KDDML Language: Reference Guide

ANDREA ROMEI

University of Pisa
Department of Computer Science
`http://kdd.di.unipi.it/kddml`

October 5, 2005
KDDML System Version: 2.0.15 beta

Abstract

Knowledge discovery in databases (KDD) covers a wide range of application domains (retail, marketing, finance, e-commerce, biology, privacy, only to cite a few ones), several models of representing extracted patterns and rules (including classification models, association rules, sequential patterns, clusters) and a large number of algorithms for data preprocessing, model extraction and model reasoning.
KDDML is a middleware XML-based language (and system) needed to support the development of final applications or higher level systems which need a mixture of database access, data preprocessing, mining extraction and deployment.
As the name suggests, KDDML is heavily based on XML as a representation language for data, models and queries. The language is primarily intended as a middleware language on the basis of which higher abstraction levels can be built, such as vertical applications or more declarative languages. Also, the language tries to be as much as possible independent from lower level implementations of data mining algorithms, with the aim of confining the technicalities at the level of the implementation of the KDDML system.
This document describes in detail KDDML as KDD language whose design principles are motivated by requirements derived from recurring patterns in the KDD process.

Background
Data Mining, Knowledge Discovery in Databases, XML, Document Type Definition (DTD).

Licence

Copyright

CONTENTS

1 Getting started **7**
- 1.1 Motivations . . . 7
- 1.2 KDDML language overview . . . 9
  - 1.2.1 A simple sample . . . 9
  - 1.2.2 KDDML as typed language . . . 10
- 1.3 Organization of this guide . . . 11

2 KDDML Objects **12**
- 2.1 Objects Hierarchy . . . 12
- 2.2 Data representation . . . 13
  - 2.2.1 Logical data . . . 14
  - 2.2.2 Taxonomies . . . 16
  - 2.2.3 Physical data: relational tables, transactional tables and timestamp tables . . . 18
  - 2.2.4 Preprocessing tables . . . 21
- 2.3 Knowledge representation . . . 23
  - 2.3.1 Association model . . . 24
  - 2.3.2 Sequence model . . . 26
  - 2.3.3 Tree model . . . 26
  - 2.3.4 Clustering model . . . 28
  - 2.3.5 Hierarchy model . . . 30
- 2.4 KDDML Scalar . . . 30
- 2.5 Queries representation . . . 30

3 KDDML operators **39**
- 3.1 Classification of operators . . . 40
  - 3.1.1 I/O operators . . . 40
  - 3.1.2 Preprocessing operators . . . 40
  - 3.1.3 Mining operators . . . 42

3.1.4 Postprocessing operators 42
3.1.5 Control flow operators 44
3.1.6 Unclassified operators 44
3.2 Operators specification 44
3.2.1 ARFF_LOADER 44
3.2.2 ARFF_WRITER 45
3.2.3 CALL_QUERY 47
3.2.4 CLUSTER_CENTROID 48
3.2.5 CLUSTER_LOADER 48
3.2.6 CLUSTER_MINER 49
3.2.7 CLUSTER_NUMBER 50
3.2.8 CLUSTER_PARTITION 51
3.2.9 CLUSTER_PARTITION_SPARROW 51
3.2.10 DATABASE_LOADER 52
3.2.11 DATABASE_WRITER 54
3.2.12 EXT_CALL 55
3.2.13 HIERARCHY_LOADER 56
3.2.14 IF 56
3.2.15 MISCLASSIFIED 57
3.2.16 PAR_QUERY 58
3.2.17 PMML_CLUSTER_LOADER 59
3.2.18 PMML_RDA_LOADER 60
3.2.19 PMML_SEQUENCE_LOADER 61
3.2.20 PMML_TREE_LOADER 61
3.2.21 PP_ADD_HIERARCHY 62
3.2.22 PP_CHANGE_TYPE 63
3.2.23 PP_DIVIDING_ATTRIBUTE 64
3.2.24 PP_FILTER_ATTRIBUTES 65
3.2.25 PP_FOLDING 66
3.2.26 PP_HIERARCHICAL_DISCRETIZATION 66
3.2.27 PP_MARKING 67
3.2.28 PP_MARK_DUPLICATES 69
3.2.29 PP_MERGE_DUPLICATES 70
3.2.30 PP_NEW_ATTRIBUTE 71
3.2.31 PP_NORMALIZATION 72
3.2.32 PP_NUMERIC_DISCRETIZATION 73
3.2.33 PP_NUMERIC_LABELING 73
3.2.34 PP_REMOVE_ROWS 74
3.2.35 PP_RENAME_ATTRIBUTES 75
3.2.36 PP_REWRITING 76
3.2.37 PP_SAMPLING 79
3.2.38 PP_SORTING_ATTRIBUTE 79
3.2.39 PP_TABLE_2_TABLE 80
3.2.40 PP_TABLE_LOADER 81

3.2.41 RDA_EXCEPTION 81
3.2.42 RDA_FILTER 82
3.2.43 RDA_LOADER 83
3.2.44 RDA_MINER 84
3.2.45 RDA_PRESERVED 85
3.2.46 RDA_SATISFY 86
3.2.47 SCALAR 87
3.2.48 SEQUENCE_AGGREGATE_FILTER 87
3.2.49 SEQUENCE_EXCEPTION 90
3.2.50 SEQUENCE_FILTER 90
3.2.51 SEQUENCE_LOADER 92
3.2.52 SEQUENCE_MAXIMAL_FILTER 93
3.2.53 SEQUENCE_MINER 93
3.2.54 SEQUENCE_RULE 94
3.2.55 SEQUENCE_SATISFY 96
3.2.56 SEQUENCE_TIMESTAMP_FILTER 97
3.2.57 SEQ_QUERY 99
3.2.58 TABLE_2_HIERARCHY 99
3.2.59 TABLE_2_PP_TABLE 100
3.2.60 TABLE_LOADER 101
3.2.61 TREE_CLASSIFY 102
3.2.62 TREE_LOADER 102
3.2.63 TREE_META_CLASSIFIER 103
3.2.64 TREE_MINER 104

4 KDDML algorithms **106**
4.1 Classification of algorithms 107
4.1.1 Discretization algorithms 107
4.1.2 Normalization algorithms 107
4.1.3 Rewriting algorithms 107
4.1.4 Sampling algorithms 108
4.1.5 RdA miner algorithms 109
4.1.6 Tree miner algorithms 109
4.1.7 Cluster miner algorithms 109
4.1.8 Sequence miner algorithms 110
4.2 Algorithms specification 110
4.2.1 By cluster sampling 110
4.2.2 CAGE (CellulAr GEneting programming tool) 111
4.2.3 DCI (Direct Count & Intersect) 113
4.2.4 EM (Expectation Maximization) 114
4.2.5 Equal frequency discretization 115
4.2.6 KMeans 116
4.2.7 Min-max normalization 118
4.2.8 Natural binning discretization 118

4.2.9 Prefix Span 120
4.2.10 Rule M 121
4.2.11 Rule S 122
4.2.12 Rule T 124
4.2.13 Simple sampling 125
4.2.14 Stratified sampling 127
4.2.15 YaDT (Yet Another Decision Tree builder) 128
4.2.16 Z-score normalization 129
Appendices 132

A Extending the KDDML language **132**
A.1 Adding a new algorithm 132
A.2 Adding a new operator 132
A.3 Adding a new model 134
A.4 Example: adding a naive bayes model 135
A.4.1 Physical model 135
A.4.2 Model entity 136
A.4.3 Operators 136
A.4.4 Operator one: model loader 136
A.4.5 Operator two: model miner 136
A.4.6 Operator three: classifier 137

LIST OF TABLES

2.1 The weather dataset 17
2.2 A sample transactional table 20
2.3 A sample timestamp table 21
2.4 Condition specification for the element IF_TERM of expressions 37
2.5 Correspondence between XML entities and KDDML types 38

3.1 I/O operators 40
3.2 Preprocessing operators 42
3.3 Mining operators 42
3.4 Postprocessing operators 43
3.5 Control flow operators 44
3.6 Unclassified operators 44
3.7 Element BASE_COND for IF operator 58
3.8 Element BASE_COND for PP_MARKING operator 69
3.9 Element BASE_COND for the PP_REMOVE_ROWS operator 76
3.10 Element BASE_COND for the PP_REWRITING operator 78
3.11 Element BASE_COND for the RDA_FILTER operator 84
3.12 An example of market sequence dataset 88
3.13 Element BASE_COND for the SEQUENCE_AGGREGATE_FILTER operator . . . 89
3.14 Element BASE_COND for the SEQUENCE_FILTER operator 92
3.15 An example of item hierarchy as relational table 100

4.1 Discretization algorithms 107
4.2 Normalization algorithms 107
4.3 Rewriting algorithms 108
4.4 Data sampling algorithms 108
4.5 RdA miner algorithms 109
4.6 Tree miner algorithms 109

4.7 Cluster miner algorithms . . . 109
4.8 Sequence miner algorithms . . . 110
4.9 `by_cluster_sampling` parameters. . . . 111
4.10 CAGE parameters . . . 112
4.11 DCI parameters . . . 114
4.12 EM parameters . . . 115
4.13 `equal_frequency_discretization` parameters. . . . 116
4.14 KMeans parameters . . . 117
4.15 `min_max_normalization` parameters. . . . 118
4.16 `natural_binning_discretization` parameters. . . . 120
4.17 prefix_span parameters . . . 121
4.18 `rule_m_rewriting` parameters. . . . 122
4.19 `rule_s_rewriting` parameters. . . . 124
4.20 `rule_t_rewriting` parameters. . . . 125
4.21 `simple_sampling` parameters. . . . 126
4.22 `stratified_sampling` parameters. . . . 128
4.23 YaDT parameters . . . 129

CHAPTER 1 Getting started

With the rapid computerization of businesses and organizations, a huge amount of data has been collected and stored in databases, and the rate at which data are stored is growing at a phenomenal rate. As a result, traditional ad hoc mixtures of statistical techniques and data management tools are no longer adequate for analyzing this vast collection of data. Knowledge Discovery in Databases (or KDD in short) has emerged as a growing field of multidisciplinary research for discovering interesting/useful knowledge from large databases. KDD covers a wide range of application domains (retail, marketing, finance, e-commerce, biology, privacy, only to cite a few ones), several models of representing extracted patterns and rules (including classification models, association rules, sequential patterns, clusters) and a large number of algorithms for data preprocessing, model extraction and model reasoning.

1.1 Motivations

KDD [1] is the process of finding "nuggets" of knowledge. It is a complex task, heavily dependent on the problem and on the data at hand. As described in the CRISP-DM process model [2], it may consist of several repeated phases including business problem understanding, data comprehension, data preparation, modelling (or data mining), evaluation and deployment. The development of KDD solutions requires then to specify the tasks at each phase and the interactions/dependencies among them. Most of the times, this results is a complex process, requiring to combine different sources of data and knowledge, and with many tasks iterated in order to reach a (unfortunately, local) optimum. Figure 1.1 shows an example of KDD process.

KDD technology has reached a maturity state as far as the design of efficient knowledge extraction algorithms is concerned. This is witnessed by the large number of commercial tools (including all major RDBMS) offering KDD algorithms. On the contrary, the design of final applications is still an "art", obtained by composing algorithm libraries, proprietary API's, SQL queries and stored procedure calls to RDBMS, and *much much* code.

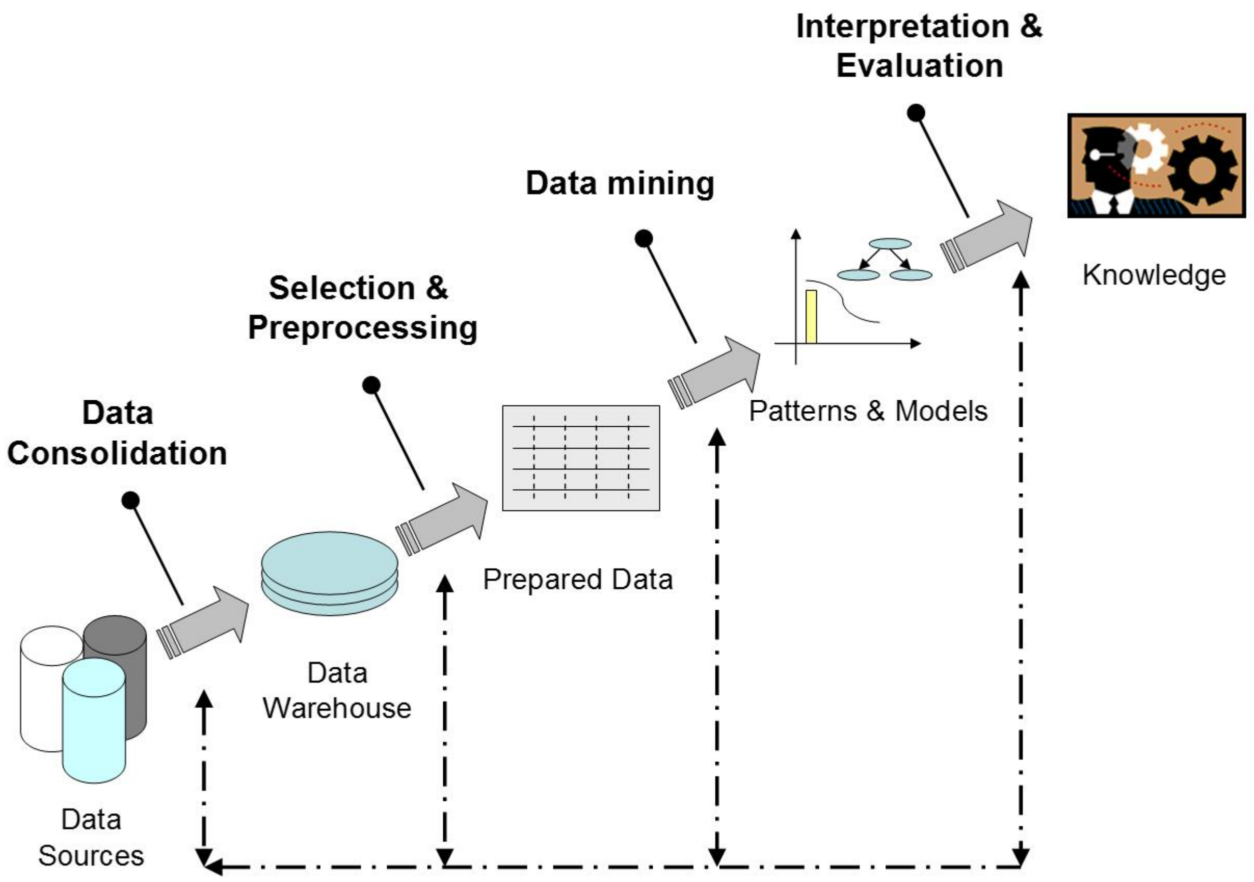

Figure 1.1: The KDD process

At present, there is a fervent activity of standardization in the area of mining model representation and access and of mining algorithms API's [3], [4]. We think that a middleware language and system is needed to support the development of final applications or higher level systems which need a mixture of database access, data preprocessing, mining extraction and deployment. XML appears as a bridge between database technology and data mining tools. However, its use seems limited to the exchange of mining models between applications. We would like to go further and conceive a language (and system) where XML is used for processing data and mining models as well. XML is largely used as a machine-processable language, e.g. in the web technology. It seems then natural to express KDD operations as XML elements.

