Knowledge Extraction

based on

Evolutionary Learning

Reference manual

Version: 1.0.
Date: 20-5-2009.

Contents

1 Basic KEEL developement guidelines

1.1 Introduction

The purpose of this document is to describe some basic concepts about the structure of KEEL (Knowledge Extraction based on Evolutionary Learning) and the format of its internal files.

The aim of this section is to present the KEEL framework, describing some guidelines to help a potential developer to build new methods inside the KEEL environment. The next sections will deal with the formats of the configuration files of KEEL (including data sets files, method descriptions and so on). Finally, the last section describes the API dataset of KEEL, which is used to handle and check the data sets files.

1.2 Developing a new method

Before to start the task of developing a new method inside of KEEL environment, some operations have to be performed in order to fully integrate it. By following this guidelines, a developer can left all the input/output operations to be accomplished by KEEL environment, focusing its efforts in the construction of the method itself.

The steps needed to complete the integration of a new method in KEEL are:

- Reading of the configuration file.
- Development of the method.
- Writing the output files.
- Registering the method in KEEL.
- Making the use case files.
- Building the executables of the method.

1.2.1 Reading of the configuration file

The KEEL methods only accept one parameter: The name and path of a configuration file. A typical main class of a method can be the following:

```
public class Main {

  private static Clas classifier;

  public static void main (String args[]) {

    if (args.length != 1){
      System.err.println("Error.");
    } else {
      classifier = new Clas(args[0]);
      classifier.execute();
    }

  } //end-method
}//end-class
```

The configuration file contains information about the input and output files of the method. In addition, it contains the values for all the parameters defined. A full description of the configuration files can be found in section 3.

By interpreting this file, the method should be able to acquire the correct values of its parameters, including the seed to initialize the random number generator if the method needs it.

Also, the names and paths of the input and output files are specified inside. Usually, a KEEL method employs two input data files: The training file, containing the data set which should be employed in the train phase of the method, and the test file, containing the data set which should be employed in the test phase. In addition, any method excepting the preprocessing methods and the test methods specify a third file, the validation file. This file contains a copy of the original dataset of the experiment, which can be used in comparisons with the train data.

The format of the data files is explained in section 4. These files must be handled with care, because they will be employed not only by the method,

but also by the KEEL API dataset (see section 7) in order to load and check the data in an efficient way.

Any KEEL method must define at least two output files: A train output file and a test output file. In addition, it is possible to define additional output files in the configuration file. They will be explained in the next subsections of this guide.

1.2.2 Developement of the method

The development of the method can be done in any programming environment. The only requirements are: The method must be developed with the Java programming language, and it must employ a package structure whose root will be the **keel/src** directory, where the sources of any KEEL method are located.

1.2.3 Writing the output files

As is explained before, at least two output files must be produced by the method (the train output file, and the test output file). Its format is described in section 5.

If it is desired to employ additional output files, they also can be created at the end of the execution on the method. These additional files will get its name from the configuration file. Also, it is important to note that, in order to let the KEEL GUI automatically generate the names of these files, the number of additional outputs of the methods must be placed in the corresponding method description file.

1.2.4 Registering the method in KEEL

When the method have been fully coded, it must be registered in the KEEL configuration files, to allow the KEEL GUI to employ the new method.

The first step is to create a method description file. The format of these files is fully described in section 2.

The second step involves modifying the master description file of each category method. Currently, 11 categories are defined:

- Discretization

- Educational Methods
- Educational Preprocess
- Feature Selection
- Instance Selection
- Method
- Postprocess
- Preprocess
- Tests
- TransOthers
- Visualize

When the correct master description file have been found (please, ask to a KEEL project manager if it is not clear what file have to be modified), a new registry containing the definition of the method must be created. The KEEL master description file registers have the following structure:

```
<method>
  Header
  Input
  Output
</method>
```

The header is composed by four nodes:

```
<name>Disc-UniformWidth</name>
<family>Discretizers</family>
<jar_file>Disc-UniformWidth.jar</jar_file>
<problem_type>unspecified</problem_type>
```

Name: The name of the method.

