



## Tutorial XELOPES

Version 4.0



The prudsys AG can be owner of patent rights, brands, copyrights, or other rights on intellectual property regarding the contents of this documentation. The provision of this documentation does not automatically mean that you have rights of disposal for these patents, brands, copyrights or other intellectual property.

Without the prior written permission by the prudsys AG, no part of this documentation may be duplicated or transferred for any purposes.

prudsys is a registered trademark of the prudsys AG. All other trademarks or registered trademarks stated in this manual are property of the respective owners.

Author: *Xeli*

© 2009 prudsys AG. All rights reserved.

# CONTENTS

- 1 Quick Intro ..... 4
  - 1.1 Association Rules ..... 4
  - 1.2 Classification..... 6
- 2 Overview ..... 9
  - 2.1 Installation..... 9
  - 2.2 Distribution Structure..... 9
  - 2.3 CWM – The Fundament of XELOPES ..... 9
  - 2.4 PMML – Data Mining Exchange Format..... 10
- 3 Elements ..... 11
  - 3.1 Data..... 11
    - 3.1.1 The Basis – MiningDataSpecification and MiningAttribute’s ..... 11
    - 3.1.2 The Coordinates – MiningVector ..... 13
    - 3.1.3 The Data Matrix – MiningInputStream ..... 13
  - 3.2 Transformations..... 15
    - 3.2.1 Transformations of Mining Vectors ..... 15
    - 3.2.2 Transformations of Mining Input Streams ..... 16
    - 3.2.3 Basis Transformations..... 18
  - 3.3 Models ..... 18
  - 3.4 Algorithms..... 19
    - 3.4.1 Mining Settings..... 20
    - 3.4.2 Mining Algorithm Specification..... 21
    - 3.4.3 Algorithm Types ..... 21
- 4 Simplified Interface ..... 22
- A Appendix ..... 25
  - A.1 Association Rules Example..... 25
  - A.2 Classification Example ..... 27

# 1 Quick Intro

prudsys XELOPES is a Business Intelligence (BI) library, developed in cooperation with Russian ZSoft Ltd, with main focus on Data Mining.

This tutorial gives a short introduction to XELOPES and starts with two practical examples, the first one utilizes an association rules model and the second one a decision tree model. Both examples use the simplified interface which is described in Chapter 4. The corresponding code that uses the standard interface, which is introduced in Chapter 3, is shown in Appendix A.

## 1.1 Association Rules

A small retailer in food industry collects the data of all purchases of the customers. Consider the case that at the beginning the data consists of the following 4 orders:

1. Cracker, Water, Beer
2. Coke, Water, Nut
3. Nut, Coke, Cracker, Coke, Water
4. Coke, Nut, Coke

This data set is available as the transactional CSV file *data/csv/transact.csv*:

transactId	itemId	itemName	itemPrice
0_	1	Cracker	12.0
0_	3	Water	4.0
0_	4_	Beer	14.0
1	2	Coke	10.0
1	3	Water	4.0
1	5	Nut	15.0
2	5	Nut	15.0
2	2	Coke	10.0
2	1	Cracker	12.0
2	2	Coke	10.0
2	3	Water	4.0
3	2	Coke	10.0
3	5	Nut	15.0
3	2	Coke	10.0

The retailer wants to know which products are typically bought together and which product combinations typically lead to other product combinations in order to extract some cross-selling potential in the purchases.

To build an association rules model for solving this problem, the example *com.prudsys.pdm.Examples.AssociationRulesTutorialServiceAPIBuild* can be used, which is located in the *src/* directory. The Java code looks like:

```
// Create Local XELOPES Service Implementation object:
LocalXelopesServiceImpl lxsi = new LocalXelopesServiceImpl();

// Create and open input stream
MiningInputStream dataSource =
    new MiningCsvStream("data/csv/transact.csv");
dataSource.open();

// Want all algorithm parameters of algorithm 'FPGrowth':
String selectAlgo = "FPGrowth";
```

```

ServiceAlgorithmParameter[] algPar =
    lxsi.getAlgorithmParameters(selectAlgo);

// Set minimum support to 50%, minimum confidence to 30%
LookupService.setSAPValue(algPar, "minimumSupport", "0.5");
LookupService.setSAPValue(algPar, "minimumConfidence", "0.3");

// Set name of item id and transaction id column
LookupService.setSAPValue(algPar, "itemIdName", "itemId");
LookupService.setSAPValue(algPar, "transactionIdName", "transactId");

// Run association rules algorithm:
System.out.println("-->Build model by service: ");
MiningModel associationRulesModel =
    lxsi.buildModelService(selectAlgo, algPar, dataSource);

// Display rules
showRules((AssociationRulesMiningModel) associationRulesModel);

// Write result into PMML file:
FileWriter writer =
    new FileWriter("data/pmml/AssociationRulesModelT.xml");
associationRulesModel.writePmml(writer);

```

The association rules model for this example contains 14 rules as shown in Table 1.1.

Premise	Conclusion	Support	Confidence
Water(id=3)	Coke(id=2)	50.00%	66.67%
Coke(id=2)	Water(id=3)	50.00%	66.67%
Coke(id=2)	Nut(id=5)	75.00%	100.00%
Nut(id=5)	Coke(id=2)	75.00%	100.00%
Water(id=3)	Coke(id=2), Nut(id=5)	50.00%	66.67%
Water(id=3), Coke(id=2)	Nut(id=5)	50.00%	100.00%
Water(id=3), Nut(id=5)	Coke(id=2)	50.00%	100.00%
Coke(id=2)	Water(id=3), Nut(id=5)	50.00%	66.67%
Coke(id=2), Nut(id=5)	Water(id=3)	50.00%	66.67%
Nut(id=5)	Water(id=3), Coke(id=2)	50.00%	66.67%
Water(id=3)	Nut(id=5)	50.00%	66.67%
Nut(id=5)	Water(id=3)	50.00%	66.67%
Cracker(id=1)	Water(id=3)	50.00%	100.00%
Water(id=3)	Cracker(id=1)	50.00%	66.67%

Table 1.1: Association rules for minimum support 50% and minimum confidence 30%.

Next, the retailer wants to know which product is most likely ordered together with the item “Nut” (id=5). This is demonstrated in the example *com.prudsys.pdm.Examples.AssociationRulesTutorialApply*, which is located in the *src/* directory. The corresponding Java code looks like:

```

// Read association rules from PMML file:
AssociationRulesMiningModel model = new AssociationRulesMiningModel();
FileReader reader =
    new FileReader("data/pmml/AssociationRulesModelT.xml");
model.readPmml(reader);
CategoricalAttribute categoryItemId = getItemIdAttribute(model);
System.out.println("-----> PMML model read successfully");

// Build recommendations:
model.buildRecommendations();

```

```
ItemSet is = new ItemSet();
is.addItem( (int) categoryId.getKey( new Category("5") ));
// Get recommendation with highest support
RuleSet rs = (RuleSet) model.applyModel(is);
if (rs != null) {
    showItems(categoryItemId, rs.getPremise());
    System.out.print(" => ");
    // Contains recommended item
    showItems(categoryItemId, rs.getConclusion());
}
else
    System.out.println("No recommendations found.");
```

## 1.2 Classification

In this example, a telecommunication provider offers two-years contracts to its customers. The contract will automatically be renewed for one year, if the customer does not cancel three months before its expiration.

