target locations, connection properties, and access credentials. ZEMENTIS will automatically detect Comma Separated Value (CSV) or JSONRecords formatted input and streams the result back in CSV format. Compressing input data with zip or gzip will result in the same compression method for the result.

Request Parameters

model_name (string): required path variable for the name of the model to be applied

maxThreads: optional query parameter for specifying the maximum number of concurrent threads (default value is twice the number of processor cores).

maxRecordsPerThread: optional query parameter for specifying the maximum number of records processed by a thread in batch (default value is 5000).

Accept: header parameter for `application/json` result information of remote repository.

Returns Returns status information, job ID and description, output handle, and start timestamp of processing job in JSON format.

Example 6.28. ZEMENTIS REST Asynchronously Apply Model to Multiple Records with PUT

Request

```
curl -u adapa:adapa -k https://localhost/adapars/apply/Iris_NN/async -X PUT -T Iris_NN_CSV.properties
```

Request Header

```
PUT /adapars/apply/Iris_NN/async HTTP/1.1
Authorization: Basic YWRhcGE6YWRhcGE=
User-Agent: curl/7.54.0
Host: localhost
Accept: */*
Content-Length: 154
Expect: 100-continue
```

Response Header

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Pragma: No-cache
Cache-Control: private
Expires: Wed, 31 Dec 1969 19:00:00 EST
Content-Type: application/json
Transfer-Encoding: chunked
Date: Thu, 10 Aug 2017 16:52:43 GMT
```

Response Body

```
{
  "status" : "STARTED",
  "id" : 5,
  "output" : "Iris_NN_output_5_20170810_095243.csv",
  "startTime" : "2017-08-10 09:52:43.205 -0700",
  "description" : "Amazon S3 Connector: bucket='myBucket', input='Iris_NN.csv'"
}
```

6.2.4. Operations on Resources

6.2.4.1. List Available Resources

Definition

GET /resources

This operation retrieves information on all available resource files uploaded on ZEMENTIS Server. Use file names as identifiers for all operations requiring a `file_name` path variable.

Request Parameters None

Returns Returns a [Resources](#) object if successful, an [Errors](#) object otherwise.

Example 6.29. ZEMENTIS REST List Resources

Request

```
curl -u adapa:adapa -k https://localhost/adapars/resources
```

Request Header

```
GET /adapars/resources HTTP/1.1
Authorization: Basic YWRhcGE6YWRhcGE=
User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r zlib/1.2.5
Host: localhost
Accept: */*
```

Response Header

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Pragma: No-cache
Cache-Control: no-cache
Expires: Wed, 31 Dec 1969 16:00:00 PST
X-Powered-By: Servlet 2.5; JBoss-5.0/JBossWeb-2.1
Content-Type: application/json
Content-Length: 363
Date: Mon, 24 Nov 2014 22:56:50 GMT
```

Response Body

```
{
  "resources" : [ {
    "fileName" : "ECommerceFraud_NN.jar",
    "resourceType" : "Custom Functions",
    "resourceIdentifier" : "Function Namespace",
    "resourceNames" : [ "fraud" ]
  }, {
    "fileName" : "ECommerceFraud_NN.xls",
    "resourceType" : "Lookup Tables",
    "resourceIdentifier" : "Table Name",
    "resourceNames" : [ "StatePoints" ]
  } ]
}
```

6.2.4.2. Get Resource Information

Definition **GET /resource/{file_name}**

Get information on the specified resource file.

Request Parameters **file_name (string):** required path variable for an existing resource file name

Returns Returns a [ResourceInfo](#) object if successful, an [Errors](#) object otherwise.