Family: The category of the method.

Jar File: The name of the Jar file which contains the method.

Problem Type: The class of problems which can manage the method. There are defined 4 classes:

- **Classification**, for supervised classification problems.
- **Regression**, for regression problems.
- **Unsupervised**, for unsupervised classification problems (e.g. clustering).
- **Unspecified**, for any problem (supervised classification, unsupervised classification or regression).

The input and output parts defines the types of data which the method is able to manage, both in input data and output data. Their fields must specify which types are allowed, by employing "yes" and "no" keywords. A description of the fields is shown as follows:

```
<continuous>Yes</continuous>
<integer>Yes</integer>
<nominal>Yes</nominal>
<missing>Yes</missing>
<imprecise>No</imprecise>
<multiclass>Yes</multiclass>
<multioutput>No</multioutput>
```

Continuous: The method is able to work with continuous values.

Integer: The method is able to work with integer values.

Nominal: The method is able to work with nominal values.

Missing: The method is able to handle missing values.

Imprecise Value: The method is able to work with imprecise values.

Multiclass: The method is able to work with problem which defines more than 2 classes.

Multioutput: The method is able to work with data which defines more than 1 output for each instance.

When the header, input and output sections were completely defined, then the new registry can be place inside the corresponding master description file. Below is shown a valid example of registry:

```
<method>
  <name>Disc-UniformWidth</name>
  <family>Discretizers</family>
  <jar_file>Disc-UniformWidth.jar</jar_file>
  <problem_type>unspecified</problem_type>
  <input>
    <continuous>Yes</continuous>
        <integer>Yes</integer>
        <nominal>Yes</nominal>
        <missing>Yes</missing>
        <imprecise>No</imprecise>
        <multiclass>Yes</multiclass>
    <multioutput>No</multioutput>
  </input>
  <output>
    <continuous>No</continuous>
        <integer>No</integer>
        <nominal>Yes</nominal>
        <missing>Yes</missing>
        <imprecise>No</imprecise>
        <multiclass>Yes</multiclass>
    <multioutput>No</multioutput>
  </output>
</method>
```

1.2.5 Making the use case files

When developing a new method, it is important to document properly its functions and objectives. Also, the users should be able to look up relevant information about the method (a brief description, some references, the description of its parameters, etc.) when they select the method in KEEL.

To manage this information, the KEEL GUI defines the use case files, which are XML files containing all the relevant information needed to employ any KEEL method. A full description of the use case files can be found in section 6.

1.2.6 Building the executables

When the method was fully developed, and its relevant configuration files have been created, the last step is to add it to the *build.xml* file (a ANT script file), so the new versions of KEEL could be able to build it inside the KEEL environment. The *build.xml* is a critical file, so it is not recommended to modify it without authorization of a KEEL project manager.

The *build.xml* changes dynamically with any new version of KEEL, thus its is not possible to fully describe its structure here. However, it is possible to describe which part of the file must be changed to allow the inclusion of new methods.

Firstly, the **jar** target must be found. It should have the following structure:

```
<target name="jar" depends="compile"
description="Build jars">
```

The **jar** target is composite by a great number of tasks, every one dealing with the construction of a jar file for each method. Inside this target, the construction of the new jar file must be described as another task. Here is a valid example:

```
<jar
   jarfile="${distMet}/KNN.jar" manifest=
   "${src}/keel/Algorithms/Lazy_Learning/KNN/Manifest">
   <fileset dir="${bin}" includes=
   "keel/Algorithms/Lazy_Learning/KNN/**/*.class"/>
   <fileset dir="${bin}"
includes="keel/Algorithms/Preprocess/Basic/**/*.class
   org/core/**/*.class
   keel/Dataset/**/*.class
   keel/Algorithms/Lazy_Learning/*.class"/>
</jar>
```

The task must define the locations of the new jar file and their corresponding manifest file. Also, must include the files from the classes which compose the method. Also, the files from the imported classes are required to fully describing the task.