The provider wants to send an offer for a new customer-friendly tariff to those customers, who are likely to cancel the two years contract, in order to keep them. It is assumed that offering a potential canceller a new tariff results in higher profits than losing the customer.

Furthermore, the provider does not want to call attention to the possibility of tariff change or cancellation for those customers who are not likely to cancel, since this could cause loss of sales and profits to the provider..

Thus, the aim is to determine which customers will likely cancel the two-years contract. Since the provider usually has a large amount of customer information, the information of the previous two-years contracts can be used to learn the profile of a canceller and then apply this profile to current customers.

In this example, only the following 6 attributes are considered:

1. gender,
2. birthday (consequently also the age),
3. current tariff,
4. units consumed in each tariff,
5. address,
6. did (not) cancel after two years.

For simplicity, a small learning data set is used:

SEX	AGE	CURRENT_TARIFF	CONSUMED_UNITS	CANCELLER
f	23	normal	345	no
m	18	power	9455	no
m	36	power	456	no
m	34	normal	3854	yes
f	52	economy	2445	no
f	19	economy	14326	no
f	45	normal	347	no
m	42	economy	5698	yes
m	21	power	267	no
m	48	normal	4711	yes

The provider wants to know whether the following customers will likely cancel:

SEX	AGE	CURRENT_TARIFF	CONSUMED_UNITS
f	34	economy	155
m	56	power	2398
m	18	power	253
f	39	normal	7544

To build a decision tree model for this problem, the example `com.prudsys.pdm.Examples.DecisionTreeTutorialServiceAPIBuild` can be used, which is located in the `src/` directory. The Java code looks like:

```
// Create Local XELOPES Service Implementation object:
LocalXelopesServiceImpl lxsi = new LocalXelopesServiceImpl();

// Create and open input stream
MiningInputStream dataSource =
    new MiningCsvStream("data/csv/cancellingTrain.csv");
dataSource.open();

// Algorithm parameters of algorithm 'Decision Tree (General)':
String selectAlgo = "Decision Tree (General)";
ServiceAlgorithmParameter[] algPar =
    lxsi.getAlgorithmParameters(selectAlgo);

// Set target attribute and algorithm parameters
LookupService.setSAPValue(algPar, "minNodeSize", "0.3");
LookupService.setSAPValue(algPar, "maxDepth", "100");
LookupService.setSAPValue(algPar, "targetName", "CANCELLER");

// Run decision tree algorithm and obtain model
MiningModel decisionTreeModel =
    lxsi.buildModelService(selectAlgo, algPar, dataSource);

// Write result into PMML file
FileWriter writer = new FileWriter("data/pmml/DecisionTreeModelT.xml");
decisionTreeModel.writePmml(writer);
```

Figure 1.1 shows a visualization of the resulting decision tree model.

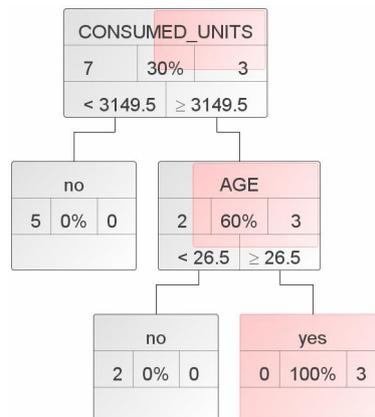


Figure 1.1: Decision tree with 3 leaf nodes, the first one represents 5 non-cancellers, the second leaf node 2 non-cancellers and the third one represents 3 cancellers.

Now, the 4 customers shall be classified. For this, see example `com.prudsys.pdm.Examples.DecisionTreeTutorialApply`.

```
// Read decision tree model from PMML file:
MiningModel model = new DecisionTreeMiningModel();
FileReader reader = new FileReader("data/pmml/DecisionTreeModelT.xml");
model.readPmml(reader);

CategoricalAttribute modelTargetAttribute = (CategoricalAttribute)
    ((SupervisedMiningSettings) model.getMiningSettings()).getTarget();

// Open data source and get metadata:
MiningInputStream inputData0 =
    new MiningCsvStream("data/csv/cancellingTest.csv");

// Transform input data into model format:
MiningInputStream inputData =
    model.transformIntoModelFormat(inputData0);
MiningDataSpecification inputMetaData = inputData.getMetaData();

// Show classification results:
while (inputData.next()) {
    // Make prediction:
    MiningVector vector = inputData.read();
    double predicted = model.applyModelFunction(vector);
    Category predTarCat = modelTargetAttribute.getCategory(predicted);
    System.out.println("Prediction: " + predTarCat);
}
```

The output will be:

```
Prediction: no
Prediction: no
Prediction: no
Prediction: yes
```

## 2 Overview

### 2.1 Installation

XELOPES requires Sun Java SE Development Kit (JDK) 1.4 or higher (<http://java.sun.com>). Before proceeding with the installation, please make sure that such a JDK is installed on your system, Java SE Runtime Environment (JRE) is not sufficient. Furthermore, Apache Ant 1.6 or later (<http://ant.apache.org>) is required.

Next, unpack the XELOPES archive to any desired directory. This directory will be referred to as the XELOPES directory. Then, change to the XELOPES directory.

For Unix or Linux, it is assumed that the environmental variable `JAVA_HOME` points to the JDK installation directory and Apache Ant is installed in `/opt/ant`. Otherwise, please set `JAVA_HOME` and edit the build script `build.sh` accordingly. Then, run `./build.sh` and check that the build completes successfully. Afterwards, the test script `./run.sh` should be called. The number of failed test should be zero.

For Windows, modify the file `build.bat`, such that the variable `JAVA_HOME` points to the JDK directory and `ANT_HOME` to the Apache Ant directory. Then run the `build.bat` script. After the successful build, the test script `run.bat` should be called. After completion, the test statistics should show that all tests have passed.

### 2.2 Distribution Structure

The XELOPES distribution contains the following files and directories:

Directory / File	Description
<code>config/</code>	Configuration files (PMML, algorithms)
<code>classes/</code>	XELOPES class files
<code>data/</code>	Data sets and PMML models for testing
<code>doc/</code>	Documentation in PDF and Javadoc format
<code>lib/</code>	Libraries required for XELOPES
<code>src/</code>	XELOPES Java source files
<code>XELICENSE</code>	XELOPES license file
<code>build.bat</code>	Build script for Windows, runs Apache Ant
<code>build.sh</code>	Build script for Unix/Linux, runs Apache Ant
<code>Build.xml</code>	Apache Ant build configuration file
<code>readme.txt</code>	XELOPES readme file
<code>run.bat</code>	Runs XELOPES tests (for Windows)
<code>run.sh</code>	Runs XELOPES tests (for Unix/Linux)
<code>Xelopes.jar</code>	Jar file of XELOPES library, contains the class files from <code>classes/</code>

Table 2.1: XELOPES distribution structure.