Example 6.30. ZEMENTIS REST Get Resource Information

Request

```
curl -u adapa:adapa -k https://localhost/adapars/resource/ECommerceFraud_NN.jar
```

Request Header

```
GET /adapars/resources/ECommerceFraud_NN.jar HTTP/1.1
Authorization: Basic YWRhcGE6YWRhcGE=
User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r zlib/1.2.5
Host: localhost
Accept: */*
```

Response Header

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Pragma: No-cache
Cache-Control: no-cache
Expires: Wed, 31 Dec 1969 16:00:00 PST
X-Powered-By: Servlet 2.5; JBoss-5.0/JBossWeb-2.1
Content-Type: application/json
Content-Length: 161
Date: Mon, 24 Nov 2014 23:05:51 GMT
```

Response Body

```
{
  "fileName" : "ECommerceFraud_NN.jar",
  "resourceType" : "Custom Functions",
  "resourceIdentifier" : "Function Namespace",
  "resourceNames" : [ "fraud" ]
}
```

6.2.4.3. Get Resource File

Definition **GET /resource/{file_name}/source**

Download a resource file.

Request Parameters **file_name (string):** required path variable for an existing resource file name

Returns Returns a copy of the resource file if successful, an [Errors](#) object otherwise.

Example 6.31. ZEMENTIS REST Get Resource File

Request

```
curl -u adapa:adapa -k https://localhost/adapars/resource/ECommerceFraud_NN.jar/source
```

Request Header

```
GET /adapars/resources/ECommerceFraud_NN.jar/source HTTP/1.1
Authorization: Basic YWRhcGE6YWRhcGE=
User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r zlib/1.2.5
Host: localhost
Accept: */*
```

Response Header

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Pragma: No-cache
Cache-Control: no-cache
Expires: Wed, 31 Dec 1969 16:00:00 PST
X-Powered-By: Servlet 2.5; JBoss-5.0/JBossWeb-2.1
Content-Type: application/octet-stream
Content-Length: 1675
Date: Mon, 24 Nov 2014 23:15:35 GMT
```

Response Body

```
PK
??uE META-INF/??PK
??uE?!
K-???#R0?3??r?Cq,HL?HU?%-?x???RKRSt?*A???
...
```

6.2.4.4. Upload New Resource File with POST

Definition

POST /model

Upload a new resource file. The file name in 'file' body parameter will be used to identify this resource.

Request Parameters

Content-Type (string): required header parameter with two accepted values: application/octet-stream or multipart/form-data

file (string): required query parameter for PMML file name, if Content-Type is application/octet-stream, or a body parameter in multipart/form-data content encoding

Content-Type (string): required body parameter for resource a file name, and its content

Returns

Returns [ResourceInfo](#) object, 201 HTTP response status code, and response header entry [Location](#) with URI of created resource if upload was successful, an [Errors](#) object otherwise.

Example 6.32. ZEMENTIS REST Upload New Resource File with POST

Request

```
curl -u adapa:adapa -k https://localhost/adapars/resource?file=ECommerceFraud_NN.xls -X POST \
-T ECommerceFraud_NN.xls -H "Content-Type:application/octet-stream"
curl -u adapa:adapa -k https://localhost/adapars/resource -X POST -F file=@ECommerceFraud_NN.xls
```

Request Header

```
POST /adapars/resource HTTP/1.1
Authorization: Basic YWRhcGE6YWRhcGE=
```

```
User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r zlib/1.2.5
Host: localhost
Accept: */*
Content-Length: 30933
Expect: 100-continue
Content-Type: multipart/form-data; boundary=-----d9c9597fd160
```

Response Header

```
HTTP/1.1 201 Created
Server: Apache-Coyote/1.1
X-Powered-By: Servlet 2.5; JBoss-5.0/JBossWeb-2.1
Location: http://localhost:8080/adapars/resource/ECommerceFraud_NN.xls
Content-Type: application/json
Content-Length: 156
Date: Wed, 26 Mar 2014 19:45:18 GMT
```

Response Body

```
{
  "fileName" : "ECommerceFraud_NN.xls",
  "resourceType" : "Lookup Tables",
  "resourceIdentifier" : "Table Name",
  "resourceNames" : [ "StatePoints" ]
}
```

6.2.4.5. Upload New Resource File with PUT

Definition

PUT /model

Upload a new resource file. The file name in 'file' query parameter will be used to identify this resource.