In the rest of the document, we concentrate on the description of KDDML as KDD language whose design principles are motivated by requirements derived from recurring patterns in the KDD process.

1.2 KDDML language overview

KDDML (Knowledge Discovery in Database Mark-up Language) is a middleware language (and system) needed to support the development of final applications or higher level systems which demand a KDD integrated environment.

The KDDML language is XML-based both for query syntax and data/model representation, in order to favor *machine-processability*. However, the semantics is purely "functional", which ensures *compositionality* of operators. Compositionality is granted by a *closure principle*.

The semantics of a KDDML language expression is either a model or a data table. Therefore, we call a KDDML language expression a *KDDML query*, in order to emphasize that a result is expected. We will survey operators on data access and preprocessing, model extraction and deployment, and control flow operators. Concerning data and model representation, an XML-based approach is adopted here as well. In particular, models are represented using an extension of the Predictive Markup Modelling Language standard (PMML) [3]. About relational tables, we use a proprietary representation that is a mixture between XML and text.

The KDDML language assumes a *data repository*, containing relational and preprocessing tables, a *model repository*, containing mining models, and a *query repository*, containing queries. Tables, models and queries can be referenced by an identifier[1]. KDDML queries are XML-documents, where XML tags correspond to operations on data and/or models, XML attributes correspond to parameters of those operations and XML sub-elements define arguments passed to the operators.

1.2.1 A simple sample

As an example, the query of figure 1.2 specifies the construction and application of a decision tree. The root tag is `<KDD_QUERY>`, with the query name as an attribute.

`<TREE_CLASSIFY>` is the operator that applies a decision tree to predict the class of tuples in a test set. The attribute `xml_dest="results.xml"` states that the results of the classification are stored in the data repository for further processing or analysis.

The tree to be applied is provided by the first sub-element (with tag `<TREE_MINER>`) which specifies the construction of a classification tree.

The test set is provided by the second element (with tag `<TABLE_LOADER>`), which specifies a relational table named `testSet.xml` gathered from the local data repository.

In turn, the construction of a decision tree (tag `<TREE_MINER>`) takes place on a training set `trainingSet.arff` in ARFF format[2] by applying a decision tree induction algorithm (here, YADT from [5]) with parameters concerning the pruning strategy of the algorithm (tag `<ALGORITHM>`). The name of the class attribute is provided as attribute of the `<TREE_MINER>` element. As it will be shown later on, the KDDML language embeds a library of (pre-processing or mining) algorithms and basic mechanisms for adding new ones.

Before applying the algorithm on the training set, a preprocessing step is performed. The `<PP_REMOVE_ROWS>` operator allow us to remove all instances having a missing value for the

[1] In the actual implementation, the identifier coincides with the name of the file where they are stored.

[2] The ARFF format is a text file consisting of a list of instances with the attribute value for each instance being separated by a comma.

```
<KDD_QUERY name="sample">
   <TREE_CLASSIFY xml_dest="results.xml">
      <TREE_MINER xml_dest="tree.xml" target_attribute="class">
         <PP_TABLE_2_TABLE>
            <PP_REMOVE_ROWS>
               <TABLE_2_PP_TABLE>
                  <ARFF_LOADER arff_file_name="trainingSet.arff" />
               </TABLE_2_PP_TABLE>
               <CONDITION>
                  <BASE_COND op_type="is_missing" term1="@temperature"/>
               </CONDITION>
            </PP_REMOVE_ROWS>
         </PP_TABLE_2_TABLE>
         <ALGORITHM algorithm_name="YADT">
            <PARAM name="confidence_for_pruning" value="0.4"/>
            <PARAM name="num_instances_for_leaf" value="2"/>
          </ALGORITHM>
       </TREE_MINER>
      <TABLE_LOADER xml_source="testSet.xml"/>
   </TREE_CLASSIFY>
</KDD_QUERY>
```

Figure 1.2: A sample KDDML query.

attribute `temperature` (tag `<CONDITION>`). Finally, the operators `<TABLE_2_PP_TABLE>` and `<PP_TABLE_2_TABLE>` allow us, respectively, to start and to finalize the preprocessing step on `trainingSet.arff`.

1.2.2 KDDML as typed language

As one could expect, arguments of an operator must be of an appropriate type and sequence, i.e. an operator *signature* must be specified. We denote the signature of an operator $f : t_1 \times \ldots t_n \rightarrow t$ returning type t by defining a DTD for KDDML queries that constraints sub-elements to be of type $t_1, \ldots, t_n$. Thus, KDDML queries corresponds to terms in the algebra of operators, though syntactically represented as XML documents.

Intuitively, there is one type for data sources, one type for preprocessing tables, one type for each mining model (classification tree, association rules, sequential patterns, clusters) and one type for hierarchies. Other proprietary objects denote special arguments such as algorithms definition or conditions on table attributes. As shown in the next chapter, we call the root tag of language objects as `KDDMLObject`.

Under this interpretation, the semantics of a KDDML query amounts to a strict functional execution of the corresponding term. The evaluation of an XML-fragment:

```
   <OPERATOR_NAME xml_dest="results.xml" att1="v1" ... attM="vM">
      <ARG1_NAME> .... </ARG1_NAME>
      ...
      <ARGn_NAME> .... </ARGn_NAME>
   </OPERATOR_NAME>
```

consists of:

1. evaluation of attributes `att1 ... attM` returning a set of scalar values;
2. recursive evaluation of fragments from `<ARG1_NAME> ... </ARG1_NAME>` to `<ARGn-_NAME> .... </ARGn_NAME>`; this evaluation returns a set of `KDDMLObject`.
3. a call to an operator $f_{\texttt{OPERATOR\_NAME}}$, accepting results from (1) and (2) and yielding the final result of the fragment; also the final result is a `KDDMLObject`.

Moreover, a copy of the final result (which may be an intermediate result of a possibly larger query) is stored in the (model or data) repository if the attribute `xml_dest` is specified. Notice that repositories are persistent, so to favor the reuse of extracted knowledge and preprocessed data.

As a by-product, the language satisfies a *closure principle*, namely that any operator returning type t can be used wherever an argument of type t is required. Also, validation of queries as XML documents against the DTD corresponds to static type-checking of operators in the query. As an example, this fragment of the DTD:

```
<!ELEMENT TREE_CLASSIFY ((%kdd_query_trees;),(%kdd_query_table;))>
<!ATTLIST TREE_CLASSIFY xml_dest CDATA #IMPLIED>
```

requires that the first sub-element of `<TREE_CLASSIFY>` be one of those in the entity `kdd-_query_trees` (i.e. all operators returning a tree model) and the second one is in the entity `kdd_query_table` (i.e. all operators returning a table). In other terms, the `TREE_CLASSIFY` operator is a function $f_{<\texttt{TREE\_CLASSIFY}>} : \texttt{tree} \times \texttt{table} \rightarrow \texttt{table}$. The DTD is then another (simple and general) way of specifying an algebra of types and operators.

1.3 Organization of this guide

This document focuses on the language KDDML: concepts, DTD specifications and algorithms definitions. Background knowledge concerns XML and the related Document Type Definition [6].

In section 2 we provide a complete description of the objects (i.e. types) composing the KDDML language: as previously reported, a DTD specification has been provided in order to define regular KDDML objects;

in sections 3 and 4 we report, respectively, the operators (such as `TREE_CLASSIFY`) and algorithms (such as `YaDT`) language specification;

finally, in appendix A we explain how the DTD's of the language have to be extended in order to introduce new algorithms, operators or models.

CHAPTER 2

KDDML Objects

KDDML is a middleware mark-up language that allows one to represent models, tables and KDD queries in a uniform way. The aim is to develop an environment in which several kinds of knowledge extraction operations can be combined, in order to describe and solve complex knowledge extraction problems. As shown in the previous section, KDDML adopts the emerging XML standard as a glue for query definition and data/model representation. In this section, we concentrate on the description of the KDDML objects that represent the core of the KDDML language.

2.1 Objects Hierarchy

When addressing the problem of defining a common representation for knowledge extraction problems and their results, we followed the viewpoint of Imielinsky and Mannila [7], who define two classes of fundamental objects: the *KDD object* (here `KDDMLObject`) that is a result of a KDD step such as a table or a set of association rules, and the *KDD query* that is a predicate which returns a result that is a KDD object.

The overall hierarchy of `KDDMLObject` is reported in figure 2.1. Note that there is a subtype of `KDDMLObject` for each possible result returned by an operator.

The set of types of KDDML operators consists of:

- `PMML` represents a PMML model, i.e, a pattern returned by a DM algorithms or a post-processing operator. In the actual implementation, we have considered association rules (`rda`), clusters (`cluster`), classification trees (`tree`), sequential patterns (`sequences`) and item hierarchies (`hierarchy`).

- `KDDML_TABLE` represents both a relational table (`table`), such as the records of a database relation or text file, and a preprocessing table (`PPtable`), obtained by applying a preprocessing operator, such as a sampling or a normalization operator. Both relational and preprocessing tables are in a proprietary format described later.

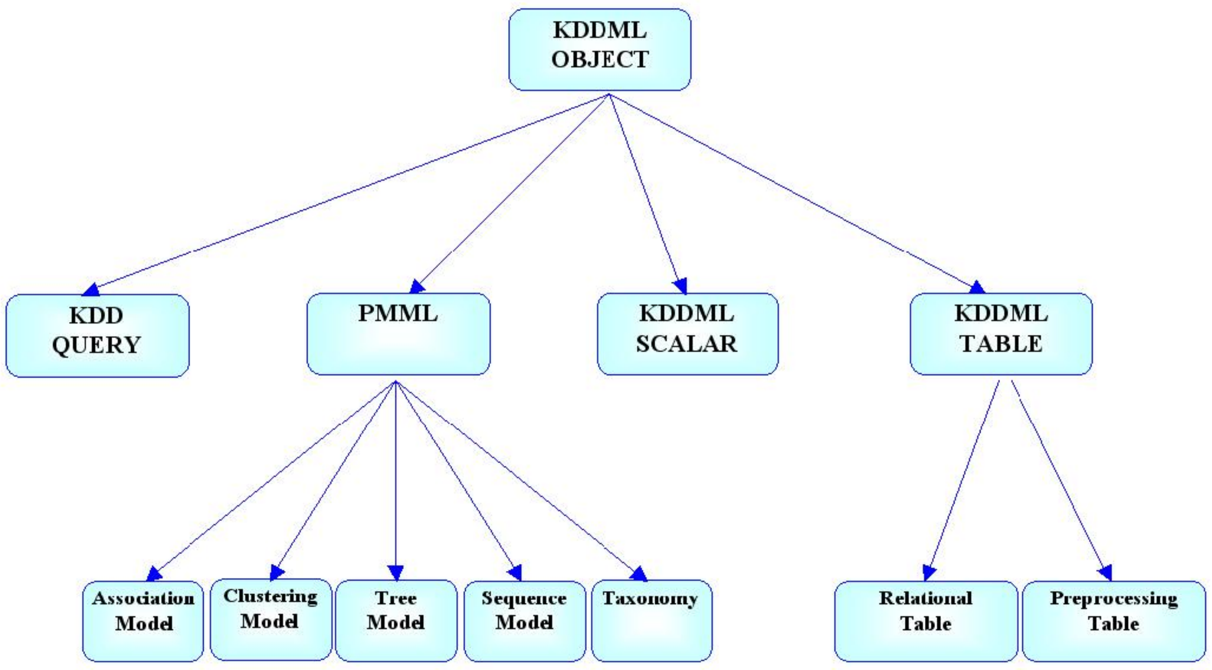

Figure 2.1: Object hierarchy in the KDDML language

- `KDDML Scalar` represents a generic scalar value (`scalar`), such as a number or a string.
- `KDDQuery` represents a KDD query, i.e. a composition of invocations to DM and preprocessing algorithms by means of appropriate operators. Queries can use a conditional or sequence operator, and they can be nested.

Other special types (not shown in figure) are used to define algorithm settings (`algorithm`), and condition specifications (`condition`) on constant and/or table attribute values. Finally, a type `expression` is defined to support the use of mathematical operations.

Notice that the use of KDDML makes it possible to represent not only models or KDD queries, bus also objects such as tables or algorithms, in order to allow the construction of complex KDD queries that may cross the border between tuples and models several times possibly by using multiple layers of nesting. In the next sections, we describe KDDML objects more in detail.

2.2 Data representation

The KDDML language refers to two data repositories, containing relational tables and preprocessing tables in different spacenames.

A relational table is represented as an XML file, containing a *schema* and a reference to the actual data, which are stored in CSV (Comma Separated Values) format. In principle, however, the coding of actual data can have any format: CSV has been chosen here as a trade-off between readability (vs binary files) and space occupancy (vs XML).