### 2.3 CWM – The Fundament of XELOPES

XELOPES is based on the Common Warehouse Metamodel (CWM) standard of the OMG. For further information see “XELOPES Library Documentation” (`XelopesManual.pdf` in the `doc/` directory) and the CWM page of the OMG (<http://www.omg.org/cwm/>).

## **2.4 PMML – Data Mining Exchange Format**

PMML is a standard for vendor-independent XML exchange of Data Mining models. PMML is supported by the core of XELOPES and all models of XELOPES can be exported into and imported from PMML. For further information see “XELOPES Library Documentation” (XelopesManual.pdf in the doc/ directory) and the homepage of the Data Mining Group (DMG, <http://www.dmg.org>).

## 3 Elements

### 3.1 Data

#### 3.1.1 The Basis – MiningDataSpecification and MiningAttribute's

The class *MiningDataSpecification* represents the basis of attribute space. Thus, this is the most important class of XELOPES. Often this class is simply referred to as *meta data* of the mining what. It corresponds to PMMLs *DataDictionary* element.

The basis vectors of *MiningDataSpecification* are the *MiningAttribute*-s representing the attributes.

*MiningAttribute* corresponds to PMMLs *DataField* element. There are two basic types of mining attributes extending the abstract class *MiningAttribute*: *NumericAttribute* for numeric attributes like age, income, time and *CategoricalAttribute* for categorical attributes like names, IDs, types. The elements of a numeric attribute are real numbers. The elements of a categorical attribute are the categories which are represented by the *Category* class. *Category* corresponds to PMMLs *Value* element.

#### Example

```
// Create category 'knife':
Category catKnife = new Category("knife");
```

The categories of a categorical attribute are stored in an array of *CategoricalAttribute*. Unlike as for the straightforward *NumericAttribute*, the mathematical nature of the *CategoricalAttribute* is rather ambivalent: It can be interpreted as one or a set of multiple numeric attributes. In the last case (e.g. binning), *CategoricalAttribute* represents a basis itself, with the *Category*-s as basis vectors. Thus, the set of categories is also called the *basis* or the *metadata* of the categorical attribute. Each category of a categorical attribute can be mapped to a unique real number (usually an integer) which is called the *key* of this category. This establishes a mutually unique mapping between the categories and a set of real numbers and allows reducing the handling of categories to that of real values.

#### Example

```
// Create categorical attribute 'cutlery':
CategoricalAttribute cutlery = new CategoricalAttribute("cutlery");

// Add categories:
Category catFork = new Category("fork"); // new fork category
cutlery.addCategory( catKnife );        // from previous example
cutlery.addCategory( catFork );
cutlery.addCategory( new Category("spoon") );

// Show key-value relationship:
double keyKnife = cutlery.getKey( catKnife );
Category catKnife2 = cutlery.getCategory( keyKnife );
// catKnife == catKnife2
```

XELOPES actively supports three storage types of categorical attributes:

- *Static category set* (default): All categories are apriori known. Examples are sex (female, male) or colours (red, green, blue).

- *Dynamic category set*: During the data processing new categories may appear and are added to the basis (option *unboundedCategories*). Examples are item or category names.
- *Dynamic category set with one category*: During the data processing only the current category is stored (option *unstoredCategories*, implies *unboundedCategories*). Examples are transaction IDs and customer names.

Of course, the proper use of the storage types for the given examples can also differ. We emphasize that the support of the *unboundedCategories* and *unstoredCategories* type is really complicated – especially in the field of basis transformations – but very valuable since it allows to handle large and live data sources.

Categorical attributes with a defined order of categories are modelled by the class *OrdinalAttribute* which extends *CategoricalAttribute*.

Further, the categories of a categorical attribute can be organized into a hierarchy (also referred to as *taxonomy*). This is e.g. required for many basket analysis algorithms or for OLAP drill-down functionality. Hierarchies of categories are modelled by the class *CategoryHierarchy* and can be assigned to a categorical attribute via its *setTaxonomy* method. *CategoryHierarchy* uses the method *addRelationship* to add a new edge to the hierarchy graph and many methods allow running calculations on the graph. *CategoryHierarchy* corresponds to PMMLs *Taxonomy* element.

### Example

```
// Create category hierarchy:
CategoryHierarchy cah = new CategoryHierarchy();

// Parent category for sharp cutlery:
Category catSharp = new Category("sharp");

// Relations:
cah.addRelationship(catSharp, catKnife); // knife is sharp
cah.addRelationship(catSharp, catFork); // fork is sharp

// Assign hierarchy to cutlery:
cutlery.setTaxonomy(cah);
```

We conclude with an example of *MiningDataSpecification*.

### Example

```
// Create object of metadata 'meal':
MiningDataSpecification meal = new MiningDataSpecification("meal");

// Create numeric attribute 'calories' and add to metadata:
NumericAttribute calories = new NumericAttribute("calories");
meal.addMiningAttribute( calories );

// Create numeric attribute 'numberOfGuests' and add to metadata:
NumericAttribute numberOfGuests = new NumericAttribute();
numberOfGuests.setName( "number of guests" );
meal.addMiningAttribute( numberOfGuests );

// Add previous categorical attribute 'cutlery' to metadata:
meal.addMiningAttribute( cutlery );
```

### 3.1.2 The Coordinates – MiningVector

After we have modelled the basis of the attribute space by *MiningDataSpecification*, we will now model the coordinates of a vector of the space in the basis *A*. This is done through the class *MiningVector*.

*MiningVector* contains a reference *metaData* (of the class *MiningDataSpecification*) to the basis *A*, and an array of real values which stores the coordinates of the vector.

#### Example

Example of a mining vector for the *meal* basis of the previous section.

```
// Create and fill value vector:
double[] mealValues = new double[3];
mealValues[0] = 33000; // calory number
mealValues[1] = 5;    // 5 guests
mealValues[2] = cutlery.getKey( new Category("spoon") ); // spoon

// Create mining vector object with values:
MiningVector mealVector = new MiningVector( mealValues );

// Add 'meal' metadata to mining vector:
mealVector.setMetaData( meal );

// Show (double) values of mining vector:
for (int i = 0; i < mealVector.getValues().length; i++)
    System.out.println("value["+i+"] = " + mealVector.getValue(i));
```

For sparse vectors, i.e. vectors which mainly contain zero coordinate values, the class *MiningSparseVector* could be used which extends *MiningVector*. It stores sparse vectors more efficiently by means of an additional array of indexes of the non-zero coordinate values. For binary sparse vectors, i.e. sparse vectors where the non-zero values are always one, the class *MiningBinarySparseVector* should be utilized which in turn extends *MiningSparseVector*.

### 3.1.3 The Data Matrix – MiningInputStream

In the previous section we have defined the class *MiningVector* that models a data vector. In order to model a whole data matrix we use the abstract *MiningInputStream* class. *MiningInputStream* is a virtual collection of mining vectors. Like each of its mining vectors, *MiningInputStream* contains a reference *metaData* to the basis of the attribute space.