2 Method Description files

Every method in KEEL (e.g., a preprocess method, a test ...) has assigned a XML file which describes its main characteristics. This file will be employed by the KEEL GUI to allow the user to select the values of the parameters of any execution of the method.

The KEEL Method Description files are located under the **../dist/algorithm** directory, inside of the folder where its associated .JAR file is generated (e.g., **../dist/algorithm/methods**). Each Method Description file is an XML composed by a unique root node, <**algorithm_specification**> . This node is divided into two parts:

```
<algorithm_specification>
        Header
        Parameters
</algorithm_specification>
```

Header : Basic information about the method.

Parameters : A list of parameters of the method.

2.1 Header

The header is composed by four nodes:

```
<name>K Nearest Neighbors Classifier</name>
<nParameters>2</nParameters>
<seed>0</seed>
<nOutput>1</nOutput>
```

<**Name**> : The name of the method.

<**nParameters**> : The number of parameters of the methods (must be 0 or highger). Seed values employed to initialize random number generators are not counted here.

<**Seed**> : Defines if the method will need a seed to initialize a random number generator. Valid values are 1, if a seed is needed, or 0, if not.

<**nOutput**> : The number of additional output files which will be generated by the method.

2.2 Parameters

The parameters of the method are listed consecutively. A <parameter> node is employed to describe each one. Each <parameter> is composed by the following nodes:

```
<parameter>
  <name>K Value</name>
  <type>integer</type>
  <domain>
    <lowerB>1</lowerB>
    <upperB>100</upperB>
  </domain>
  <default>1</default>
</parameter>
```

<**Name**> : The name of the parameter.

<**Type**> : Type of parameter. KEEL defines four valid types:

integer : An integer value. Can be positive, 0, or negative.

real : A real value. The dot "**.**" is employed as decimal separator.

text : A string of text.

list : A predefined list of text options

When employing **text** parameters, no checking operations are done by the KEEL GUI. Thus, the use of list parameters is recommended when a fixed number of text options are defined, so the method does not have to check the parameters by itself.

<**Domain**> : The domain of the parameter. For **list** parameters is mandatory. For **text** parameters cannot be defined. For **integer** and **real** parameters is optional (if it is not defined, the KEEL GUI will not check its value).

<**lowerB**> : The lower value of the parameter (valid only in **integer** and **real** parameters).

<**upperB**> : The highest value of the parameter (valid only in **integer** and **real** parameters).

<**item**> : A text value for the parameter (it can be employed only in **list** parameters).

<**Default**> : Default value of the parameter.

2.3 Example of use

This is a valid example of a Method Description file:

```
<algorithm_specification>
  <name>K Nearest Neighbors Classifier</name>
  <nParameters>2</nParameters>
  <seed>0</seed>
  <nOutput>1</nOutput>
  <parameter>
    <name>K Value</name>
    <type>integer</type>
    <domain>
      <lowerB>1</lowerB>
      <upperB>100</upperB>
    </domain>
    <default>1</default>
  </parameter>
  <parameter>
    <name>Distance Function</name>
    <type>list</type>
    <domain>
      <item>Manhattan</item>
      <item>Euclidean</item>
    </domain>
    <default>Euclidean</default>
  </parameter>
</algorithm_specification>
```

3 Method Configuration files

In KEEL, every method uses a configuration file to extract the values of the parameters which will be employed during its execution. Although it is generated automatically by the KEEL GUI (by using the information contained in the corresponding method description file, and the values of the parameters specified by the user), it is important to fully describe its structure because any KEEL method must be able to completely read it, in order to get the values of its parameters specified in each execution.