A preprocessing table (PP table in short) is used in the preprocessing step of the KDD process. With respect to the relational table, a PP table is composed by the data schema, the actual data and, in addition, by preprocessing information such as marks associated to the physical instance values. In the actual implementation, also the preprocessing information is stored in CSV format, but other kind of representations can be available in future.

The `KDDML_TABLE` element represents both relational tables and PP tables and his DTD is reported in figure 2.2:

```
<!ELEMENT KDDML_OBJECT (KDDML_TABLE)>
<!ELEMENT KDDML_TABLE(SCHEMA,(PPSCHEMA)?)>
<!ATTLIST KDDML_TABLE data_file CDATA #REQUIRED>
<!ATTLIST KDDML_TABLE pp_data_file CDATA #IMPLIED>
```

Figure 2.2: The `KDDML_TABLE` element.

Summarizing, a `KDDML_TABLE` is always composed by:

1. a logical schema (element `SCHEMA`) that includes attribute types and some simple statistic on attributes values;
2. the physical instances referred by using the attribute `data_file` containing the name of the CSV file that is stored in the data repository.

A preprocessing table is similar to a relational table, but, in addition, it uses the element `PPSCHEMA` and the attribute `pp_data_file` in order to localize preprocessing description of data (see later). Notice that both the element `PPSCHEMA` and the attribute `pp_data_file` are not required; this distinguishes relational tables from preprocessing tables.

2.2.1 Logical data

Data connectivity standards offer APIs for connecting to a data source, for issuing SQL queries, for navigating returned record sets, and for accessing database and record set meta-data. However, this level of APIs can be considered as a physical level. A higher abstraction level concerns logical data, i.e. domains of data to be used as input to data mining operations in order to specify the type of usage of attributes in building and applying a mining model.

The element `SCHEMA` specifies metadata information, that is necessary to obtain some kind of information about each attribute, which cannot be automatically derived from the attribute values. The statistics for a table are made of the collection of the statistics for the single fields.

The figure 2.3 shows the DTD related to the element `SCHEMA`. This element is composed by one or more elements `ATTRIBUTE`, each of which specifies the name, the type and the statistics about a particular field of the table. The name of a data field must be unique in the data schema, and the order the attributes corresponds to the column position in the data section of the physical CSV file. For example, if an attribute is declared as the third one, then KDDML expects

```
<!ELEMENT SCHEMA (ATTRIBUTE+)>
<!ATTLIST SCHEMA logical_name CDATA #REQUIRED>
<!ATTLIST SCHEMA number_of_attributes CDATA #REQUIRED>
<!ATTLIST SCHEMA number_of_instances CDATA #REQUIRED>
<!ELEMENT ATTRIBUTE (STRING_DESCRIPTION |
                     NOMINAL_DESCRIPTION |
                     NUMERIC_DESCRIPTION), TAXONOMY?>
<!ATTLIST ATTRIBUTE name CDATA #REQUIRED>
<!ATTLIST ATTRIBUTE type (numeric|nominal|string) #REQUIRED>
<!ATTLIST ATTRIBUTE number_of_missed_values CDATA #REQUIRED>
<!ATTLIST ATTRIBUTE number_of_missed_values_perc CDATA #REQUIRED>
```

Figure 2.3: The `SCHEMA` element.

that all values of that attribute will be found in the third comma delimited column. The attribute `logical_name` contains the logical name of the table. Other attributes contain the number of columns (`number_of_attributes`) and the number of instances (`number_of_instances`) belonging to the table.

The name (resp. type) of the attribute is expressed by using the attribute `name` (resp. `type`) in the `ATTRIBUTE` element. The datatype can be any of the three types currently supported by KDDML:

- numeric (both integer and real),
- discrete (binary, nominal or categorical),
- string.

Statistics on attribute values depend on the type of the attribute, as expressed by means the elements `NUMERIC_DESCRIPTION`, `NOMINAL_DESCRIPTION` and `STRING_DESCRIPTION`. The `number_of_missed_values` (resp. `number_of_missed_values_perc`) attribute contains the absolute (resp. percentage) number of values that are missing for that attribute with respect to the total number of instances.

Finally, the element `TAXONOMY` defines a new logical level, and it allows us to assign an item hierarchy to a table column as meta-data information (see section 2.2.2).

Discrete attributes

Discrete values are defined by providing a nominal specification listing the possible values belonging to a set of one or more elements, as reported in the DTD of figure 2.4. As shown in the figure, the element NOMINAL_DESCRIPTION is composed by one or more elements VALUE, each of them representing the category of the nominal attribute. The `number_of_values` attribute counts the number of distinct categories belonging to the attribute. For each category, the `value` attribute contains the name of the category, while the `cardinality` (resp. `cardinality_perc`) attribute provides the absolute (resp. percentage) number of instances with value equals to the category name, with respect to the total number of instances without missing values for that attribute.

```
<!ELEMENT NOMINAL_DESCRIPTION (VALUE)+>
<!ATTLIST NOMINAL_DESCRIPTION number_of_values CDATA #REQUIRED>
<!ELEMENT VALUE EMPTY>
<!ATTLIST VALUE value CDATA #REQUIRED>
<!ATTLIST VALUE cardinality CDATA #REQUIRED>
<!ATTLIST VALUE cardinality_perc CDATA #REQUIRED>
```

Figure 2.4: The `NOMINAL_DESCRIPTION` element.

Numeric attributes

As to numeric attributes (see figure 2.5), the element contains the mean, the standard deviation, the sum, the squared sum, the min and the max values defined as usual.

```
<!ELEMENT NUMERIC_DESCRIPTION EMPTY>
<!ATTLIST NUMERIC_DESCRIPTION mean CDATA #REQUIRED>
<!ATTLIST NUMERIC_DESCRIPTION std_dev CDATA #REQUIRED>
<!ATTLIST NUMERIC_DESCRIPTION sum CDATA #REQUIRED>
<!ATTLIST NUMERIC_DESCRIPTION sumSq CDATA #REQUIRED>
<!ATTLIST NUMERIC_DESCRIPTION min CDATA #REQUIRED>
<!ATTLIST NUMERIC_DESCRIPTION max CDATA #REQUIRED>
```

Figure 2.5: The `NUMERIC_DESCRIPTION` element.

String attributes

String attributes (see figure 2.6) do not contain further features.

```
<!ELEMENT STRING_DESCRIPTION EMPTY>
```

Figure 2.6: The `STRING_DESCRIPTION` element.

As an example, in figure 2.7 the XML document describing the `weather` data set of table 2.1 is reported.

2.2.2 Taxonomies

A taxonomy represents hierarchical relationships between categories. Generally, the topmost categories are most general, and the leaves are most specific or referring to specific item categories.

In KDDML taxonomies can exist both as explicit relationships between categories represented as a mining model (see figure 2.1), and as logical data element related to a non-numeric attribute. The element `TAXONOMY` of figure 2.8 models the last one case.

```
<KDDML_TABLE data_file="weather.csv">
   <SCHEMA logical_name="weather" number_of_attributes="4"
           number_of_instances="12">
      <ATTRIBUTE name="outlook" number_of_missed_values="2"
                 number_of_missed_values_perc="17%" type="nominal">
         <NOMINAL_DESCRIPTION number_of_values="3">
            <VALUE value="rainy" cardinality="3" cardinality_perc="30%"/>
            <VALUE value="overcast" cardinality="4" cardinality_perc="40%"/>
            <VALUE value="sunny" cardinality="3" cardinality_perc="30%"/>
         </NOMINAL_DESCRIPTION>
      </ATTRIBUTE>
      <ATTRIBUTE name="temperature" number_of_missed_values="0"
                 number_of_missed_values_perc="0%" type="numeric">
         <NUMERIC_DESCRIPTION mean="75.08" variance="47.35" sum="901.0"
                              sumSq="68171.0" min="64.0" max="85.0"/>
      </ATTRIBUTE>
      <ATTRIBUTE name="humidity" number_of_missed_values="2"
                 number_of_missed_values_perc="17%" type="numeric">
         <NUMERIC_DESCRIPTION mean="78.5" variance="80.5" sum="785.0"
                              sumSq="62347.0" min="65.0" max="90.0"/>
      </ATTRIBUTE>
      <ATTRIBUTE name="play" number_of_missed_values="0"
                 number_of_missed_values_perc="0%" type="nominal">
         <NOMINAL_DESCRIPTION number_of_values="2">
            <VALUE value="yes" cardinality="8" cardinality_perc="67%"/>
            <VALUE value="no" cardinality="4" cardinality_perc="33%"/>
         </NOMINAL_DESCRIPTION>
      </ATTRIBUTE>
   </SCHEMA>
</KDDML_TABLE>
```

Figure 2.7: The logical schema of the weather dataset.

outlook	**temperature**	**humidity**	**play**
sunny	85	85	no
sunny	80	90	no
overcast	83	86	yes
NULL	65	70	no
overcast	64	65	yes
sunny	72	95	no
sunny	69	*NULL*	yes
rainy	75	80	yes
sunny	75	70	yes
overcast	72	90	yes
overcast	81	75	yes
rainy	71	*NULL*	no
NULL	80	74	yes

Table 2.1: The weather dataset

The name of the hierarchy is expressed by using the attribute `name` in the `TAXONOMY` element. As shown, a taxonomy is created from a sequence of one or more parent/child tables (element `CHILD_PARENT`) with associated some attributes:

```
<!ELEMENT TAXONOMY (CHILD_PARENT+) >
<!ATTLIST TAXONOMY name CDATA #REQUIRED >
<!ELEMENT CHILD_PARENT (INLINE_TABLE)>
<!ATTLIST CHILD_PARENT child_field CDATA #REQUIRED
                       parent_field CDATA #REQUIRED
                       parent_level_field CDATA #IMPLIED
                       is_recursive (no | yes) "no"
                       root_name CDATA #REQUIRED>
<!ELEMENT INLINE_TABLE (ROW*) >
<!ELEMENT ROW EMPTY>
<!ATTLIST ROW member CDATA #REQUIRED>
<!ATTLIST ROW group CDATA #REQUIRED>
```

Figure 2.8: The TAXONOMY element.

- child_field defines the name of the field which contains the child value for each record in the INLINE_TABLE element.
- parent_field defines the name of the field which contains the parent value for each record in the INLINE_TABLE element.
- root_name contains the root hierarchy name.
- is_recursive is “yes” if a value in the parent field can also be used in a child field.

The tabular data is part of the XML document itself by using the element INLINE_TABLE that includes one or more rows (element ROW), each of which defines the parent/child relationship by means of the attributes group and member respectively.

As an example, the XML fragment of figure 2.9 describes how to assign the hierarchy *cities-states-countries* of figure 2.10 to the string attribute city as metadata information (see also the PP_ADD_HIERARCHY operator in sect. 3.2.21).

2.2.3 Physical data: relational tables, transactional tables and timestamp tables

Physical data are represented in KDDML as text files in Comma Separated Value (CSV), as it is used in Microsoft Excel. In a CSV file, each record takes one line, and each field is separated by a comma. Leading and trailing space-characters adjacent to comma field separators are ignored. By convention, missing or null values are represented with the symbol ''?''. As an example, the figure 2.11 illustrates the weather dataset of table 2.1 in a CSV format.

Concerning data format, in KDDML, physical data can occur in three different forms:

1. *relational table*,
2. *transactional table*,

```
...
<ATTRIBUTE name="city" number_of_missed_values="0"
           number_of_missed_values_perc="0%" type="string">
   <STRING_DESCRIPTION/>
   <TAXONOMY name="cities-states-countries">
      <CHILD_PARENT child_field="member" parent_field="group"
                    is_recursive="yes" root_name="USA">
         <INLINE_TABLE>
            <ROW member="California" group="USA"/>
            <ROW member="Illinois" group="USA"/>
            <ROW member="Chicago" group="Illinois"/>
            <ROW member="Long Beach" group="California"/>
            <ROW member="San Jose" group="California"/>
         </INLINE_TABLE>
      </CHILD_PARENT>
   </TAXONOMY>
</ATTRIBUTE>
```

Figure 2.9: The hierarchy cities-states-countries as metadata information.

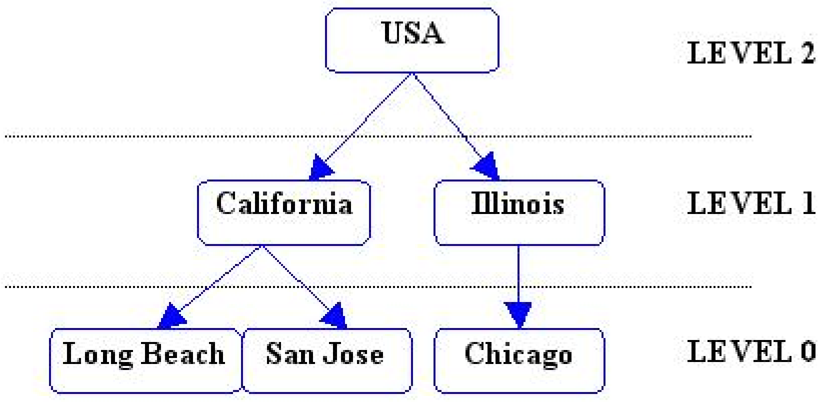

Figure 2.10: The hierarchy cities-states-countries.

3. and *timestamp table*.

The format of KDDML tables is automatically recognized by the system when loaded from the data repository or from an external resource.

Relational format

In the *relational format*, each column of the data corresponds to a logical attribute, e.g., `temperature, play`. Each row of the data corresponds to an individual case (transaction) to be considered during mining. This data format is also known as *single-record case table*. Figure 2.11

```
sunny,85,85,no
sunny,80,90,no
overcast,83,86,yes
rainy,70,?,yes
?,65,70,no
overcast,64,65,yes
rainy,75,80,yes
sunny,75,70,yes
overcast,72,90,yes
overcast,81,75,yes
rainy,71,?,no
?,80,74,yes
```

Figure 2.11: weather.csv

illustrates a typical example of relational table.