*MiningInputStream* contains a graded spectrum of data access methods depending on its implementation. In the simplest case, the data matrix can be traversed only once using a cursor-based approach using the method *next*. If the *reset* method is supported, the cursor can be set at the initial position. This access type is often supported by files and databases. In a more comfortable case, the cursor can be moved arbitrary using the *move* method (e.g., for databases supporting JDBC 2.0). Even more comfortable is the direct access to the data array of the data matrix, if the matrix fits into memory (e.g. class *MiningStoredData*).

The *read* method returns the mining vector at the current cursor position. Each full implementation of *MiningInputStream* must at least support the *next* and *read* methods. In addition, *MiningInputStream* contains methods to write data to the data source. Each stream that supports these methods is called *updateable*. Each mining input stream is reflective: The method *getSupportedStream* returns all data access (and update) methods supported by the current implementation.

The mining input stream concept is a direct consequence of the fact that each Data Mining algorithm requires a *data matrix* as input.

The physical model describes the physical data source that is used for mining, like a text file or a database. For the Data Mining process, the physical model must be mapped to the logical one.

Resource Stream	Superclass	Description
<i>MiningArrayStream</i>	<i>MiningInputStream</i>	Access to data stored in array
<i>MiningStoredData</i>	<i>MiningInputStream</i>	Access to data stored in vector
<i>MiningIteratorStream</i>	<i>MiningInputStream</i>	Access to Iterator objects
<i>MiningSqlStream</i>	<i>MiningInputStream</i>	Access to data stored in database
<i>MiningFileStream</i>	<i>MiningInputStream</i>	Access to data stored in a file
<i>MiningCsvStream</i>	<i>MiningFileStream</i>	Access to data in CSV file
<i>MiningArffStream</i>	<i>MiningFileStream</i>	Access to data in ARFF file
<i>MiningExcelStream</i>	<i>MiningFileStream</i>	Access to data in Excel file
<i>LogFileStream</i>	<i>MiningFileStream</i>	Access to data in web server log file
<i>MiningFilterStream</i>	<i>MiningInputStream</i>	Access to transformed stream
<i>MultidimensionalStream</i>	<i>MiningInputStream</i>	Access to multidimensional stream
<i>MultidimensionalSqlStream</i>	<i>MultidimensionalStream</i>	Access to multidimens. SQL stream

Table 3.1: Important resource streams.

In XELOPES this mapping is done by subclassing: Different types of physical data sources can be accessed through different mining input stream classes that extend *MiningInputStream*. The basic resource stream classes of XELOPES are listed in Table 3.1. Often it is useful to write own resource classes which extend *MiningInputStream* or one of its subclasses.

Notice that the last three streams are composed streams which take an arbitrary mining input stream as input and apply a transformation and multidimensional selection / ordering to the stream, respectively.

### Example

```
// Open CSV file 'iris.dat':
MiningCsvStream inputStream = new MiningCsvStream("csv/iris.dat");
inputStream.open();

// Get metadata of Iris:
MiningDataSpecification metaData = inputStream.getMetaData();

// Read all data vectors of Iris:
while( inputStream.next() ) {
    MiningVector mv = inputStream.read();
    // ... //
}

inputStream.close();
```

We summarize. The mining input stream concept makes XELOPES independent of physical data sources. Each mining algorithm takes a *MiningInputStream* as input, probably requests the supported data access methods, and accesses the stream data through the (supported) standard access methods. The physical stream model, i.e. the subclass of *MiningInputStream* passed to the algorithm, is in general not required to be known by a XELOPES mining algorithm.

## 3.2 Transformations

In the previous sections, we have introduced the basis and vector classes of the attribute space. In this section the transformations of bases and vectors will be discussed. Transformations are a central part of XELOPES.

There are two basic types of transformations supported by XELOPES:

- transformations of mining vectors, and
- transformations of mining input streams.

### 3.2.1 Transformations of Mining Vectors

The first type are transformations of mining vectors which implement the *MiningTransformer* interface. It has a simple structure which clearly reflects the XELOPES approach to basis transformations:

```
public interface MiningTransformer
{
    public MiningDataSpecification transform( MiningDataSpecification
        metaData ) throws MiningException;

    public MiningVector transform( MiningVector vector )
        throws MiningException;
}
```

The first *transform* method transforms basis *A* into a basis *B*. The second *transform* method transforms the coordinates of a vector in basis *A* into the coordinates of the transformed vector in basis *B*.

We mention three important special cases of vector transformations: If *A* and *B* are bases of the same space and the vector is not transformed (but just its coordinates), this is called a *pure basis transformation*. Basis transformations are discussed in Section 3.2.3. If the bases are equal, i.e.  $A = B$  (first *transform* method is identity), and the vector is transformed, this is called a *pure vector transformation*. Further, basis *B* could be the basis of an other space, then we have a *space transformation*. Often these types of transformations are mixed.

Back to *MiningTransformer*. Its main advantage is the clear separation of basis and coordinate transformations. This has large practical consequences.

#### ***MiningFilterStream***

An example of the advantages of separating the basis from the coordinate transformation is the *MiningFilterStream* which applies transformations dynamically to a mining input stream. *MiningFilterStream* is itself a special type of a mining input stream (Section 3.1.3). Its constructor takes an arbitrary mining input stream object *miningInputStream* and a mining transformer object *miningTransformer* as arguments. Then, in the *getMetaData* and *read* methods of *MiningFilterStream* the transformations of *miningTransformer* are applied to the metadata and mining vectors of *miningInputStream*. The work of *MiningFilterStream* is illustrated in Figure 3.1.

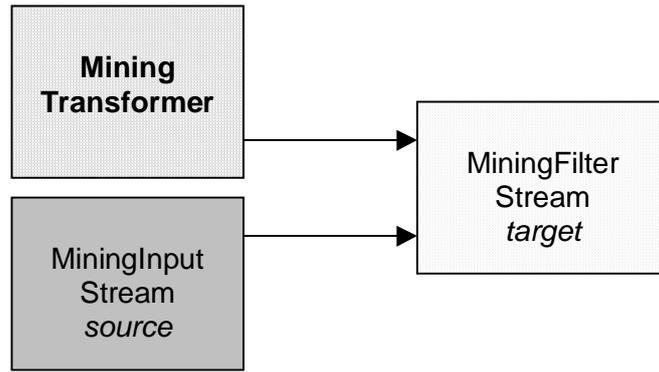


Figure 3.1: Scheme of dynamic transformations (*MiningFilterStream*).

The other stream methods of *MiningFilterStream* are simply passed to *miningInputStream*. *MiningFilterStream* is universal and easy to use. It can be applied to streams of almost unbounded size. The disadvantage is the lower access speed since each call of *read* runs a transformation. *MiningFilterStream* is often used in XELOPES.

We have seen that *MiningTransformer* transforms one mining vector into another. Hence this transformation type transforms attribute values and represents a *vector transformation* in the attribute space.

### 3.2.2 Transformations of Mining Input Streams

The transformations of mining input streams are the more general ones and implement the *MiningStreamTransformer* interface. The *MiningStreamTransformer* interface consists of one method *transform* which takes a *source* mining input stream as input and a *target* mining input stream as output of the transformation. The target mining input stream must be updateable because its content is deleted before the transformation starts and then the transformed source stream is written to this stream. Obviously, this type of transformation covers almost any type of transformations of mining input streams. We call this type of transformations *stream transformations* (Figure 3.2).