Each configuration file has the following structure:

algorithm : Name of the method.

inputData : A list with the input data files of the method.

outputData : A list with the output data files of the method.

parameters : A list of parameters of the method, containing the name of each parameter and its value (one line is employed to each one).

3.1 Input files

The files in the inputData list must be separated by one space, being each one surrounded by quotation marks (“). If a validation file is employed by the method, the files will appear in the following order:

input data= <training file> <validation file> <test file>

If not, the order employed will be:

input data= <training file> <test file>

The validation file is a copy of the original train data employed at the start of the experiment. It is often employed for comparison tasks between the initial data and training data when it has been preprocessed.

3.2 Output files

- A file with the output for training data.
- A file with the output for test data.
- (Optional) Additional output files.

Additional output files can be specified in this list. Although the will be not managed by other KEEL methods, it is possible to employ them to extract useful information from the execution of the method (representation of models extracted, additional outputs, performance measures ...). They must be marked with the extension *.txt*.

3.3 Parameters

The rest of the configuration file is used to describe the values of the parameters. In each line appears one parameter, followed by a "**=**" sign and its assigned value.

If the method needs a seed to initialize a random number generator, it must be the first parameter described, employing "**seed**" as name of the parameter.

3.4 Example of use

This is a valid example of a Method Configuration file (data files lists are not fully shown):

```
algorithm = genetic algorithm
inputData = ''./dataset/iris/iris11.dat'' ...
outputData = ''./result/ga/iris/result1.tra'' ...

seed = 1234578
nGenerations = 500
cross = two_points
crossProbability = 0.6
mutationProbability = 0.2
```

4 Data files

In KEEL, the data sets are managed by plain ASCII text files, with the *.dat* extension. Usually, they are located under the **../dist/data** directory, each one in its own folder (which also should contains the partitions created from the whole data set). In addition, preprocess methods will also create data files as its output, which will be placed on the **../datasets** directory of its experiment.

This section describes the format employed to define them (which is fairly similar to WEKA arff format). Each KEEL data file is composed by 2 sections:

Header : Basic metadata describing the data set.

Data : Content of the dataset.

In both sections it is possible to insert comments, by employing the "%"character.

4.1 Header

The header is composed by the following metadata:

```
@relation bupa2
@attribute mcv nominal {a,b,c}
@attribute alkphos integer [23, 138]
@attribute sgpt integer [4, 155]
@attribute sgot integer [5, 82]
@attribute gammagt integer [5, 297]
@attribute drinks real [0.0, 20.0]
@attribute selector {true,false}
@inputs mcv, alkphos, sgpt, sgot, gammagt, drinks
@outputs selector
```

@relation : The name of the data set.

@attribute : Describes one attribute of the data (a column). It is possible to define three different types of attributes:

integer : @attribute <name> integer [min, max]

real : @attribute <name> real [min, max]

nominal : @attribute <name> {Value1,value2,...,valueN}

The <name> is the identifier of the attribute. Its maximum length allowed is 12 characters. The *min* and *max* values fon integer and real attributes, and the list of possible values for nominal attributes, are optional. If they are missing, the corresponding values will be extracted from the data by the KEEL data process module.

@inputs : Identifiers of the attributes which must be processed as inputs.

@outputs : Identifiers of the attributes which must be processed as outputs.

The @inputs and @outputs definitions are optional. If they are missing, all the attributes will be considered as input attributes, except the last, which will be considered as output attribute.

4.2 Data

The data instances are represented as rows of comma separated values, where each value corresponds to one attribute, in the order defined by the header. Missing or null values are defined as <null> or *?* .