Transactional format

Sparse data is more effectively stored in a *transactional format*. Here, data that have a variable number of entries (or items) out of many possible ones can be stored more compactly, since only the actually present items are stored in the table. A transactional table (also known as *multi-record case table*) has an attribute `transaction` identifying the transaction and an attribute `event` containing the single item. Transactions are ordered with respect to the attribute `transaction`. Other columns are allowed in the table (e.g., the price or quantity of items for each transaction), but they can be ignored by the operator, depending on the context. Missing values are not allowed in `transaction` field and `event` field. This representation is typically used for association rules. Table 2.2 illustrates a typical example in the market basket analysis field. The order, in which attributes `transaction` and `event` occur, does not matter; attributes `quantity` and `price` are optional.

Transaction	**Event**	**Quantity**	**Price**
id_1	milk	2	0.75
id_1	water	10	3.20
id_1	bread	2	0.60
id_2	water	5	2.10
id_2	wine	1	5.50
id_3	potatoes	4	6.60
id_3	milk	2	0.75
id_4	bread	4	1.20

Table 2.2: A sample transactional table

Timestamp format

The `timestamp format` is similar to a transactional table, but with an extra attribute `timestamp`, that does not admit missing values. Typically, this format is used for sequential pattern analysis, since the attribute `timestamp` defines a partial time order between transactions and items. The semantics of attributes depend on the context. As instance, in the web log analysis (resp. medical record analysis), the `transaction` attribute can coincide with the user identifier (resp. name of the patient), the `timestamp` attribute coincides with the time of the visit (resp. day of control) and, finally, the `event` attribute coincides with the web page visited (resp. symptom). Table 2.3 shows the same data of table 2.2, but with the `timestamp` attribute in addition. As for transactional tables, the ordering in which attributes `transaction`, `timestamp` and `event` occur does not matter; attributes `quantity` and `price` are optional.

Transaction	**Timestamp**	**Event**	**Quantity**	**Price**
id_1	monday	milk	2	0.75
id_1	saturday	water	10	3.20
id_1	saturday	bread	2	0.60
id_2	monday	water	5	2.10
id_2	monday	wine	1	5.50
id_3	tuesday	potatoes	4	6.60
id_3	wednesday	milk	2	0.75
id_4	saturday	bread	4	1.20

Table 2.3: A sample timestamp table

2.2.4 Preprocessing tables

This object is used to represent tables as used in the preprocessing step of the KDD process. A PPtable is composed by:

- the *data schema*, that includes attribute types and some simple statistics on attribute values;
- the *physical data section* as a text file in a CSV format;
- the *preprocessing data section*, including preprocessing information such as *marks* associated to a physical instance value belonging to the data section. Also the preprocessing section is in a CSV format, with the number of columns and rows coinciding with the number of attributes and rows of the data section. By convention, all instances values of preprocessing section are set to a missing value when the preprocessing phase starts;
- the *preprocessing history* used to list the set of preprocessing operations performed on the table.

As a consequence, a preprocessing table defines both the `SCHEMA` element of figure 2.3 and the `PPSCHEMA` element of figure 2.12. The last one contains the logical description of preprocessing attributes[1] and the preprocessing history (element `HISTORY`) related to the table.

```
<!ELEMENT PPSCHEMA ((ATTRIBUTE+), HISTORY)>
<!ATTLIST PPSCHEMA logical_name CDATA #REQUIRED>
<!ATTLIST PPSCHEMA number_of_attributes CDATA #REQUIRED>
<!ATTLIST PPSCHEMA number_of_instances CDATA #REQUIRED>
```

Figure 2.12: The `PPSCHEMA` element.

Preprocessing section

Preprocessing information concern *marks* and *exceptions*.

Marks contains some information related to physical instance values. This information is added in the preprocessing section of a PPtable, in correspondence to the related attribute value of a physical record. As an instance consider the `weather` dataset of table 2.1. We can decide to mark all instances in which the attribute `temperature` is not in the interval $[70, 80]$. The result obtained is shown in figure 2.13, in which the preprocessing section of the weather dataset is reported. This information can be processed later, for example, in order to filter the instances out of range, or to rewrite their values with a new temperature value, according to a specific mathematical function. KDDML supports more than one mark for each instance value. Each mark is separated by a semicolon symbol. Marks are always explicitly added by the user by using a well-defined operator.

```
?,out_of_range;,?,?
?,?,?,?
?,out_of_range;,?,?
?,?,?,?
?,?,?,?
?,out_of_range;,?,?
?,?,?,?
?,?,?,?
?,?,?,?
?,out_of_range;,?,?
?,?,?,?
?,?,?,?
```

Figure 2.13: weather_metadata.csv

[1]The number of preprocessing attributes coincides with the number of attributes belonging to the data section. The preprocessing attribute name coincides with the attribute name of the data section followed by the string "_`metadata`"; its type is string.

Exception are similar to marks, but they are automatically added by the system when a particular event occurs. A typical exception can be generated when we try to match an attribute value against a pattern using a regular expression, but the patterns matching is not satisfied. If this is the case, then the error can be raised and its description can be added in the preprocessing section of the PPtable as an exception, in correspondence to the instance in which the error occurs. With respect to marks, an exception starts with the tag *"Exception"*.

Preprocessing history

As shown in figure 2.14, the element `HISTORY` is composed by one or more `PREPROCESSING_TASK` elements, each of them containing information on preprocessing operations performed on the PPtable. More precisely, the attribute `operator_name` contains the name of preprocessing operator used, while the `description` attribute contains the list of parameter used by the operator. An example of preprocessing history is reported in figure 2.15.

```
<!ELEMENT HISTORY (PREPROCESSING_TASK+)>
<!ELEMENT PREPROCESSING_TASK EMPTY>
<!ATTLIST PREPROCESSING_TASK operation_name CDATA #REQUIRED>
<!ATTLIST PREPROCESSING_TASK description CDATA #REQUIRED>
```

Figure 2.14: The `HISTORY` element.

```
<HISTORY>
   <PREPROCESSING_TASK operation_name="TABLE_2_PPTABLE"
                       description="Start pre-processing"/>
   <PREPROCESSING_TASK operation_name="PP_MARKING"
        description="Marked attribute temperature if temperature is not
                     in [70, 80] with label out_of_range"/> </HISTORY>
```

Figure 2.15: A sample `HISTORY` element.

2.3 Knowledge representation

As for data, the KDDML language uses a *model repository*, containing extracted data mining models, which can be referenced by an identifier (in a different namespace for each model). Model entities are defined to represent DM models such as association rules, clusters, classification trees, sequential patterns and item hierarchies. KDDML represents models as an extension of *PMML (Predictive Model Markup Language)* and currently it uses the PMML 2.0 version [3].

PMML is an industry standard for representating models as XML documents. It consists of DTDs for a wide spectrum of models, and it is used for describing the structure and intents of the data mining models. PMML helps in defining semantically expressive data mining models from which different predictive models can be built. While PMML is becoming a primary standard, adopted by major commercial suites, it is worth noting that it does not cover the process of extracting models, but rather the exchange of the extracted knowledge.

Each PMML model is always composed by three basic elements[2]:

- a *data dictionary* containing the definitions for the fields that are used in the mining model, such as the type and the value range. These definitions are assumed to be independent from specific data sets, used for training or scoring a specific model;
- a *mining schema* that lists fields used by the model (i.e., it lists the fields which a user must provide in order to apply the model). These fields are a subset of the fields in the data dictionary. In other terms, the mining schema contains information that is specific to a certain model, while the data dictionary contains data definitions that do not vary with the model. For instance, the mining schema specifies the usage type of an attribute (i.e. active, predicted or supplementary). Data dictionary and mining schema define the logical level of a model;
- a *model description* containing the physical model and varying from a model to another.

An example of PMML model containing a set of association rules is reported in figure 2.16.

The PMML DTD contains a mechanism for extending the contents of a model. Extension elements are included in the contents definition of many element types. These extension elements have ANY as the contents model to allow considerable freedom in the nature of the extensions. As mentioned later, KDDML uses the extension mechanism in some cases.

2.3.1 Association model

Description

Association rule mining finds interesting associations or correlation relationships among a large set of data items. A typical example of association rule mining is market basket analysis, that analyzes customer buying habits by finding associations among the different items that customers place in their shopping baskets.

KDDML association model deals with two types of association rules:

1. *inter-attribute association rules* than have the form *"outlook=sunny AND windy=false* $\rightarrow$ *play=yes"*. It is the association among a set of attributes in a flat relation.
2. *intra-attribute association rules* such as *"spaghetti AND tomato* $\rightarrow$ *parmesan"*; also known as *boolean association rules*.

The model reported in figure 2.16 contains inter-attribute association rules. As we can notice, the attribute `value` of the element `Item` allows us to distinguish the two types of association rules.

[2]PMML v. 2.0 uses the element `TransformationDictionary` in order to map user data to values that are easier to use in the specific model. The `TransformationDictionary` element is a bridge between the `MiningSchema` and the `DataDictionary` elements. In the current version, KDDML do not support the transformation dictionary of PMML.

```
<?xml version="1.0" encoding="UTF-8"?> <PMML version="2.1">
   <Header copyright="Copyright (c) 2004 - Universita di Pisa, Dipartimento di Informatica.">
      <Application name="KDDML (Knowledge Discovery in Databases Markup Language)" version="2.0"/>
   </Header>
   <DataDictionary numberOfFields="5">
      <DataField name="outlook" optype="categorical">
         <Value value="rainy"/>
         <Value value="overcast"/>
         <Value value="sunny"/>
      </DataField>
      <DataField name="temperature" optype="continuous"/>
      <DataField name="humidity" optype="continuous"/>
      <DataField name="windy" optype="categorical">
         <Value value="TRUE"/>
         <Value value="FALSE"/>
      </DataField>
      <DataField name="play" optype="categorical">
         <Value value="yes"/>
         <Value value="no"/>
      </DataField>
   </DataDictionary>
   <AssociationModel functionName="associationRules" algorithmName="DCI - ISTI-CNR, Pisa, Italy"
                     modelName="weather_rda" minimumSupport="0.3" minimumConfidence="0.3"
                     numberOfTransactions="20" numberOfItems="4" numberOfItemsets="5"
                     numberOfRules="2" maxNumberOfItemsPerTA="3" avgNumberOfItemsPerTA="2.6">
      <MiningSchema>
         <MiningField name="outlook" usageType="active"/>
         <MiningField name="temperature" usageType="supplementary"/>
         <MiningField name="humidity" usageType="supplementary"/>
         <MiningField name="windy" usageType="active"/>
         <MiningField name="play" usageType="active"/>
      </MiningSchema>
      <Item id="1" value="outlook=sunny"/>
      <Item id="2" value="play=no"/>
      <Item id="3" value="play=yes"/>
      <Item id="4" value="windy=FALSE"/>
      <Itemset id="1" numberOfItems="1" support="0.6">
         <ItemRef itemRef="3"/>
      </Itemset>
      <Itemset id="2" numberOfItems="1" support="0.5">
         <ItemRef itemRef="4"/>
      </Itemset>
      <Itemset id="3" numberOfItems="1" support="0.4">
         <ItemRef itemRef="1"/>
      </Itemset>
      <Itemset id="4" numberOfItems="1" support="0.4">
         <ItemRef itemRef="2"/>
      </Itemset>
      <Itemset id="5" numberOfItems="2" support="0.4">
         <ItemRef itemRef="3"/>
         <ItemRef itemRef="4"/>
      </Itemset>
      <AssociationRule support="0.4" confidence="0.8" antecedent="2" consequent="1"/>
      <AssociationRule support="0.4" confidence="0.666667" antecedent="1" consequent="2"/>
   </AssociationModel>
</PMML>
```

Figure 2.16: A sample of PMML document

Compatibility with PMML v. 2.0

KDDML offers a full compatibility towards PMML v. 2.0 association model.

Extension mechanism

Not used.

2.3.2 Sequence model

Description

The problem of discovering sequential patterns is to find inter-transaction patterns such that the presence of a set of items is followed by another item in the time-stamp ordered transaction set. For instance, in a web server transaction logs, a visit by a customer is recorded over a period of time. The time stamp associated with a transaction in this case will be a time interval which is determined and attached to the transaction during the data cleaning or transaction identification processes.

Compatibility with PMML v. 2.0

In its current version, the sequence model supported by KDDML is a subset of the PMML sequence model. The PMML elements that KDDML do not support are:

- `SetPredicate` that is a set of predicates made up of simple boolean expressions;
- `Delimiter` (partially) that is the separation between two sets in a `Sequence`, or between two sequences in a `SequenceRule`. In this case, KDDML supports only the element with attributes `delimiter` equal to `acrossTimeWindow` and `gap` equal to `unknown`.

Extension mechanism

Not used.

2.3.3 Tree model

Description

A tree model in data mining is used to predict the class of an observation with unknown categorical label. A classification tree is a flow-chart-like structure, where each internal node denotes a test on an attribute, each branch represents an outcome of the test, and the leaf nodes represent classes or class distributions. The top-most node in a tree is the root node.

In order to classify an unknown sample, the attribute values of the sample are tested against the decision tree. A path is traced from the root to a leaf node that holds the class prediction for that sample. Given a pre-determined set of classes in the target attribute, classification analyzes the build data to determine to which class a given observation belongs.

A decision tree is a classification tree in which the target attribute is binary. A tree model consists of a reference to the `Node` root. Each node holds a logical predicate expression that defines the rule for choosing the node or any of the branching nodes.

Compatibility with PMML v. 2.0

KDDML offers a full compatibility towards PMML v. 2.0 tree model.

Extension mechanism

Extension mechanism has been used in two cases.