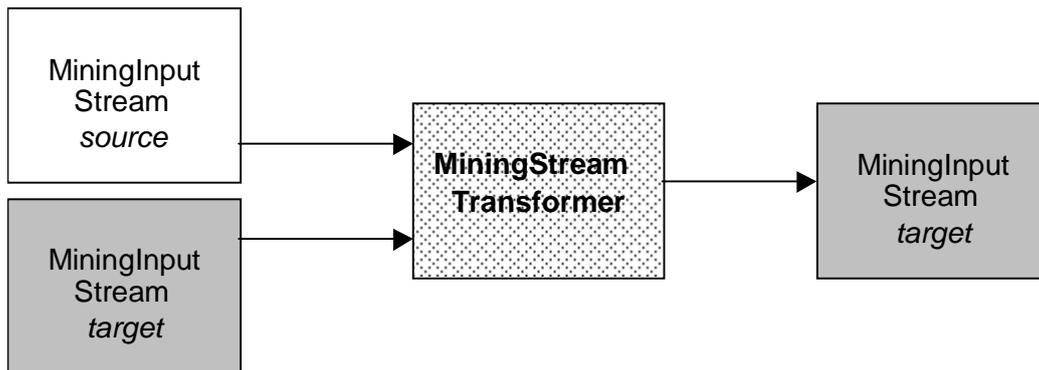


Figure 3.2: Scheme of static transformations (*MiningStreamTransformer*).

Regarding mining input streams, both types of transformations result in the following transformation types:

- *Static transformations*: Here a stream transformation object (that hence implements the *MiningStreamTransformer* interface) converts the source stream into the target stream only *once* and then the transformed data is available in the target mining input stream.

- *Dynamic transformations*: Here a vector transformation object (that hence implements the *MiningTransformer* interface) is used in *MiningFilterStream* to run the transformation any time when the *read* method is called (see previous section).

Both types of transformations have advantages and disadvantages: In case of static transformations, the transformation is done only once and then the target stream contains the complete transformed data. The disadvantage of this approach is that we need two streams to be supported (and usually about twice of memory amount). This means that static transformations are optimal in speed but non-optimal in memory consumption.

For dynamic transformations via mining filter streams we do not need additional memory but any time when we access the data from the stream using the *read* method, the transformation of the current vector is carried out again. Hence, dynamic transformations are non-optimal in time but optimal in memory. The static and dynamic transformations represent the classic dilemma that increased speed requires increased memory and vice versa.

There are many vector transformations implemented into XELOPES based on an extensive CWM framework. For general stream transformations, if they are not based on vector transformations, there is no further framework provided in XELOPES. Implementations of general stream transformations are contained in the *Special* subpackage of *Transformation*.

Often a transformation should be applied to a group of attributes. Examples are normalizations of all numeric attributes or binning of all categorical attributes of the metadata. For this purpose, there exist transformation classes extending *VectorTransformationStream* which are helpful because they allow writing the transformations in a compact form. *VectorTransformationStream* has a constructor with the mining input stream as argument and a method *createTransformedStream* creating the transformed mining input stream (using *MiningFilterStream*).

### **Example**

The class *LinearNormalStream* extends *VectorTransformationStream* for linear normalizations of all numeric attributes of a stream. It automatically calculates all required statistics and delivers the complete transformed stream. In the same way, *BinningStream* applies binning to all categorical attributes of a stream. The example shows how to transform all attributes of a mining input stream into numeric attributes of the interval [0,1].

```
// (0,1)-Normalization of all numeric attributes:
LinearNormalStream lns = new LinearNormalStream( inputStream );
lns.setLowerBound(0);
lns.setUpperBound(1);

// Create normalized stream:
MiningInputStream normStream = lns.createTransformedStream();

// Binning of all categorical attributes:
BinningStream bns = new BinningStream( normStream );

// Create binned and normalized stream:
MiningInputStream transStream = bns.createTransformedStream();
```

As we see in the example, vector stream transformations can easily be concatenated.

There exist numerous classes extending *VectorTransformationStream*; the most important ones are listed in Table 3.2. Although these transformations are easy to use, they offer a lot of options and – in most cases – multiple algorithms.

Vector Transformation Stream	Description
<i>BinningStream</i>	Binning of categorical attributes
<i>DiscretizationStream</i>	Discretization of numeric attributes
<i>LinearNormalStream</i>	Linear normalization of numeric attributes
<i>NumerizationStream</i>	Numerization of categorical attributes by taking the keys
<i>NumTargetStream</i>	Numerization of categorical attributes via target attribute
<i>ReplaceMissingValueStream</i>	Replacement of missing values
<i>TreatOutlierValueStream</i>	Replacement of outliers
<i>ZetNormalStream</i>	Zet normalization of numeric attributes

Table 3.2: Important vector transformation streams.

### 3.2.3 Basis Transformations

Basis transformations are very important but also somewhat abstract. Luckily for most applications the XELOPES user does not have to care about basis transformations because they are automatically executed internally. So we will be very brief.

In XELOPES, basis transformations are implemented in the spirit of *tensor algebra*. They are provided by the class *MetaDataOperations* which is a singleton class owned by the metadata class *MiningDataSpecification*. Thus, each *MiningDataSpecification* object owns an object *MetaDataOperations* to transform another *MiningDataSpecification* and appendant mining vectors into its own basis.

## 3.3 Models

The abstract class *MiningModel* represents the Data Mining model which is mainly the mining function. The central method of *MiningModel* is *applyModelFunction* which takes a mining vector as argument and returns the function value. Thus, *applyModelFunction* is used to apply the mining model to data. There exists a second application method *applyModel* which is more general and returns objects (e.g. a mining vector for SV clustering, an item set for an association rules model or a node for a decision tree model).

*MiningModel* implements the *Pmmlable* interface for serialization into PMML format since all models of XELOPES can be written to and read from PMML files.

Each class representing a type of Data Mining models extends *MiningModel*. For instance, *AssociationRulesMiningModel* extends *MiningModel* for association rules and contains the implementations (*applyModel* for rules and PMML export/import of rules, etc.). For a special association rule model *AssociationRuleMiningModel* may be further subclassed. For instance, for flat association rules it may be useful to introduce a new class *FlatRulesMiningModel* which extends *AssociationRulesMiningModel*. XELOPES already contains a wide hierarchy of all basic classes of Data Mining models including the main implementations like the *apply* methods and PMML serialization. If the user requires a special model, she can extend one of the existing models.

Because of the wide variety of Data Mining models and algorithms a two-level system of their classification is used.

The *function* level defines the basic types of mining models. It consists of the following types predefined in CWM:

- *StatisticalAnalysis*,
- *FeatureSelection*,

- *AssociationRules*,
- *Classification*,
- *Clustering*,
- *Regression*,

and the following functions added for XELOPES:

- *Sequential* (sequence analysis),
- *CustomerSequential* (sequential basket analysis),
- *TimeSeriesPrediction* (time series predication),
- *PriceOptimization* (price optimization).

The function type of *MiningModel* is stored in the variable *function*. The mining models of XELOPES are organized in packages whose names correspond to the functions. For instance, all classification models are contained in the package *Classification* which contains further subpackages for special classification models.