If the dataset corresponds to a classification problem, the output type must be nominal:

```
...
@attribute selector {true,false}
...
@outputs selector
@data
a, 92, 45, 27, 31, 0.0, true
a, 64, 59, 32, 23, <null>, false
b, 54, <null>, 16, 54, 0.0, false
```

If the dataset corresponds to a regression problem, the output type must be real:

```
...
@attribute selector real [0.0, 20.0]
...
@outputs selector
@data
a, 92, 45, 27, 31, 0.0, 0.9
a, 64, 59, 32, 23, <null>, 17.5
b, 54, <null>, 16, 54, 0.0, 3.5
```

4.3 Example of use

This is a valid example of a data file:

```
@relation bupa2
@attribute mcv nominal {a,b,c}
@attribute alkphos integer [23, 138]
@attribute sgpt integer [4, 155]
@attribute sgot integer [5, 82]
@attribute gammagt integer [5, 297]
@attribute drinks real [0.0, 20.0]
@attribute selector {true,false}
@inputs mcv, alkphos, sgpt, sgot, gammagt, drinks
@outputs selector
@data
a, 92, 45, 27, 31, 0.0, true
a, 64, 59, 32, 23, <null>, false
b, 54, <null>, 16, 54, 0.0, false
a, 78, 34, 24, 36, 0.0, false
a, 55, 13, 17, 17, 0.0, false
b, 62, 20, 17, 9, 0.5, true
c, 67, 21, 11, 11, 0.5, true
a, 54, 22, 20, 7, 0.5, true
b, 60, 25, 19, 5, 0.5, true
b, 52, 13, 24, 15, 0.5, true
b, 62, 17, 17, 15, 0.5, true
```

5 Output files

Every method in KEEL must produce at least two output files: A train results file (marked with the extension *.tra*) and a test results file (marked with the extension *.tst*). Although the method can employ additional output files to show more information about the process performed, those additional files must be handled entirely by the method. Thus, KEEL will only handle the two standards output files.

Both output files share the same structure: They are composite by the same header of the data employed as input of the method, and a set of rows (one for each instance of the data set) describing the expected outputs and the outputs obtained by the application of the method. Thus, they are structured as follows:

$<\text{Expected}_{1,1}> \ldots <\text{Expected}_{1,n}><\text{Method}_{1,1}> \ldots <\text{Method}_{1,n}>$

$<\text{Expected}_{2,1}> \ldots <\text{Expected}_{2,n}><\text{Method}_{2,1}> \ldots <\text{Method}_{2,n}>$

5.1 Example of use

As related before, the structure of the output method is derived from the input data files employed. For example, if the following file is employed as input data of a method:

```
@relation banana
@attribute at1 real [-3.09, 2.81]
@attribute at2 real [-2.39, 3.19]
@attribute class {-1.0, 1.0}
@inputs at1, at2
@outputs class
@data
1.14, -0.114, -1.0
-1.05, 0.72, -1.0
-0.916, 0.397, 1.0
-1.09, 0.437, 1.0
-0.584, 0.0937, 1.0
1.83, 0.452, -1.0
-1.25, -0.286, 1.0
...
```

A valid output file should be formatted like the following file (note the single spacing between columns):

```
@relation banana
@attribute at1 real [-3.09, 2.81]
@attribute at2 real [-2.39, 3.19]
@attribute class {-1.0, 1.0}
@inputs at1, at2
@outputs class
@data
-1.0 -1.0
-1.0 1.0
1.0 -1.0
1.0 1.0
1.0 1.0
-1.0 -1.0
1.0 1.0
...
```

By employing this structure, it is easy to understand how well the task was performed by the method (in the example shown above, the method failed to predict the output for the instances 2 and 3, and predicted correctly the remaining ones).

6 Use Case files

The use case files of KEEL provides valuable information to understanding every of the methods which are available to use. They are XML files, located in the **../dist/help** directory.

Each KEEL use case file is composed by 4 sections:

```
<method>
Name
Reference
General Description
Example
</method>
```

Name : The name of the method.

Reference : A list of references associated with the method.

General Description : Generic information about the method.

Example : A example about the use of the method.