In the first one, it has been used in order to add the notion of confusion matrix to a decision tree model. A confusion matrix is a two-dimensional table that reports the number of times a case with actual class c is predicted by the classification model as having class p, where c and p range over all class values. As described by the DTD of figure 2.17, a confusion matrix can be related both to the *training set* (element `X-ConfusionMatrixTraining`), used to build the model, and to the *test set* (element `X-ConfusionMatrixTest`), used to test the model. The definition is similar in both cases.

```
<!ELEMENT X-ConfusionMatrixTraining (Array, Matrix)>
<!ATTLIST X-ConfusionMatrixTraining x-incorrectlyInst CDATA #REQUIRED>
<!ATTLIST X-ConfusionMatrixTraining x-incorrectlyInstPerc CDATA #IMPLIED>
<!ATTLIST X-ConfusionMatrixTraining x-totalInst CDATA #REQUIRED>
<!ELEMENT X-ConfusionMatrixTest (Array, Matrix)>
<!ATTLIST X-ConfusionMatrixTest x-incorrectlyInst CDATA #REQUIRED>
<!ATTLIST X-ConfusionMatrixTest x-incorrectlyInstPerc CDATA #IMPLIED>
<!ATTLIST X-ConfusionMatrixTest x-totalInst CDATA #REQUIRED>
```

Figure 2.17: The DTD representing a confusion matrix.

A confusion matrix is composed by an `Array` element containing the list of class values and a `Matrix` element containing the number of correct and incorrect predictions in which the row and column indexes refer to the classes of the target attribute[3]. The required attribute `x-totalInst` counts the total number of cases in the training or test set. The `x-incorrectlyInst` (resp `x-incorrectlyInstPerc`) attribute counts the absolute (resp. percentage) number of incorrectly classified cases with respect to the total number of cases.

In the example reported in figure 2.18, a confusion matrix for a class with values `yes` and `no` is reported. As we can notice, the accuracy of the model in the training set is 50%.

The second extension concerns meta-classifiers. We allow for classification models that exploit predictions of two or more decision trees. A classic example concerns voting trees, which are intended to overcome the bias due to the random selection of the training set or due to the choice of specific algorithms and parameters. For instance, given n distinct classifier $c_1, \ldots, c_n$, a *voting classifier* assigns to a tuple the class mostly assigned by $c_1, \ldots, c_n$.

To represent a voting classifier, we augment PMML with the `X-VotingTree` tag, as reported in figure 2.19. The `combination_type` attribute contains the combination procedure that has been used. In the current implementation, KDDML supports three voting strategies: *committe, and, or*. We refer to the `TREE_META_CLASSIFIER` operator in section 3.2.63 for details. The attributes `positive_class` and `negative_class` contains, respectively, the positive (e.g.

[3]For the definition of the elements `Array` and `Matrix` see PMML.

```
...
<TreeModel ...>
   <MiningSchema>
   ...
   </MiningSchema>
   <Node>
   ...
   </Node>
   <Extension>
      <X-ConfusionMatrixTraining x-incorrectlyInst="200"
                                 x-incorrectlyInstPerc="50%"
                                 x-totalInst="400">
         <Array n="2" type="string">"yes" "no"</Array>
         <Matrix>
            <Array n="2" type="real">100 150</Array>
            <Array n="2" type="real">50 100</Array>
         </Matrix>
      </X-ConfusionMatrixTraining>
   </Extension>
</TreeModel>
```

Figure 2.18: A sample of confusion matrix.

```
<!ELEMENT X-VotingTree EMPTY>
<!ATTLIST X-VotingTree combination_type (and|or|committee) #REQUIRED>
<!ATTLIST X-VotingTree number_of_trees CDATA #REQUIRED>
<!ATTLIST X-VotingTree positive_class CDATA #IMPLIED>
<!ATTLIST X-VotingTree negative_class CDATA #IMPLIED>
```

Figure 2.19: The DTD representing a voting tree.

false, *no*) and negative classes (e.g. *true*, *yes*) required if the combination type is and/or. Finally, the attribute `number_of_trees` counts the number of models that have been used for the voting.

The example reported in figure 2.20 represents a boolean AND classifier among two decision trees.

2.3.4 Clustering model

Description

The process of grouping a set of physical objects into classes of similar objects is called clustering. A cluster is a collection of data objects that are similar within the same cluster and are different from the objects in other cluster.

Clustering methods may be classified into three groups: distance-based, distribution-based (or model-based), density-based methods.

```
...
<TreeModel ...>
   <MiningSchema>
   ...
   </MiningSchema>
   <Node>
      <Extension>
         <X-VotingTree combination_type="and" number_of_trees="2"
                       positive_class="yes" negative_class="no"/>
      </Extension>
      <True/>
      <Node score="will play">
         ...
      </Node>
      <Node score="no play">
         ...
      </Node>
   ...
   </Node>
</TreeModel>
```

Figure 2.20: A sample of voting tree.

Distance-based clustering needs a distance or dissimilarity measurement based on which they try to group those most similar objects into one cluster. K-Means [8] is distance-based partitioning method.

Model-based or *distribution-based clustering* methods assume that the data of each cluster respect a specific statistical distribution (e.g. the Gaussian distribution) and the whole dataset is a mixture of several distribution models. EM [9] is an example of distribution-based partitioning clustering that does not require the specification of distance measures.

Density-based approaches consider a cluster as a dense region of data objects.

The current version of PMML manages both center-based clustering and distribution-based clustering. As a consequence, KDDML offers a full support to the two types of clustering.

Compatibility with PMML v. 2.0

KDDML offers a full compatibility towards PMML v. 2.0 clustering model except for the element `Covariances` belonging to the `Cluster` element. The covariances matrix is used to store coordinate-by-coordinate variances and covariances of the cluster points.

Extension mechanism

The extension mechanism has been used to define a proprietary distance measure for the EM [9] clustering algorithm, as reported in the DTD of figure 2.21 (see sect. 4.2.4).

```
<!ELEMENT X-EMDistance (%NUM-ARRAY;)>
```

Figure 2.21: Algorithm EM extension.

The `X-EMDistance` element takes an array of doubles[4] as input, containing the prior probabilities for each cluster in a distribution-based clustering.

2.3.5 Hierarchy model

Description

The values of a categorical field can be organized into a hierarchy. The representation of hierarchies in PMML is based on parent/child relationships. A tabular format is used to provide the data for these relationships. A taxonomy is then constructed from a sequence of one or more parent/child tables.

In KDDML, the actual values are stored in the hierarchy object. So, the tabular data is part of the PMML document itself. The table is recursive, in the sense that a value in the parent column can also appear in the child column.

Compatibility with PMML v. 2.0

In the current version, KDDML supports the PMML element `InlineTable` that stores data inside the XML document. A further extension of the system can be provided in order to support also the `TableLocator` strategy.

Extension mechanism

The PMML elements `TableLocator` and `InlineTable` are not yet completely defined because other standardization groups are working on these issues. As a consequence, a proprietary definition of the element `InlineTable` has been adapted to the model (see sect. 2.2.2). The figure 2.22 describes the hierarchy *cities-states-countries* of figure 2.9 as physical PMML model.

The figure 2.23 shows the needed DTD extension.

2.4 KDDML Scalar

A KDDML scalar is a basic object that contains a number or a string constant. The figure 2.24 describes his DTD.

2.5 Queries representation

Recall the KDDML language operator structure:

[4]For the definition of the entity `NUM-ARRAY` see PMML.

```
<PMML version="2.0">
   <Header copyright="Copyright (c) 2004 - Universita’ di Pisa, Dipartimento di Informatica">
      <Application name="KDDML (Knowledge Discovery in Databases Markup Language)" version="2.0"/>
   </Header>
   <DataDictionary numberOfFields="2">
      <DataField name="child" optype="ordinal"/>
      <DataField name="parent" optype="ordinal"/>
      <Taxonomy name="cities-states-countries">
         <ChildParent childField="member" parentField="group" isRecursive="yes" x-RootName="USA">
            <InlineTable>
               <Extension>
                  <Row member="California" group="USA"/>
                  <Row member="Illinois" group="USA"/>
                  <Row member="Chicago" group="Illinois"/>
                  <Row member="Long Beach" group="California"/>
                  <Row member="San Jose" group="California"/>
               </Extension>
            </InlineTable>
         </ChildParent>
      </Taxonomy>
   </DataDictionary>
</PMML>
```

Figure 2.22: The hierarchy *cities-states-countries* as PMML model.

```
<!ELEMENT row EMPTY>
<!ATTLIST row member CDATA #REQUIRED>
<!ATTLIST row group CDATA #REQUIRED>
<!ATTLIST ChildParent x-RootName CDATA #REQUIRED>
```

Figure 2.23: Taxonomy model extension.

```
<!ELEMENT KDDML_OBJECT (KDDML_SCALAR)>
<!ELEMENT KDDML_SCALAR EMPTY>
<!ATTLIST KDDML_SCALAR value CDATA #REQUIRED>
<!ATTLIST KDDML_SCALAR type (numeric | string) "string">
```

Figure 2.24: The KDDML_SCALAR element.

```
<OPERATOR_NAME xml_dest="results.xml" att1="v1" ... attM="vM">
   <ARG1_NAME> .... </ARG1_NAME>
   ...
   <ARGn_NAME> .... </ARGn_NAME>
</OPERATOR_NAME>
```

XML tags correspond to operations on data and models; the xml_dest attribute is the name of the object to be saved in the system data/model repository. The attribute is optional; if it is omitted, no results will be stored in the repository. Other XML attributes correspond to parameters of the operator (e.g. the target attribute name for a tree miner operator). Finally, XML sub-elements define the arguments passed to the operator.

Under this interpretation, arguments of an operator must be of an appropriate type and each operator returns a well-defined object. The element KDD_QUERY (see figure 2.25) is used to represent a generic query. The attribute `name` in the `KDD_QUERY` element contains the name of the query. The optional attribute `par_list` lists the set of formal parameters to be replaced by the actual parameters when the query is invoked by means the operator `CALL_QUERY` (see section 3.2.3).

```
<!ELEMENT KDDML_OBJECT (KDD_QUERY)>
<!ELEMENT KDD_QUERY (%kdd_operator;)>
<!ATTLIST KDD_QUERY name %string; #REQUIRED>
<!ATTLIST KDD_QUERY par_list %string_list; #IMPLIED>

<!ENTITY % kdd_operator
    "(%kdd_query_clusters;|%kdd_query_rules;|%kdd_query_sequence;|
      %kdd_query_table;|%kdd_query_trees;|%kdd_query_hierarchy;|
      %kdd_query_scalar;|%kdd_query_PPtable;|%kdd_query_object;)">
```

Figure 2.25: The `KDD_QUERY` element

As mentioned above, the result of a KDD query must be a set of either models, or tables, or scalars. Furthermore the query language must allow the reuse of previous knowledge, and, above all, it must support a *closure principle* in order to combine and refine the extracted knowledge. To do that, a query must have a nested structure, in which it should be possible to combine an arbitrary number of sub-queries containing invocations to DM algorithms or other operators. Moreover, it is necessary to check that sub-queries are properly nested, in order to avoid that a sub-query returns a result that doesn't meet the requirements of the operators combining them.

In order to control the nesting of sub-queries, we group the invocations of operators that returns the same kind of knowledge in the same class, defined by means of a XML entities, as described below. The entity `kdd_operator` reported in figure 2.25 contains an enumeration of all KDDML operators classified according to the entity they belong.

Relational tables entity

The entity `kdd_query_table` (see figure 2.26) contains all operators returning a relational[5] table as output.

```
<!ENTITY % kdd_query_table
    "(ARFF_LOADER|CLUSTER_CENTROID|CLUSTER_NUMBER|CLUSTER_PARTITION|
      DATABASE_LOADER|MISCLASSIFIED|PP_TABLE_2_TABLE|RDA_EXCEPTION|
      RDA_SATISFY|SEQUENCE_EXCEPTION|SEQUENCE_SATISFY|TABLE_LOADER|
      TREE_CLASSIFY|%kdd_query_object;)">
```

Figure 2.26: The `kdd_query_table` entity.

[5]that includes also transactional tables and timestamp tables, as described in sect. 2.2.3.

Preprocessing tables entity

The entity `kdd_query_PPtable` (see figure 2.27) contains all operators returning a preprocessing table.

```
<!ENTITY % kdd_query_PPtable
    "(PP_ADD_HIERARCHY|PP_CHANGE_TYPE|PP_DIVIDING_ATTRIBUTE|
      PP_FILTER_ATTRIBUTES|PP_FOLDING|PP_HIERARCHICAL_DISCRETIZAION|
      PP_MARKING|PP_MARK_DUPLICATES|PP_MERGE_DUPLICATES|
      PP_NEW_ATTRIBUTE|PP_NORMALIZATION|PP_NUMERIC_DISCRETIZATION|
      PP_NUMERIC_LABELING|PP_REMOVE_ROWS|PP_RENAME_ATTRIBUTES|
      PP_REWRITING|PP_SAMPLING|PP_SORTING_ATTRIBUTE|PP_TABLE_LOADER|
      TABLE_2_PP_TABLE|%kdd_query_object;)">
```

Figure 2.27: The `kdd_query_PPtable` entity.

Association rules entity

The entity `kdd_query_rules` (see figure 2.28) contains all operators returning an association model.

```
<!ENTITY % kdd_query_rules
    "(RDA_FILTER|JDM_CONNECTION|RDA_LOADER|PMML_RDA_LOADER|RDA_MINER|
      RDA_PRESERVED|%kdd_query_object;)">
```

Figure 2.28: The `kdd_query_rules` entity.

Tree model entity

The entity `kdd_query_trees` (see figure 2.29) contains all operators returning a classification tree.

```
<!ENTITY % kdd_query_trees
    "(JDM_CONNECTION|PMML_TREE_LOADER|TREE_LOADER|
      TREE_META_CLASSIFIER|TREE_MINER|%kdd_query_object;)">
```

Figure 2.29: The `kdd_query_trees` entity.

Clustering model entity

The entity `kdd_query_cluster` (see figure 2.30) contains all operators returning a clustering model.

```
<!ENTITY % kdd_query_clusters
    "(CLUSTER_LOADER|CLUSTER_MINER|JDM_CONNECTION|PMML_CLUSTER_LOADER|
      %kdd_query_object;)">
```

Figure 2.30: The `kdd_query_clusters` entity.