The *algorithm* level represents special algorithm types of the functions. Many algorithm types are predefined in CWM and many have been added for XELOPES. An example is the *decisionTree* algorithm which belongs to the function *Classification* and represents a decision tree. The algorithm type of *MiningModel* is stored in the variable *algorithm*.

We mention that these function and algorithm types are used in many parts of XELOPES.

Finally, we mention that *MiningModel* contains the variable *miningSettings* of the class *MiningSettings* that will be described in the next section.

### **3.4 Algorithms**

The abstract class *MiningAlgorithm* represents the Data Mining algorithm that constructs a *MiningModel*.

Thus, *MiningAlgorithm* takes a mining input stream of the training data as input and returns the mining model of the mining function as output. The training parameters are passed through the mining settings on model-type level and mining algorithm specification on algorithm level. Through a callback mechanism the training process can be monitored and controlled. The complete data flow is shown in Figure 3.2.

From Figure 3.3 it follows that *MiningAlgorithm* required variables for the mining input stream, mining settings, mining algorithm specification, mining model, and event listener (callback). Additionally, the variable *metaData* allows direct access to the metadata of the mining settings (that is, the metadata of the input stream) and the variable *miningAutomationAssignment* references the mining automation object for automatic model generation, see below.

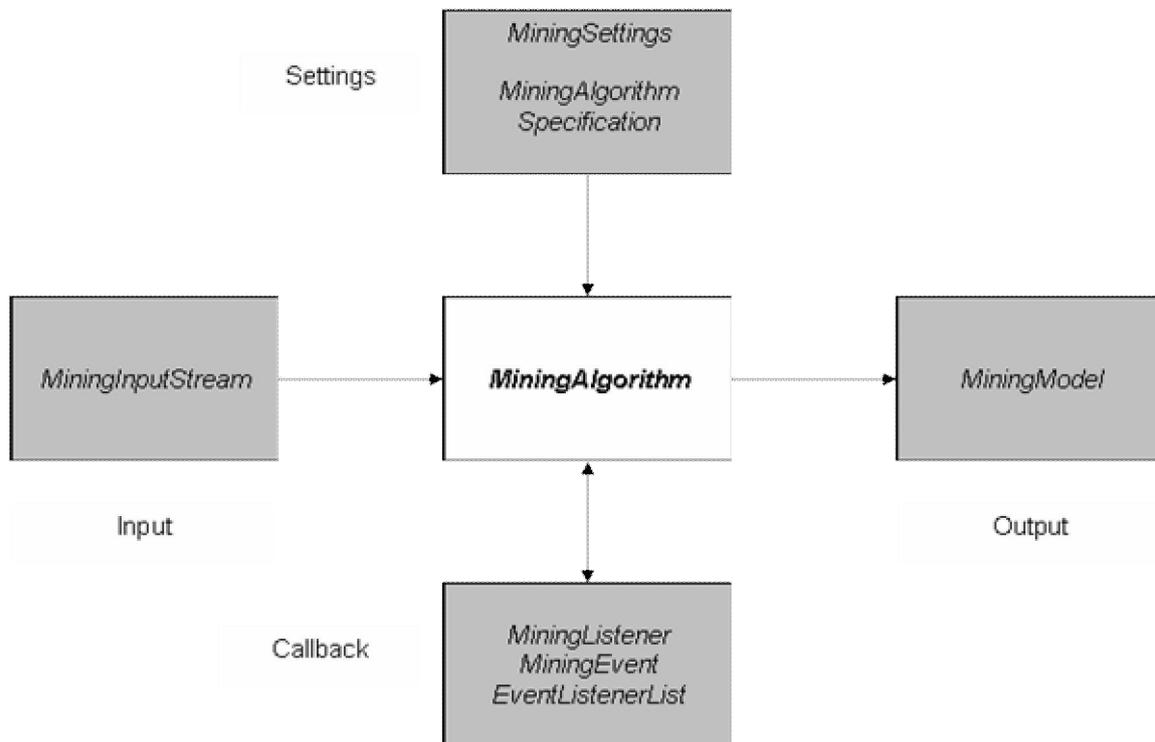


Figure 3.3: Main interfaces of MiningAlgorithm.

The central method of *MiningAlgorithm* is *buildModel* which runs the mining algorithm and returns the mining model created by the algorithm. Internally, *buildModel* calls the protected *runAlgorithm* method of the actual training process. The *buildModelWithAutomation* method generates a mining model using techniques for automatic parameter tuning, allowing to build mining models fully automatically.

*MiningAlgorithms* owns a *verify* method which checks all parameters of the algorithms class for correctness and completeness.

### 3.4.1 Mining Settings

*MiningSettings* contains the general parameters of a mining model independent of the specific mining algorithm that has created the model. For instance, an association rules settings class must contain the minimum support and confidence parameters because they are required for *each* association rules model. In contrast, parameters like the decomposition size, which is required for specific association rules decomposition algorithms, are contained in the algorithm-specific parameter class *MiningAlgorithmSpecification* that will be described in the next section.

*MiningSettings* has a reference to its mining model. The most important variable of *MiningSettings* is *dataSpecification* of the class *MiningDataSpecification*. It contains the metadata of the training data used to build the model and is referred to as the *metadata of the mining model*.

*MiningSettings* contains a *verifySettings* method which checks all parameters of the settings class for correctness and completeness.

## Settings Types

Similar to *MiningModel* each class representing a type of Data Mining settings extends *MiningSettings*. For the example at the beginning of this section, the settings of association rules models are contained in the class *AssociationRuleSettings* which extends *MiningSettings*. Of course, this is the settings class associated with *AssociationRuleMiningModel* mentioned in the previous section.

Along with all mining models XELOPES provides their associated settings classes containing all basic parameters of the respective models.

*MiningSettings* contains the same variables *function* and *algorithm* for storing the function and algorithm type of the mining model. Their values are identical to those of the associated mining model.

### 3.4.2 Mining Algorithm Specification

The algorithm-specific class *MiningAlgorithmSpecification* contains the function and algorithm, the name, the class path, the version, and an array of specific *parameters* of a mining algorithm. The *parameter* variable contains the specific parameters that are defined in *MiningAlgorithmParameter* class. Every parameter is described by its name, type, value, description, setter method, and contains the reference to its associated *MiningAlgorithmSpecification* object.

In most XELOPES implementations, the complete information of *MiningAlgorithmSpecification* for all algorithms and parameters is stored in the configuration file *algorithms.xml*.

#### Example

Example of the section of the fast sequential algorithm *Sequential* of *algorithms.xml*:

```
<AlgorithmSpecification name="Sequential"
  function="Sequential"
  algorithm="sequenceAnalysis"
  classname="com.prudsys.Models.Sequential.Algorithms.Seq.SequentialCycle"
  version="1.0">
  <AlgorithmParameter name="minimumItemSize" type="int" value="1"
    method="setM_minItemSize" descr="Minimum size for large items" />
  <AlgorithmParameter name="maximumItemSize" type="int" value="-1"
    method="setM_maxItemSize" descr="Maximum size for large items" />
</AlgorithmSpecification>
```

### 3.4.3 Algorithm Types

Similar to *MiningModel* and *MiningSettings* each class representing a type of Data Mining algorithms extends *MiningAlgorithm*. For example, the general class of association rules algorithms is *AssociationRulesAlgorithm* which extends *MiningAlgorithm*. Again, this is the algorithm class associated with *AssociationRulesMiningModel* and *AssociationRulesSettings* mentioned in the previous sections.