6.1 Name

The first part of the use case contains the name of the method, enclosed by <name> tags:

```
<name>Name of the method</name>
```

6.2 Reference

The second part of the use case contains a list of associated references. They are enclosed each one by <ref> tags.

```
<reference>
        <ref>First reference</ref>
        <ref>Second reference</ref>
</reference>
```

6.3 General description

The general description describes some common features about the method, as its objective, parameters, type of data which can be handle, etc. It is composed by the following fields:

Type: General type of method.

Objetive: Objective of the method.

How work: A brief explanation about how the method works.

Parameter spec: A specification of each parameter of the method. They are enclosed each one by <param> tags.

Properties: Generic properties of the methods. Each field can contain "yes" or "no" strings, defining the following capabilities of the method:

Continuous: The method is able to work with continuous values.

Discretized: The method is able to work with discretized values.

Integer: The method is able to work with integer values.

Nominal: The method is able to work with nominal values.

Value Less: The method is able to handle missing values.

Imprecise Value: The method is able to work with imprecise values.

```
<generalDescription>
<type>General type of method.</type>
<objective>Objective of the method.</objective>
<howWork>Explanation of how works.</howWork>
<parameterSpec>
        <param>Parameter one</param>
        <param>Parameter two</param>
</parameterSpec>
<properties>
        <continuous>Yes</continuous>
        <discretized>Yes</discretized>
        <integer>Yes</integer>
        <nominal>Yes</nominal>
        <valueLess>Yes</valueLess>
        <impreciseValue>Yes</impreciseValue>
</properties>
</generalDescription>
```

6.4 Example

The last part of the use case is employed to show an example of utilization of the method. It can be employ any number of lines (though, it is not recommended to place huge examples here).

```
<example>
A example of utilization of the method.
</example>
```

6.5 Example of use

A valid example of a use case file is shown in the next page:

```
<method>
  <name>Prototype Nearest Neighbor</name>
  <reference>
  <ref>Chin-Liang, Chang. Finding Prototypes for
  Nearest Neighbor Classifiers. IEEE Trans.
   on Computers, vol. c-23, No. 11, 1179-1184.</ref>
  </reference>
  <generalDescription>
    <type>Preprocess Method.</type>
    <objective>Reduce the size of the training
    set without losing precision or accuracy
    in order to a posterior classification</objective>
    <howWork>This algorithm merge nearest prototypes
    of the set, if classification accuracy of the
    original training data set does not decrease and
    if they have got the same class.
    If not, they are removed from the set.</howWork>
    <parameterSpec>
      <param>Percentage of prototypes: Real</param>
    </parameterSpec>
    <properties>
      <continuous>Yes</continuous>
      <discretized>Yes</discretized>
      <integer>Yes</integer>
      <nominal>Yes</nominal>
      <valueLess>No</valueLess>
      <impreciseValue>No</impreciseValue>
    </properties>
  </generalDescription>
  <example>
  Problem type: Classification
  Method: PG-PNN
  Dataset: iris
  Parameters: default values
  We can see output set in Experiment\Results\PG-PNN:
  ...
</example>
</method>
```

7 API Dataset

One of the main components of KEEL is the API Dataset. It manages the entire process of acquisition, processing and validation of the data files, offering the data sets to the developer in a suitable way, freeing him from the task of acquiring the data needed to perform any experiment.

This section describes three key concepts of the API Dataset:

- **Data files grammar**. The grammar employed to define the data files. Any file generated by this grammar will be a valid data file, according to the rules shown in section 4.
- **Semantic restrictions of the data files**. Apart from the syntax restrictions, some semantic verifications are performed by the API Dataset over the data files.
- **Description of the classes**. To close this section, the main public classes of the API Dataset are described.