Items hierarchy entity

The entity `kdd_query_hierarchy` (see figure 2.31) contains all operators returning an item hierarchy.

```
<!ENTITY % kdd_query_hierarchy
    "(HIERARCHY_LOADER|JDM_CONNECTION|TABLE_2_HIERARCHY|
      %kdd_query_object;)">
```

Figure 2.31: The `kdd_query_hierarchy` entity.

Sequences entity

The entity `kdd_query_sequence` (see figure 2.32) contains all the operators returning sequential patterns.

```
<!ENTITY % kdd_query_sequence
    "(PMML_SEQUENCE_LOADER|SEQUENCE_AGGREGATE_FILTER|SEQUENCE_FILTER|
      SEQUENCE_LOADER|SEQUENCE_MAXIMAL_FILTER|SEQUENCE_MINER|
      SEQUENCE_RULE|SEQUENCE_TIMESTAMP_CONSTRAINT|%kdd_query_object;)">
```

Figure 2.32: The `kdd_query_sequence` entity.

Scalar entity

The entity `kdd_query_scalar` (see figure 2.33) contains all operators returning a number or a string.

```
<!ENTITY % kdd_query_scalar
    "(ARFF_WRITER|DATABASE_WRITER|EXT_CALL|SCALAR|%kdd_query_object;)">
```

Figure 2.33: The `kdd_query_scalar` entity.

kdd object entity

Some KDDML operators do not have a precise signature. In other terms, the output type of the operator can be unknown at compile-time, and it can be derived only at run-time by the KDDML interpreter, when the query is executed. For instance, consider the `IF` operator (sect. 3.2.14) that evaluates a condition on an input object and, on the basis of the boolean result, it evaluates a `THEN` branch or an `ELSE` branch. Therefore, a further entity (see figure 2.34) is needed in order to group all operators returning a generic KDDML object whose type is unknown at compile-time.

```
<!ENTITY % kdd_query_object
     "(CALL_QUERY|SEQ_QUERY|PAR_QUERY|IF)">
```

Figure 2.34: The `kdd_query_scalar` entity.

Other special objects are used to define algorithm settings, condition and expression specifications and generic XML elements, as reported below. They are defined by means XML elements (and not by entities) since they are only input types for operators (never output types).

Algorithm settings

An algorithm settings object captures the parameters associated with a particular algorithm. It allows a knowledgeable user to fine tune algorithm parameters. Generally, not all parameters must be specified, however, those specified are taken into account by the KDDML executor.

This entity is used to specify a data mining or preprocessing algorithm; it is composed by the *algorithm name* identifying the algorithm, and by a list of *parameter specifications* with parameter name and parameter value (e.g., the minimum support for a rda miner algorithm). Its definition is reported in the DTD of figure 2.35.

```
<!ELEMENT ALGORITHM (PARAM)*>
<!ATTLIST ALGORITHM algorithm_name %string; #REQUIRED>
<!ELEMENT PARAM EMPTY>
<!ATTLIST PARAM name %string; #REQUIRED>
<!ATTLIST PARAM value %any_type; #REQUIRED>
```

Figure 2.35: The element ALGORITHM.

The attribute `algorithm_name` is the name of the algorithm (e.g. *apriori*, *em*, etc.). The element root can be composed by a list of `PARAM` elements, representing a single algorithm parameter. We can specify the parameter name and the parameter value by using the respective required attributes. In the definition of DTD, we prefer to maintain a non-strict semantic for algorithm specification in order to preserve an high language extendibility: adding a new pre-processing or mining algorithm in future do not require any DTD modification. Therefore, the parameter value can assume any generic type (number, or string).

Conditions

This entity is defined to represent a condition specification. The condition can be used to evaluate boolean operators (such as "$\leq$") on table attributes and/or constants. Here, table attributes stand for both relational (or preprocessing) table columns and model properties (e.g. the support of an association rule).

As shown in figure 2.36, the element `CONDITION` is composed by an `AND, OR, NOT` combination between primitive cases (element `BASE_COND`). Each base condition can be expressed using the attribute `op_type`, representing the name of the boolean operator. The arguments to the operator can be given using the `term` XML attributes. As shown, operators can be unary, binary or ternary. By convention, it is possible to denote an input table column instead of a constant using the special symbol "@" before the attribute name in the `term` XML attribute.

An example of condition application has been reported in the query of figure 1.2.

```
<!ELEMENT CONDITION (TRUE|FALSE|OR_COND|NOT_COND|AND_COND|BASE_COND)>
<!ELEMENT TRUE EMPTY>
<!ELEMENT FALSE EMPTY>
<!ELEMENT OR_COND ((OR_COND|NOT_COND|AND_COND|BASE_COND),
                   (OR_COND|NOT_COND|AND_COND|BASE_COND)+)>
<!ELEMENT AND_COND ((OR_COND|NOT_COND|AND_COND|BASE_COND),
                    (OR_COND|NOT_COND|AND_COND|BASE_COND)+)>
<!ELEMENT NOT_COND ((OR_COND|NOT_COND|AND_COND|BASE_COND))>
<!ELEMENT BASE_COND EMPTY>
<!ATTLIST BASE_COND op_type %string; #REQUIRED
                term1 %any_type; #REQUIRED
                term2 %any_type; #IMPLIED
                term3 %any_type; #IMPLIED>
```

Figure 2.36: The element CONDITION.

Expressions

This object is used to represent language expressions. Expressions are similar to conditions, but they return a scalar (i.e., a number or a string) instead of a boolean value.

As shown in figure 2.37, the `EXPRESSION` element admits a sequential statement (element `SEQ_TERM`), including basic operations (*addition*, *multiplication*, *subtraction*, *division* of numbers and *concatenation* of strings) on primitive terms. Also a conditional statement (element `IF_TERM`) is admitted. It is used to evaluate a *then* statement or an (optional) *else* statement according to a condition whose specification is reported in figure 2.36 and table 2.4. Finally, the element `BASE_TERM` is used to provide numeric/string constants or table attribute names[6].

[6]As for conditions, with the special symbol "@" in the `value` attribute of the `BASE_TERM` element we denote an input table column.

```
<!ELEMENT EXPRESSION (BASE_TERM|SEQ_TERM|IF_TERM)>
<!ELEMENT SEQ_TERM ((BASE_TERM|SEQ_TERM|IF_TERM),
                    (BASE_TERM|SEQ_TERM|IF_TERM)+)>
<!ATTLIST SEQ_TERM op_type
        (concat|equal|sum|multiply|subtract|divide) #REQUIRED>
<!ELEMENT BASE_TERM EMPTY>
<!ATTLIST BASE_TERM value %any_type; #REQUIRED>
<!ELEMENT IF_TERM (CONDITION, (BASE_TERM|SEQ_TERM|IF_TERM),
                              (BASE_TERM|SEQ_TERM|IF_TERM)?)>
```

Figure 2.37: The element EXPRESSION.

OpType	**Term 1 type**	**Term 2 type**	**Term 3 type**
equal, **not_equal**	@attribute	@attribute numeric or string constant	-
greater, **greater_or_equal** **less**, **less_or_equal**	@numeric_attribute	@numeric_attribute numeric constant	-
is_missing	@attribute	-	-

Table 2.4: Condition specification for the element IF_TERM of expressions

Generic XML

The XML type denotes arguments that are generic XML elements to be evaluated directly by the operator. Its semantics is to model XML tags that are not interpreted directly by KDDML, but simply passed to the operator as arguments. So, as shown in figure 2.38, its DTD is totally generic.

```
<!ELEMENT GENERIC_ELEMENT ANY>
```

Figure 2.38: The element GENERIC_ELEMENT.

In summary, the set of types of a KDDML operator is reported in table 2.5.

KDDML Type	**DTD Entity / XML element**	**Description**
table	*kdd_query_table*	all operators returning a relational table
PPtable	*kdd_query_PPtable*	all operators returning a preprocessing table
rda	*kdd_query_rules*	all operators returning an association model
tree	*kdd_query_trees*	all operators returning a classification model
cluster	*kdd_query_cluster*	all operators returning a cluster model
hierarchy	*kdd_query_hierarchy*	all operators returning an item hierarchy
sequence	*kdd_query_sequence*	all operators returning a sequence model
scalar	*kdd_query_scalar*	all operators returning a scalar
any	*kdd_query_object*	all operators returning any type
alg	*ALGORITHM*	an algorithm specification
condition	*CONDITION*	a condition specification
expression	*EXPRESSION*	an expression specification
xml	*GENERIC_ELEMENT*	a generic XML element

Table 2.5: Correspondence between XML entities and KDDML types

CHAPTER 3

KDDML operators

This chapter contains all the KDDML language operators specification. For each operator it reports:

1. *the operator name represented by an XML root tag;*
2. *the XML Document Type Definition;*
3. *the usage description of the operator;*
4. *the KDD phase supported by the operator;*
5. *the operator signature;*
6. *a list of required XML attributes with usage description;*
7. *a list of optional XML attributes with usage description.*

Language operators will be presented according to a lexicographic ordering.

3.1 Classification of operators

In chapter two we have seen that operators can be classified on the basis of the type they return. Before introducing the KDDML language specification, we give a different classification of operators, according to the KDD phase supported.

3.1.1 I/O operators

I/O class includes operators that populate the KDDML data/models repository from external resources or that export a table/model in the repository to external format. The table 3.1 shows the I/O operators currently implemented in KDDML.

Operator Name	Description	Sect.
ARFF_LOADER	It loads an ARFF source	3.2.1
ARFF_WRITER	It accesses to a table in the repository and transforms it into an ARFF file	3.2.2
CLUSTER_LOADER	It loads a cluster model from the repository	3.2.5
DATABASE_LOADER	It loads a relational table from a database	3.2.10
DATABASE_WRITER	It accesses to a table in the repository and transforms it into a SQL table	3.2.11
HIERARCHY_LOADER	It loads an item hierarchy from the repository	3.2.13
PMML_CLUSTER_LOADER	It loads an external PMML cluster model	3.2.17
PMML_RDA_LOADER	It loads an external PMML association model	3.2.18
PMML_SEQUENCE_LOADER	It loads an external PMML sequence model	3.2.19
PMML_TREE_LOADER	It loads an external PMML tree model	3.2.20
PP_TABLE_LOADER	It loads a preprocessing table from the repository	3.2.40
RDA_LOADER	It loads an association model from the repository	3.2.43
SEQUENCE_LOADER	It loads a sequence model from the repository	3.2.51
TABLE_2_HIERACHY	It loads an item hierarchy from a table	3.2.58
TREE_LOADER	It loads a classification tree from the repository	3.2.62

Table 3.1: I/O operators

3.1.2 Preprocessing operators

Preprocessing is a time-consuming phase of the KDD process, including tasks such as:

- data cleaning;
- data reduction;
- data discretization;

- data transformation.

The table 3.2 reports the preprocessing operators implemented in KDDML divided in respect of the preprocessing step supported; however, some operators are multi-task. Every preprocessing operator takes a `PPtable` object as first argument and returns a `PPtable` object, as shown by the following signature:

$$f_{<\mathrm{PP}\_\ldots>} : \texttt{PPtable} \times \ldots \longrightarrow \texttt{PPtable}.$$

Operator Name	**Description**	**Sect.**	**Prepr. Step**
PP_MARKING	It marks the preprocessing values of an attribute according to a condition	3.2.27	Data cleaning
PP_MARK_DUPLICATES	It marks duplicated instances	3.2.28	
PP_MERGE_DUPLICATES	It finds and unifies instances that are duplicates	3.2.29	
PP_FILTER_ATTRIBUTES	It select (or remove) a list of attributes	3.2.24	Data reduction
PP_REMOVE_ROWS	It deletes rows according to a condition	3.2.34	
PP_SAMPLING	It performs a sampling on input table	3.2.37	
PP_ADD_HIERARCHY	It assigns an hierarchy object to a table column as meta-data information	3.2.21	Data discretiz.
PP_HIERARCHICAL_ DISCRETIZATION	It performs a categoric discretization of a nominal attribute	3.2.26	
PP_NUMERIC_ DISCRETIZATION	It discretizes a numeric attribute	3.2.32	
PP_CHANGE_TYPE	It changes the logical type of one or more attributes	3.2.22	Data transform.
PP_DIVIDING_ATTRIBUTE	It partitions the values of an input attribute into two new attributes	3.2.23	
PP_FOLDING	It removes a specified attribute copying its values into a destination attribute	3.2.25	
PP_NEW_ATTRIBUTE	It adds a new attribute to the input table	3.2.30	
PP_NORMALIZATION	It performs a normalization of a numeric attribute	3.2.31	
PP_NUMERIC_LABELING	It converts a nominal attribute into a numeric attribute	3.2.33	
PP_RENAME_ATTRIBUTES	It renames a list of attributes	3.2.35	

PP_REWRITING	It rewrites the values of an attribute according to a regular expression	3.2.36	
PP_SORTING_ATTRIBUTE	It sorts the values of a given attribute	3.2.38	
PP_TABLE_2_TABLE	It finalizes the preprocessing phase	3.2.39	-
TABLE_2_PP_TABLE	It starts the preprocessing phase	3.2.59	

Table 3.2: Preprocessing operators

3.1.3 Mining operators

KDDML enables users to build model in the functional areas: classification, association rules, clustering, etc. To build models, users define tasks, which minimally require the input parameters model name and mining settings expressed by using the XML element `ALGORITHM` (see figure 2.35). Mining data can be given as first argument of the operator. The result is a mining model.

Summarizing, there is an operator for each mining model supported by the language (see table 3.3) and the signature of operators is fixed apriori for each operator:

$$f_{<\texttt{KNOWLEDGE\_MINER}>} : \texttt{table} \times \texttt{alg} \longrightarrow \texttt{model}.$$

Operator Name	**Description**	**Sect.**
CLUSTER_MINER	It extracts a clustering model by using a mining algorithm	3.2.6
RDA_MINER	It extracts a set of association rules by using a mining algorithm	3.2.44
SEQUENCE_MINER	It extracts a sequence model by using a mining algorithm	3.2.53
TREE_MINER	It extracts a classification tree by using a mining algorithm	3.2.64

Table 3.3: Mining operators

3.1.4 Postprocessing operators

Extracted models can be applied on (new) data to predict features or to select data accordingly to the knowledge stored in the model. The table 3.4 shows the postprocessing operators that include:

- *model application* including operators to apply extracted model;

- *model testing* that gives an estimate of the accuracy a model has in predicting the target of a supervised model;
- *model meta-reasoning* including operators to combine two or more models;
- *model filtering* including operators to filter extracted models according to some feature.