Along with all mining models and their mining settings, XELOPES provides the associated algorithm classes containing the basic implementations.

## 4 Simplified Interface

The XELOPES standard interfaces as shown in Figure 3.3 have a very clear and object-oriented structure. However, since there exist many different types of Data Mining models with corresponding settings and algorithms, the structure of the *Models* package is quite complex. This is the price to be paid for object-orientation. So especially for simple or very compact applications it might be better to have a less object-oriented, more easy-to-use interface.

To meet this requirement, XELOPES provides the interface *XelopesService* and its extension *LocalXelopesService* (Figure 4.1).

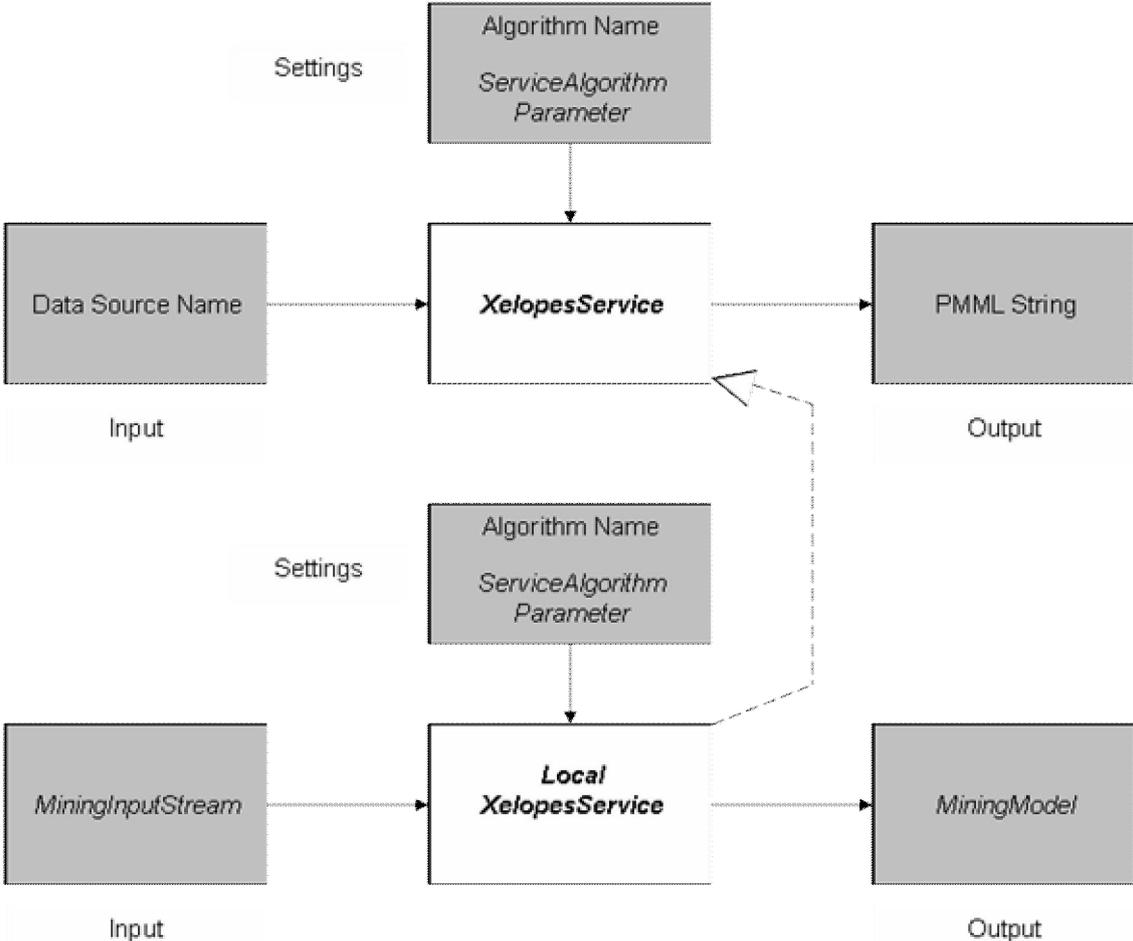


Figure 4.1: Simplified interfaces of using a mining algorithm.

*XelopesService* is designed to create models remotely, e.g. via Web Services. It contains a couple of methods to request XELOPES for its supported functions and algorithms as well as their parameters.

At this, *all* algorithm parameters (from both mining settings and mining algorithm specification) are modelled by the simple class *ServiceAlgorithmParameter* which is very similar to *MiningAlgorithmParameter* but does not refer to other XELOPES parts and thus can easily be used as a remote object for standard communication protocols. *ServiceAlgorithmParameter* stores the parameter's name, type, value, description (like *MiningAlgorithmParameter*) plus the domain (0 - mining settings, 1 – mining algorithm specification), the status (0 – supported, 1 – required) and some other variables of minor importance.

The central methods of *XelopesService*

```
public String buildModelService(String algorithm,
    ServiceAlgorithmParameter[] parameters,
    String sourceName) throws RemoteException;

public double[] applyModelFunctionService(String model,
    String sourceName) throws RemoteException;
```

are used to build and apply mining models remotely. The first method *buildModelService* corresponds to *buildModel* of *MiningAlgorithm*. It requires the name of the algorithm to be invoked, the array of all service algorithm parameters, and the name of the remote data source for training. Notice that all this information can be requested through special service methods of *XelopesService*.

The method returns the PMML string representation of the constructed model that can be easily converted into XELOPES model classes using the methods *readPmml* and *writePmml* of *MiningModel*.

The second method *applyModelFunctionService* corresponds to *applyModelFunction* of *MiningModel* and delivers the values for all data vectors of a specified remote data source. Again, PMML is used to pass the Data Mining model.