7.1 Data files grammar

In this subsection is shown the grammar which describes the format of the KEEL data files. The final tokens of the grammar are:

- $\{\}$. Denotes the void production. It is also known as λ or ϵ.
- **IDENT**. Denotes an identifier (IDENT = ('A' -'Z' , 'a'-'z' , '0'-'9')*).
- **INTEGER**. Is an integer value (INTEGER = (0 . 9)$^+$).
- **REAL** . Is a real value (REAL = (0-9)$^+$[.(0-9)*]).

```
principal -> Relation
          -> Attributes
          -> Inputs
          -> Outputs
          -> Data

Relation  -> "@relation" IDENT
```

```
Attributes -> "@attribute" IDENT attributeType Attributes
           -> {}

attributeType -> "integer" intBoundaries
              -> "real" realBoundaries
              -> "{" IDENT idList "}"

intBoundaries -> "[" INTEGER "," INTEGER "]"
              -> {}

realBoundaries -> "[" REAL "," REAL "]"
               -> {}

idList -> "," IDENT idList
       -> {}

Inputs -> "@inputs" IDENT idList
       -> {}

Outputs -> "@outputs" IDENT idList
        -> {}

Data -> @data dataList

dataList -> lineData dataList
         -> {}

lineData -> IDENT lineDataCont

lineDataCont -> "," IDENT lineDataCont2

lineDataCont2 -> "," IDENT lineDataCont2
              -> {}
```

7.2 Semantic restrictions of the data files

7.2.1 Attributes

An attribute can be defined as integer, real or nominal, as the grammar of the data files defines. It is optional to define de minimum and maximum values, or the list of values for any attribute (if they are not defined, they correct values will be extracted during the processing of the training file). Anyway, if they are defined for integer or real attributes, the minimum value defined must be lower than the maximum.

This way, the limits of the values for any attribute will be established during the processing of the training file. However, it is possible to find values in the test file which exceed the limits for a concrete attribute (i.e., in some schemes of cross-validation). Depending of the type of the attribute, the actions performed by the API dataset are the following:

- **Integer or Real attributes:** The new value is changed by its nearest correct value (i.e, if the value is greater than the maximum, it is replaced by the maximum; if it is lower than the minimum, it is replaced by this one)
- **Nominal attributes:** The new value is accepted, and the domain of the attribute is enlarged, adding the new value. In addition, the flag newValueInTest is marked on.

Finally, if one of these cases appears, the API Dataset throws a *TestDataBoundsExcedeedException* to inform about the changes performed. However, the files will be parsed correctly.

7.2.2 Inputs and outputs definition

The definition of inputs and outputs in the data files is optional. The API Dataset will automatically extract the missing definitions, following these rules:

- If no outputs are defined:
  - If no input are defined, the last attribute is taken as output. The remaining ones will be taken as inputs.

  - If there are some inputs defined, the attributes not marked as inputs will be taken as outputs.

- If no inputs are defined, the attributes not marked as outputs will be taken as inputs.
- If inputs and outputs are defined, those attributes who are not currently defined in one of these categories, are discarded.

Also, it is important to note that the inputs and outputs attributes will be defined in the same order as they appear in the header of the data file.

7.2.3 Missing values

The API Dataset allows the presence of missing values in the data files, defined with the <null> or *?* tokens. However, only input attributes can present missing values. If a missing value is detected in an output attribute, a *OutputValueNotKnownException* will be cast, aborting the processing of the data file.

7.2.4 Train and test files

The semantic verifications performed by the API Dataset will vary depending on the concrete data file processed. Concretely, the actions performed are:

- The definition of the attributes is taken from the training file.
- During the test file reading, the definitions of the attributes are checked. If they are not consistent with the ones read from the training file, the processing of the test file is aborted. Moreover, the inputs and outputs defined by the test file must be the same which were defined by the training file. Otherwise, the processing of the test file will be aborted.

7.3 Description of the classes

The API Dataset is composed by four main classes:

- InstanceSet: This class contains a complete set of instances defining a data base.
- Instance: This class represents a single instance.
- Attributes: This static class contains definitions about every attribute of the data contained in the Instance set.
- Attribute: This class contains relevant information about a single attribute.