Operator Name	**Description**	**Sect.**	**Model(s) involved**
CLUSTER_CENTROID	Given a cluster model it returns tuples describing the centroids	3.2.4	clusters
CLUSTER_NUMBER	Returns the tuples of a dataset belonging to a given cluster	3.2.7	
CLUSTER_PARTITION	Partitions the tuple of a dataset according to a clustering model	3.2.8	
CLUSTER_PARTITION _SPARROW	Partitions the tuple of a dataset by using the SPARROW algorithm	3.2.9	
RDA_EXCEPTION	Selects the transaction that are exception to a set of association rules	3.2.41	rda
RDA_SATISFY	Selects the transaction satisfying a set of association rules	3.2.46	
RDA_PRESERVED	Selects the rules preserving over an item hierarchy	3.2.45	rda, hierarchy
SEQUENCE_AGGREGATE _FILTER	It returns the sequential patterns satisfying an aggregate constraint	3.2.48	sequence
SEQUENCE_EXCEPTION	It selects the transaction that are exceptions to a set of sequential patterns	3.2.49	
SEQUENCE_FILTER	It returns the sequential patterns satisfying a condition	3.2.50	
SEQUENCE_RULE	It computes sequence rules from a set of sequential patterns	3.2.54	
SEQUENCE_SATISFY	It selects the transaction satisfying a set of sequential patterns	3.2.55	
SEQUENCE_TIMESTAMP _FILTER	It returns the sequential patterns satisfying a time-stamp constraint	3.2.56	
MISCLASSIFIED	It selects the misclassified instances of an input table	3.2.15	tree
TREE_CLASSIFY	If classifies an input table on a classification tree	3.2.61	
TREE_META_CLASSIFIER	It combines two or more tree classifier using a committee strategy	3.2.63	

Table 3.4: Postprocessing operators

3.1.5 Control flow operators

The semantic of the KDDML language can be augmented with operators that allow for better control of flow of data and model in queries. Table 3.5 shows that operators.

Operator Name	Description	Sect.
CALL_QUERY	It retrieves and evaluates queries in the query repository	3.2.3
IF	Definition of the decision statement	3.2.14
PAR_QUERY	Definition of potential parallelism	3.2.16
SEQ_QUERY	Definition of sequential statement	3.2.57

Table 3.5: Control flow operators

3.1.6 Unclassified operators

The table 3.6 shows all the other KDDML language operators.

Operator Name	Description	Sect.
EXT_CALL	It calls an external program	3.2.12
SCALAR	It returns the value of an XML attribute as scalar	3.2.47

Table 3.6: Unclassified operators

3.2 Operators specification

3.2.1 ARFF_LOADER

DTD

```
<!ELEMENT ARFF_LOADER EMPTY>
<!ATTLIST ARFF_LOADER xml_dest %string; #IMPLIED>
<!ATTLIST ARFF_LOADER arff_file_name %string; #REQUIRED>
<!ATTLIST ARFF_LOADER arff_file_path %string; #IMPLIED>
```

Description

ARFF (*Attribute-Relation File Format*) file is a text file that consists of a list of instances with the attribute value for each instance being separated by commas. The name of the dataset is introduced by a `@relation` tag, and the names, types and values of each attribute are defined by `@attribute` tags. The data section of the ARFF file begin with the `@data` tag. By convection, missing values are represented by a single question mark. Values that contain spaces must be quoted. Each attribute in the data set has its own `@attribute` statement which uniquely defines the name of that attribute and it is data type. The order the attributes are declared indicates the column position in the data section of the file. The format for the `@attribute` statement is:

$$@\text{attribute} < \text{attribute name} > < \text{datatype} >$$

The datatype can be numeric (integer or real), string or nominal. Nominal values are defined by providing a nominal specification listing the possible values: {<nominal-name1>, <nominal-name2>, <nominal-name3>, etc}. As an example, the weather dataset in ARFF format is reported in figure 3.1. The operator loads an ARFF source gathered via local file system or via ftp, http protocols. Mapping from ARFF types to the logical types of the output table is automatic.

KDD phase

Resource loading.

Signature

$f_{\texttt{ARFF\_LOADER}} : empty \rightarrow \texttt{table}$

Required attributes

- `arff_file_name`: the name of the input ARFF file.

Optional Attributes

- `xml_dest`;
- `arff_file_path`: is the external path identifying the ARFF file. ARFF file can be obtained via a local file system (e.g. *D:/MyRepository/*) or via an internet repository (e.g. *ftp://www.foo.edu/ARFF/*). If the attribute is omitted, the ARFF source is read directly from the data system repository.

3.2.2 ARFF_WRITER

DTD

```
<!ELEMENT ARFF_WRITER (%kdd_query_table;)>
<!ATTLIST ARFF_WRITER xml_dest %string; #IMPLIED>
<!ATTLIST ARFF_WRITER arff_dest %string; #REQUIRED>
```

```
@relation weather

@attribute outlook {sunny, overcast, rainy}
@attribute temperature numeric
@attribute humidity numeric
@attribute windy {TRUE, FALSE}
@attribute play {yes, no}

@data
sunny,85,85,?,no
sunny,80,90,TRUE,no
overcast,83,86,FALSE,yes
rainy,70,?,FALSE,yes
rainy,68,80,FALSE,yes
?,65,70,?,no
...
```

Figure 3.1: The weather dataset in ARFF format.

Description

The operator accesses a relational table contained in the data repository, and it transforms the table into an ARFF file (see sect. 3.2.1). The mapping from the data source to the logical types of the output ARFF format is automatic.
The operator returns an integer, containing the total number of table rows involved.

KDD phase

None.

Signature

$f_{\texttt{ARFF\_WRITER}}$: `table` $\longrightarrow$ `scalar`

Required attributes

- `arff_dest`: it is the complete path containing the destination ARFF file (e.g. *D:/MyRepository/weather.arff*).

Optional Attributes

- `xml_dest`.

3.2.3 CALL_QUERY

DTD

```
<!ELEMENT CALL_QUERY EMPTY>
<!ATTLIST CALL_QUERY query_name %string; #REQUIRED>
<!ATTLIST CALL_QUERY query_path %string; #IMPLIED>
<!ATTLIST CALL_QUERY formal_parameters_list %string; #IMPLIED>
<!ATTLIST CALL_QUERY actual_parameters_list %string; #IMPLIED>
```

Description

The `CALL_QUERY` statement retrieves and evaluates queries in the repository and it replaces formal parameters with actual parameters at run-time. Query admits parameters, whose list is specified by means XML attributes; the operator performs a position mapping between formal parameters and actual parameters. The syntax for using a formal parameter inside a query (the called query) requires to write it between $ sign (e.g.: $perc$).

The operator returns the same kind of result of the called query. Since the type may not be known at compile time (e.g., when the query name itself is provided by a parameter), the type of the result is checked at run-time.

KDD phase

Control flow.

Signature

$f_{\texttt{CALL\_QUERY}} : empty \rightarrow \texttt{any}$

Required attributes

- `query_name`: the name of the xml file referring the query.

Optional Attributes

- `query_path`: is the external path identifying the query. If the attribute is omitted, the query source is read directly from the queries system repository.
- `formal_parameters_list` the list of formal parameters that is in a comma separated format (e.g. *perc, source, dest*).
- `actual_parameters_list` the list of actual parameters. The list is given in a comma separated format (e.g. *0.6, weather, result*) and a position mapping with the values of `formal_parameters_list` is performed.

3.2.4 CLUSTER_CENTROID

DTD

```
<!ELEMENT CLUSTER_CENTROID (%kdd_query_clusters;)>
<!ATTLIST CLUSTER_CENTROID xml_dest %string; #IMPLIED>
```

Description

Given a cluster model, it returns tuples describing the cluster centroids. The centroid is dependent on the type of clustering.
For centroid-based clustering, there is an instance identifying the cluster. This instance will be reported by the operator as centroid.
For distribution-based clustering, the centroid is defined by statistics and depends on the type of attributes. In particular, for numeric attributes, the centroid contains the mean of instances belonging to the cluster. For nominal attributes, the most probable category (i.e. the category with the largest frequency) is reported as value for that attribute.

KDD phase

Model application.

Signature

$f_{\texttt{CLUSTER\_CENTROID}} : \texttt{cluster} \rightarrow \texttt{table}$

Required attributes

None.

Optional Attributes

- `xml_dest`.

3.2.5 CLUSTER_LOADER

DTD

```
<!ELEMENT CLUSTER_LOADER EMPTY>
<!ATTLIST CLUSTER_LOADER xml_source %string; #REQUIRED>
```

Description

It loads a cluster model from the system repository.

KDD phase

Resource loading.

Signature

$f_{<\texttt{CLUSTER\_LOADER}>} : empty \rightarrow$ `tree`.

Required attributes

- `xml_source`: the XML file source contained in the models repository.

Optional attributes

None.

3.2.6 CLUSTER_MINER

DTD

```
<!ELEMENT CLUSTER_MINER((%kdd_query_table;), ALGORITHM)>
<!ATTLIST CLUSTER_MINER xml_dest %string; #IMPLIED>
```

Description

The operator extracts a cluster model by using a mining algorithm. The operator takes a table representing the training set and a cluster miner algorithm, and it returns a set of clusters as output. The algorithm specification (i.e. the algorithm name and the list of expected parameters) is expressed by using the XML element `ALGORITHM` (see figure 2.35). In section 4.1.7, the list of supported clustering algorithms is reported.
The data schema of the input data source depends on the algorithm specification. In other words, some attributes can be ignored during mining if their types are not supported by the algorithm. For example, some clustering algorithms cannot work on string attributes. However, preprocessing operators can be used to adapt the input table to specific data mining algorithms.

KDD phase

Data mining.

Signature

$f_{<\texttt{CLUSTER\_MINER}>} : \texttt{table} \times \texttt{alg} \rightarrow \texttt{cluster}$.

Required attributes

None.

Optional attributes

- `xml_dest`.

3.2.7 CLUSTER_NUMBER

DTD

```
<!ELEMENT CLUSTER_NUMBER ((%kdd_query_clusters;),
                          (%kdd_query_table;))>
<!ATTLIST CLUSTER_NUMBER xml_dest %string; #IMPLIED
                         cluster_number %integer; #IMPLIED>
```

Description

Given a cluster model and a dataset, this operator returns only the tuples of the dataset belonging to a specified cluster.
The mapping between attributes used in the clustering and attributes in the input table is by name. Therefore, the clustering model and the data source must be compatible. In particular, a table is compatible with a model if for each active mining field [1] belonging to the model, there is an attribute in the table with the same name and type.
The prediction of which cluster contains the instance is performed by comparing the instance and the cluster centroid. When two records are compared then either the distance or the similarity is of interest. In both cases the measures can be computed by a combination of an *inner function* and an *outer function*. The *inner function* compares two single fields (e.g. absolute difference between continuous attributes) values and the *outer function* (e.g. euclidean distance) computes an aggregation over all fields. *Inner function* and *outer function* are stored in the clustering model when the clusters are built.
The operator can be also used to predict the tuples belonging to the cluster with maximal cardinality.

KDD phase

Model application.

Signature

$f_{<\texttt{CLUSTER\_NUMBER}>}$: $\texttt{cluster} \times \texttt{table} \longrightarrow \texttt{table}$.

Required attributes

None.

Optional Attributes

- `xml_dest`.

[1]An active mining field is a field used as input to the model. In other terms, the user must supply all the active mining fields in order to apply the model.

- `cluster_number`: a non-negative integer representing the index of the cluster. If the attribute is omitted, the operator returns all the instances belonging to the cluster with maximal cardinality.

3.2.8 CLUSTER_PARTITION

DTD

```
<!ELEMENT CLUSTER_PARTITION ((%kdd_query_clusters;),
                             (%kdd_query_table;))>
<!ATTLIST CLUSTER_PARTITION xml_dest %string; #IMPLIED>
```

Description

Given a cluster model and a dataset, this operator adds to the tuples a new numeric attribute `cluster_number` containing the index of the cluster that includes the record. In order to understand how the mapping between instances and clusters is performed, see the CLUSTER_NUMBER operator (sect. 3.2.7).

KDD phase

Model application.

Signature

$f_{<\texttt{CLUSTER\_PARTITION}>} : \texttt{cluster} \times \texttt{table} \rightarrow \texttt{table}.$

Required attributes

None.

Optional Attributes

- `xml_dest`.

3.2.9 CLUSTER_PARTITION_SPARROW

DTD

```
<!ELEMENT CLUSTER_PARTITION_SPARROW ((%kdd_query_table;))>
<!ATTLIST CLUSTER_PARTITION_SPARROW xml_dest %string; #IMPLIED>
<!ATTLIST CLUSTER_PARTITION_SPARROW num_agents %integer; "100">
<!ATTLIST CLUSTER_PARTITION_SPARROW num_iterations %integer; "200">
<!ATTLIST CLUSTER_PARTITION_SPARROW density_threshold %integer; "19">
<!ATTLIST CLUSTER_PARTITION_SPARROW visibility_radius %integer; "9">
```

Description

Given a dataset, this operator computes a clustering partition of input instances and it adds to the tuples a new numeric attribute `cluster_number` containing the index of the cluster that includes the record. The operator uses the Sparrows [10] algorithm in order to extract the set of clusters.

Sparrow is a new parallel algorithm that uses the new swarm intelligence based techniques to investigate clustering in spatial data. The algorithm combines a smart exploratory strategy based on a flock of birds that move around a cellular landscape that contains the data set with a density-based cluster algorithm to discover clusters of arbitrary shape and size in spatial data. Agents use modified rules of the standard flock algorithm to transform an agent into a hunter foraging for clusters in spatial data. Clusters are discovered applying the heuristic principles of the spatial clustering algorithm DBSCAN.