### Example

```
// Get XELOPES Service object:
XelopesService xsi = ...;

// Want supported functions:
String[] suppFunc = xsi.getSupportedFunctions();

// Want all supported algorithms:
String[] suppAlg = xsi.getAllSupportedAlgorithms();

// Want all supported association rules algorithms:
String[] suppAssAlg = xsi.getSupportedAlgorithms(
    MiningModel.ASSOCIATION_RULES_ALGORITHM );

// Want all algorithm parameters of algorithm 'FPGrowth':
String selectAlgo = suppAssAlg[0]; // takes first algorithm
selectAlgo = "FPGrowth"; // want 'FPGrowth'
ServiceAlgorithmParameter[] algPar =
    xsi.getAlgorithmParameters(selectAlgo);

// Want Service API data sources available:
String[] dataSources = xsi.getDataSources();
System.out.print("-->Data sources for association rules: ");

// Want meta data of first data source:
String dataSource = dataSources[0];
String dsMeta = xsi.getDataSourceMetaData( dataSource );

// Change values of selected parameter:
LookupService.setSAPValue(algPar, "minimumSupport", "0.5");
LookupService.setSAPValue(algPar, "minimumConfidence", "0.3");
LookupService.setSAPValue(algPar, "itemIdName", "itemId");
LookupService.setSAPValue(algPar, "transactionIdName", "transactId");
LookupService.setSAPValue(algPar, "maximumItemSize", "4");

// Run association rules algorithm and obtain result as PMML string:
String pmmlStr = xsi.buildModelService(selectAlgo, algPar, dataSource);
```

*XelopesService* has mainly been designed to meet the requirements of remote services, resulting in simple data structures for communication. This makes this interface (and its implementations) also a good choice as a utility for easily calling mining algorithms. For local XELOPES calls, a special interface *LocalXelopesService* that extends *XelopesService* has been developed. It provides an additional *buildModelService* method

```
public MiningModel buildModelService(String algorithm,
    ServiceAlgorithmParameter[] parameters,
    MiningInputStream source) throws MiningException;
```

which directly takes a mining input stream and returns a mining model. Notice that no further *applyModelFunctionService* is provided since this functionality now can directly be used through the *applyModelFunction* of the constructed mining model.

### Example

The following example of running a clustering algorithm demonstrates how easy *LocalXelopesService* can be used.

```
// Create XELOPES Local Service object:
LocalXelopesService lxsi = new LocalXelopesServiceImpl();

// Open mining input stream:
MiningInputStream inputData = new MiningArffStream(
    "data/arff/iris.arff" );

// Get algorithm parameters and change two of them:
ServiceAlgorithmParameter[] algPar =
lxsi.getAlgorithmParameters("KMeans");
LookupService.setSAPValue(algPar, "type", "1");
LookupService.setSAPValue(algPar, "numberOfClusters", "3");

// Run clustering algorithm:
MiningModel model = lxsi.buildModelService("KMeans", algPar, inputData);
```

Summing up, the interface *XelopesService* is well-suited for providing XELOPES functions as a remote service whereas the interface *LocalXelopesService* is the easiest way to build mining models locally (replacing the direct usage of *MiningAlgorithm* classes).

# A Appendix

## A.1 Association Rules Example

The following Java code, which provides the same functionality as *com.prudsys.pdm.Examples.AssociationRulesTutorialServiceAPIBuild* from Section 1.1, can be found in *com.prudsys.pdm.Examples.AssociationRulesTutorialBuild* in the *src/* directory:

```
// Create and open input stream
MiningInputStream dataSource =
    new MiningCsvStream("data/csv/transact.csv");
dataSource.open();
MiningDataSpecification metaData = dataSource.getMetaData();

// Get transactional attributes:
CategoricalAttribute categoryItemId =
    (CategoricalAttribute) metaData.getMiningAttribute("itemId");
CategoricalAttribute categoryTransactId =
    (CategoricalAttribute) metaData.getMiningAttribute("transactId");

// Create MiningSettings object and assign metadata:
AssociationRulesSettings miningSettings =
    new AssociationRulesSettings();
miningSettings.setDataSpecification(metaData);

// Assign settings:
miningSettings.setItemId(categoryItemId);
miningSettings.setTransactionId(categoryTransactId);
miningSettings.setMinimumSupport(0.5);
miningSettings.setMinimumConfidence(0.3);
miningSettings.verifySettings();

// Generate mining algorithm specification directly:
String selectAlgo = "FPGrowth";
MiningAlgorithmSpecification miningAlgoSpec =
    MiningAlgorithmSpecification.getMiningAlgorithmSpecification(
        selectAlgo);

// Get class name from algorithms specification:
String className = miningAlgoSpec.getClassName();
if( className == null )
    throw new MiningException( "classname attribute expected." );

// Create algorithm object with default values:
AssociationRulesAlgorithm algorithm =
    (AssociationRulesAlgorithm) Class.forName(className).newInstance();

// Put it all together:
algorithm.setMiningInputStream(dataSource);
algorithm.setMiningSettings(miningSettings);
algorithm.setMiningAlgorithmSpecification(miningAlgoSpec);
// Parameter specific for AssociationRulesAlgorithm but not in MAS:
algorithm.setExportTransactIds(true);
algorithm.setExportTransactItemNames(
    AssociationRulesMiningModel.EXPORT_PMML_NAME_TYPE_XELOPES);
algorithm.verify();

// Run association rules algorithm:
MiningModel associationRulesModel = algorithm.buildModel();
System.out.println("calculation time [s]: "
```

```
+ algorithm.getTimeSpentToBuildModel());

// Display rules
showRules((AssociationRulesMiningModel) associationRulesModel);

// Write result into PMML file:
FileWriter writer =
    new FileWriter("data/pmml/AssociationRulesModelT.xml");
associationRulesModel.writePmml(writer);
```

## A.2 Classification Example

The following Java code, which provides the same functionality as *com.prudsys.pdm.Examples.DecisionTreeTutorialServiceAPIBuild* from Section 1.2, can be found in *com.prudsys.pdm.Examples.DecisionTreeTutorialBuild* in the *src/* directory:

```
// Create and open input stream
MiningInputStream dataSource =
    new MiningCsvStream("data/csv/cancellingTrain.csv");
dataSource.open();
MiningDataSpecification metaData = dataSource.getMetaData();

// Get target attribute:
MiningAttribute targetAttribute =
    metaData.getMiningAttribute("CANCELLER");

// Create MiningSettings object and assign meta data:
DecisionTreeSettings miningSettings = new DecisionTreeSettings();
miningSettings.setDataSpecification( metaData );

// Assign settings:
miningSettings.setTarget(targetAttribute);
miningSettings.setMinNodeSize(0.3,
    DecisionTreeSettings.SIZE_UNIT_PERCENTAGE);
miningSettings.setMaxDepth(100);
miningSettings.verifySettings();

// Get default mining algorithm specification from 'algorithms.xml':
MiningAlgorithmSpecification miningAlgorithmSpecification =
    MiningAlgorithmSpecification.getMiningAlgorithmSpecification(
        "Decision Tree (General)" );
if( miningAlgorithmSpecification == null )
    throw new MiningException("Can't find application Decision Tree.");

// Get class name from algorithms specification:
String className = miningAlgorithmSpecification.getClassName();
if( className == null )
    throw new MiningException( "classname attribute expected." );

// Create algorithm object with default values:
DecisionTreeAlgorithm algorithm = (DecisionTreeAlgorithm)
    GeneralUtils.createMiningAlgorithmInstance(className);

// Put it all together:
algorithm.setMiningInputStream( dataSource );
algorithm.setMiningSettings( miningSettings );
algorithm.setMiningAlgorithmSpecification(miningAlgorithmSpecification);
algorithm.verify();

// Build the mining model:
MiningModel model = algorithm.buildModel();
System.out.println("calculation time [s]: " +
    algorithm.getTimeSpentToBuildModel());

// Show results:
showTree((DecisionTreeMiningModel) model);

// Write to PMML:
FileWriter writer = new FileWriter("data/pmml/DecisionTreeModelT.xml");
model.writePmml(writer);
```