Request Parameters

file (string): required query parameter for resource file name

Returns

Returns a [ResourceInfo](#) object, 201 HTTP response status code, and a response header entry `Location` with URI of the created resource if the upload was successful, an [Errors](#) object otherwise.

Example 6.33. ZEMENTIS REST Upload New Resource File with PUT

Request

```
curl -u adapa:adapa -k https://localhost/adapars/resource?file=ECommerceFraud_NN.xls -X PUT -T ECommerceFraud_NN.xls
```

Request Header

```
PUT /adapars/resource?file=ECommerceFraud_NN.xls HTTP/1.1
Authorization: Basic YWRhcGE6YWRhcGE=
User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r zlib/1.2.5
Host: localhost
Accept: */*
```

```
Content-Length: 30720
Expect: 100-continue
```

Response Header

```
HTTP/1.1 201 Created
Server: Apache-Coyote/1.1
Pragma: No-cache
Cache-Control: no-cache
X-Powered-By: Servlet 2.5; JBoss-5.0/JBossWeb-2.1
Location: http://localhost:8080/adapars/resource/ECommerceFraud_NN.xls
Content-Type: application/json
Content-Length: 156
Date: Mon, 24 Nov 2014 23:37:26 GMT
```

Response Body

```
{
  "fileName" : "ECommerceFraud_NN.xls",
  "resourceType" : "Lookup Tables",
  "resourceIdentifier" : "Table Name",
  "resourceNames" : [ "StatePoints" ]
}
```

6.2.4.6. Remove Resource File

Definition **DELETE /resource/{file_name}**

Remove the specified resource file and list all remaining resources.

Request Parameters **file_name (string):** required path variable for existing resource file name

Returns Returns a [Resources](#) object with a list of all remaining resource files if successful, an [Errors](#) object otherwise.

Example 6.34. ZEMENTIS REST Remove Resource File

Request

```
curl -u adapa:adapa -k https://localhost/adapars/resource/ECommerceFraud_NN.jar -X DELETE
```

Request Header

```
DELETE /adapars/resource/ECommerceFraud_NN.xls HTTP/1.1
Authorization: Basic YWRhcGE6YWRhcGE=
User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r zlib/1.2.5
Host: localhost
Accept: */*
```

Response Header

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Pragma: No-cache
Cache-Control: no-cache
```



```
Expires: Wed, 31 Dec 1969 16:00:00 PST
X-Powered-By: Servlet 2.5; JBoss-5.0/JBossWeb-2.1
Content-Type: application/json
Content-Length: 195
Date: Mon, 24 Nov 2014 23:50:13 GMT
```

Response Body

```
{
  "resources" : [ {
    "fileName" : "ECommerceFraud_NN.jar",
    "resourceType" : "Custom Functions",
    "resourceIdentifier" : "Function Namespace",
    "resourceNames" : [ "fraud" ]
  } ]
}
```

6.2.4.7. Remove All Resource Files

Definition **DELETE /resources**

Remove all available resources and list the remaining resources.

Request Parameters None

Returns Returns a [Resources](#) object with an empty `resources` array if successful, an [Errors](#) object otherwise.

Example 6.35. ZEMENTIS REST Remove All Resource Files

Request

```
curl -u adapa:adapa -k https://localhost/adapars/resources -X DELETE
```

Request Header

```
DELETE /adapars/resources HTTP/1.1
Authorization: Basic YWRhcGE6YWRhcGE=
User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r zlib/1.2.5
Host: localhost
Accept: */*
```

Response Header

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Pragma: No-cache
Cache-Control: no-cache
Expires: Wed, 31 Dec 1969 16:00:00 PST
X-Powered-By: Servlet 2.5; JBoss-5.0/JBossWeb-2.1
Content-Type: application/json
Content-Length: 23
Date: Mon, 24 Nov 2014 23:57:57 GMT
```

Response Body

```
{
  "resources" : [ ]
}
```