The next subsections will describe their main characteristics.

7.3.1 InstanceSet

This class contains a complete set of instances. Its public methods are:

- **numInstances**. Returns the number of instances of the Instance Set.
- **getInstance**. Returns a concrete instance contained in the Instance Set.
- **getInstances**. Returns an array with all the instances of the Instance Set.

7.3.2 Instance

The objects of this class represents instances of the data sets. Its pubic methods are:

- **getInputRealValues**. Returns an array containing all the input values of the instance (only the positions with INTEGER or REAL attributes values will produce a value).
- **getInputNominalValues**. Returns an array containing all the input values of the instance (only the positions with NOMINAL attributes values will produce a value).

- **getInputMissingValues**. Returns a boolean array defining which input values are missing.
- **getInputRealValue**. Returns the value of a concrete input attribute (only the positions with INTEGER or REAL attributes values will produce a value).
- **getInputNominalValue**. Returns the value of a concrete input attribute (only the positions with NOMINAL attributes values will produce a value).
- **getInputMissingValue**. Returns a boolean value defining if the input value is missing.
- **getOutputRealValues**. Returns an array containing all the output values of the instance (only the positions with INTEGER or REAL attributes values will produce a value).
- **getOutputNominalValues**. Returns an array containing all the output values of the instance (only the positions with NOMINAL attributes values will produce a value).
- **getOutputMissingValues**. Returns a boolean array defining which output values are missing.
- **getOutputRealValue**. Returns the value of a concrete output attribute (only the positions with INTEGER or REAL attributes values will produce a value).
- **getOutputNominalValue**. Returns the value of a concrete output attribute (only the positions with NOMINAL attributes values will produce a value).
- **getInputMissingValue**. Returns a boolean value defining if the output value is missing.
- **getAllInputValues**. Returns an array containing all the input values. REAL values are returned as double values. INTEGER values are casted to double. NOMINAL values are transformed to INTEGER and casted to double.
- **getAllOutputValues**. Returns an array containing all the output values. REAL values are returned as double values. INTEGER values are casted to double. NOMINAL values are transformed to INTEGER and casted to double.

7.3.3 Attributes

Attributes is an static class which stores the definitions of the attributes represented in the data set. It contains an array of Attribute objects, and two additional arrays storing references about the input and output attributes. The order of the attributes stored is the same order than it was found in the input data file.

Its public methods are:

- **getInputAttributes**. Returns an array containing all the input Attributes.
- **getOutputAttributes**. Returns an array containing all the output Attributes.
- **getInputAttribute**. Returns a single input attribute.
- **getOutputAttribute**. Returns a single output attribute.
- **getAttribute**. Returns a single attribute, defined neither as input nor as output attribute.
- **getNumInputAttributes**. Returns the number of input attributes.
- **getNumOutputAttributes**. Returns the number of output attributes.
- **getNumAttributes**. Returns the number attributes, including input, output and undefined ones.

7.3.4 Attribute

The Attribute class contains the definition an attribute of the dataset. Its public methods are:

- **getType**. Returns a integer value defining the type of the attribute (the type is defined as NOMINAL, INTEGER or REAL).
- **getName**. Returns the name of the attribute.
- **getMinAttribute**. Returns the minimum value of the attribute (only available in INTEGER or REAL attributes).

- **getMaxAttribute**. Returns the minimum value of the attribute (only available in INTEGER or REAL attributes).
- **getNominalValuesList**. Returns an array with all the values defined for the attribute (only available in NOMINAL attributes).
- **convertNominalValue**. Converts a nominal value to its representations as integer (an integer between [0...N-1, where N is the number of values defined for the attribute).
- **getDirectionAttribute**. Returns an integer showing if the attribute is defined as input attribute (INPUT), output attribute (output), or undefined (DIR_NOT_DEF)
- **getNewValuesInTest**. Returns n array with the new values of the attribute observed in test data.