Input table admits two numerical attributes only. The first one represents the spacial x-coordinate; the second one represents the spacial y-coordinate. Other columns are not admitted in the input table and no missing values are allowed.

KDD phase

Model application.

Signature

$f_{<\mathtt{CLUSTER\_PARTITION\_SPARROW}>}$: `table` $\longrightarrow$ `table`.

Required attributes

None.

Optional Attributes

- `xml_dest`.
- `num_agents`: a positive integer representing the number of agents to use. Default: 100.
- `num_iterations`: a positive integer representing the maximum number of iterations before the algorithm converges. Default: 200.
- `density_threshold`: a positive integer representing the density threshold used by Sparrow during mining. Default: 19.
- `visibility_radius`: a positive integer representing the visibility radius. Default: 9.

3.2.10 DATABASE_LOADER

DTD

```
<!ELEMENT DATABASE_LOADER EMPTY>
```

```
<!ATTLIST DATABASE_LOADER xml_dest %string; #IMPLIED
                          sql_query %string; #REQUIRED
                          database_name %string; #REQUIRED
                          user %string; #IMPLIED
                          password %string; #IMPLIED
                          logical_relation_name %string; #IMPLIED>
```

Description

The operator allows a transparent access to relational tables belonging to local or remote RDBMS sources via simple SQL SELECT queries.
The mapping from SQL types to the logical types of the output table is automatic. Numeric SQL types (as *small int* or *real*) are mapped into numeric attributes, while non-numeric SQL types (as *varchar*) are mapped into string attributes. However, preprocessing operators allows for specifying different logical types of attributes for loaded tables.
The operator uses the JDBC (*Java Database Connectivity*) relational database connectivity standard as a DBMS bridge. The URL for the connection with the DBMS and the SQL query can be specified directly as input XML attributes. User and password are not required in the operator definition; if they are omitted, a pop-up frame will be automatically opened in order to initialize the DBMS connection.

KDD phase

Resource loading.

Signature

$f_{\texttt{DATABASE\_LOADER}} : empty \rightarrow \texttt{table}$

Required attributes

- `sql_query`: the sql query as string. If the SQL query does not return a table as output (for example, when using an `UPDATE` statement), the operator will return an error message.
- `database_name`: the database URL identifying the RDBMS to be used during connection. At present, KDDML accepts connectivity via Oracle or SQL Server databases. Below, the URL connection strings currently available are reported[2]:
  - SQL Server via JTDS driver:
    `jdbc:jtds:sqlserver://<host>:<port>/<database>`;

[2]Notice that JDBC drivers, in general, are pure Java classes, independent from operating system, that are packed as Java archive (.jar) files. Other JDBC drivers can be easily added to the system (language). In order to make a JDBC driver available for KDDML, you should obtain the driver from your database (or 3rd party JDBC driver) developer, and then put its .jar file into the KDDML installation directory.

  - SQL Server via Microsoft driver:
    `jdbc:microsoft:sqlserver://<server_name>:<port>; DatabaseName=<database_name>`;
  - Oracle JDBC driver:
    `jdbc:oracle:thin:[<user>/<password>]@//<host>[:<port>]/<service>`;
  - PostgreSQL:
    `jdbc:postgresql://<host>:<port>/<database>`;

Optional Attributes

- `xml_dest`.
- `user`: the user name to be used during connection.
- `password`: the password to be used during connection.
- `logical_relational_name`: the logical relation name to be assigned to output table.

3.2.11 DATABASE_WRITER

DTD

```
<!ELEMENT DATABASE_WRITER (%kdd_query_table;)>
<!ATTLIST DATABASE_WRITER xml_dest %string; #IMPLIED>
<!ATTLIST DATABASE_WRITER database_name %string; #REQUIRED>
<!ATTLIST DATABASE_WRITER table_name %string; #REQUIRED>
<!ATTLIST DATABASE_WRITER user %string; #IMPLIED>
<!ATTLIST DATABASE_WRITER password %string; #IMPLIED>
```

Description

The operator accesses a relational table contained in the data repository and it transform the table into a SQL table. The mapping from the data source to the logical types of the database format is not automatic. A requirement of the operator is that in the destination database an empty SQL table must exist; this is the target relational table. No type control is performed by the operator between the logical proprietary types of the attributes and the SQL types of columns belonging to the target SQL table. The operator returns an integer, containing the total number of table rows involved.

KDD phase

None.

Signature

$f_{\texttt{DATABASE\_WRITER}} : \texttt{table} \longrightarrow \texttt{scalar}$

Required attributes

- `database_name`: it is the database URL identifying the RDBMS to use during connection (see the DATABASE_LOADER operator in section 3.2.10 for more details).

Optional Attributes

- `xml_dest`.
- `user`: it is the user name used during connection.
- `password`: it is the password used during connection.
- `table_name`: it is the empty target SQL table created in the specified database.

3.2.12 EXT_CALL

DTD

```
<!ELEMENT EXT_CALL (%kdd_query_scalar;)*>
<!ATTLIST EXT_CALL xml_dest %string; #IMPLIED>
<!ATTLIST EXT_CALL path %string; #REQUIRED>
```

Description

The operator allows for calling external programs, including e.g., calls to RDBMS stored procedures.
It takes a set of scalars used as a command line argument to the called program, and it returns a new scalar containing a boolean value representing the success or the failure of the external call.

KDD phase

None.

Signature

$f_{<\text{SCALAR}>} : \texttt{scalar} \times \cdots \times \texttt{scalar} \longrightarrow \texttt{scalar}.$

Required attributes

- `path`: the complete path containing the external program (e.g. *D:/usr/bin/MyProgram.exe*).

Optional Attributes

- `xml_dest`.

3.2.13 HIERARCHY_LOADER

DTD

```
<!ELEMENT HIERARCHY_LOADER EMPTY>
<!ATTLIST HIERARCHY_LOADER xml_source %string; #REQUIRED>
```

Description

It loads an item hierarchy from the system model repository.

KDD phase

Resource loading.

Signature

$f_{\texttt{<HIERARCHY\_LOADER>}} : empty \rightarrow \texttt{hierarchy}$.

Required attributes

- `xml_source`: the XML file source contained in the models repository.

Optional attributes

None.

3.2.14 IF

DTD

```
<!ELEMENT IF (%kdd_operator;, CONDITION, THEN, ELSE)>
<!ATTLIST IF xml_dest %string; #IMPLIED>
<!ELEMENT THEN (%kdd_operator;)>
<!ELEMENT ELSE (%kdd_operator;)>
```

Description

The operator evaluates a condition on an input object and, on the basis of the boolean result, it evaluates (in non-strict semantic) a `THEN` branch or an `ELSE` branch. The condition regards a Xquery [11] expression evaluated on the physical XML document representing the input object (see later).

More precisely, the operator take as input four objects:

1. a query fragment that is evaluated as first and returning a `KDDML object`;
2. a boolean condition evaluated on the `KDDML object` above;

3. a query fragment representing the `THEN` branch evaluated only if the condition returns true;
4. a query fragment representing the `ELSE` branch evaluated only if the condition returns false.

The operator returns the same kind of result of the `THEN` branch or the `ELSE` branch (i.e. any type returned by the operators belonging to the entity `kdd_operator`). Since the type may not be known at compile time, the type of the result is checked at run-time.

KDD phase

Control flow.

Signature

$f_{\texttt{<IF>}}$: `any` $\times$ `condition` $\times$ `any` $\times$ `any` $\longrightarrow$ `any`.

Required attributes

None.

Optional attributes

- `xml_dest`.

Condition specification

In reference to the figure 2.36, legal values for the XML attribute `op_type` are reported below:

- **equal**, **not_equal**, **greater**, **greater_or_equal**, **less**, **less_or_equal**: used for relational operators on a XQuery expression and a constant. More precisely, the first term is a Xquery expression whose evaluation must return a constant value that is then compared with the second term of the relational operator. The Xquery expression is evaluated on the input XML physical object that can be referred by using the notation $input (e.g. return $input//X-ConfusionMatrixTraining/@x-incorrectlyInst). Current version of KDDML adopts the Qizx [12] implementation of XQuery.

The table 3.7 contains the types of primitive terms (XML attributes `term1, term2, term3`) to be used for each `op_type` legal value.

3.2.15 MISCLASSIFIED

DTD

```
<!ELEMENT MISCLASSIFIED (%kdd_query_table;)>
<!ATTLIST MISCLASSIFIED xml_dest %string; #IMPLIED>
```

OpType	Term 1 type	Term 2 type	Term 3 type
equal, **not_equal**	x-query expression constant	constant	-
greater, **greater_or_equal** **less**, **less_or_equal**	x-query expression numeric constant	numeric constant	-

Table 3.7: Element BASE_COND for IF operator

Description

The operator takes a table whose instances have been previously classified via a classification tree (see sect. 3.2.61), and it returns a new table with the misclassified records[3] only.
The input table must contain at least two attributes:

1. the real target attribute containing the observed values;
2. the predicted target attribute whose values have been predicted by the classification tree; its name coincides with the name of the real attribute, but with extension `_predicted`.

KDD phase

Preprocessing.

Signature

$f_{\texttt{MISCLASSIFIED}} : \texttt{table} \longrightarrow \texttt{table}$

Required attributes

None.

Optional Attributes

- `xml_dest`.

3.2.16 PAR_QUERY

DTD

```
<!ELEMENT PAR_QUERY (%kdd_operator;,(%kdd_operator;)+)>
```

[3]In a misclassified instance, the value of the classification attribute and the value of the predicted attribute differ.

Description

The PAR_QUERY element models potential parallelism between KDDML operators. The return value of the PAR_QUERY is assumed to be the last operator in the sequence of their arguments. Note that with this assumption, PAR_QUERY is functionally equivalent to SEQ_QUERY (see sect. 3.2.57), and then it can be implemented as a sequential operator when physical parallelism is not available. Currently KDDML version do not support explicit parallelism.

KDD phase

Control flow.

Signature

$f_{\texttt{<PAR\_QUERY>}} : \texttt{any} \times \cdots \times \texttt{any} \rightarrow \texttt{any}.$

Required attributes

None.

Optional attributes

None.

3.2.17 PMML_CLUSTER_LOADER

DTD

```
<!ELEMENT PMML_CLUSTER_LOADER EMPTY>
<!ATTLIST PMML_CLUSTER_LOADER xml_dest %string; #IMPLIED>
<!ATTLIST PMML_CLUSTER_LOADER pmml_source %string; #REQUIRED>
```

Description

It loads an external PMML model containing a cluster model. At the present, KDDML supports the PMML 2.0 version for a cluster model.

KDD phase

Resource loading.

Signature

$f_{\texttt{<PMML\_CLUSTER\_LOADER>}} : empty \rightarrow \texttt{cluster}.$

Required attributes

- `pmml_source`: the external PMML source. PMML file can be gathered from the local file system (e.g. *D:/MyRepository/weather.xml*) or from web, using the ftp, http protocols (e.g. *ftp://www.foo.edu/PMML/weather.xml*).

Optional attributes

- `xml_dest`.

3.2.18 PMML_RDA_LOADER

DTD

```
<!ELEMENT PMML_RDA_LOADER EMPTY>
<!ATTLIST PMML_RDA_LOADER xml_dest %string; #IMPLIED>
<!ATTLIST PMML_RDA_LOADER pmml_source %string; #REQUIRED>
```

Description

It loads an external PMML model containing a set of association rules. At the present, KDDML supports the PMML 2.0 version for association rules.

KDD phase

Resource loading.

Signature

$f_{\texttt{<PMML\_RDA\_LOADER>}} : empty \rightarrow \texttt{rda}$.

Required attributes

- `pmml_source`: the external PMML source. PMML file can be gathered from the local file system (e.g. *D:/MyRepository/weather.xml*) or from web, using the ftp, http protocols (e.g. *ftp://www.foo.edu/PMML/weather.xml*).

Optional attributes

- `xml_dest`.

3.2.19 PMML_SEQUENCE_LOADER

DTD

```
<!ELEMENT PMML_SEQUENCE_LOADER EMPTY>
<!ATTLIST PMML_SEQUENCE_LOADER xml_dest %string; #IMPLIED>
<!ATTLIST PMML_SEQUENCE_LOADER pmml_source %string; #REQUIRED>
```

Description

It loads an external PMML model containing a sequence model. At present, KDDML supports the PMML 2.0 version for sequential patterns.

KDD phase

Resource loading.

Signature

$f_{<\texttt{PMML\_SEQUENCE\_LOADER}>} : empty \rightarrow \texttt{sequence}$.

Required attributes

- `pmml_source`: the external PMML source. The PMML file can be gathered either from the local file system (e.g. *D:/MyRepository/weather.xml*) or from the web, by using the ftp, http protocols (e.g. *ftp://www.foo.edu/PMML/weather.xml*).

Optional attributes

- `xml_dest`.

3.2.20 PMML_TREE_LOADER

DTD

```
<!ELEMENT PMML_TREE_LOADER EMPTY>
<!ATTLIST PMML_TREE_LOADER xml_dest %string; #IMPLIED>
<!ATTLIST PMML_TREE_LOADER pmml_source %string; #REQUIRED>
```

Description

It loads an external PMML model containing a tree model. At present, KDDML supports the PMML 2.0 version for a tree model.

KDD phase

Resource loading.

Signature

$f_{\texttt{<PMML\_TREE\_LOADER>}} : empty \rightarrow$ `tree`.

Required attributes

- `pmml_source`: the external PMML source. PMML file can be gathered from the local file system (e.g. *D:/MyRepository/weather.xml*) or from web, using the ftp, http protocols (e.g. *ftp://www.foo.edu/PMML/weather.xml*).

Optional attributes

- `xml_dest`.

3.2.21 PP_ADD_HIERARCHY

DTD

```
<!ELEMENT PP_ADD_HIERARCHY ((%kdd_query_PPtable;),
                            (%kdd_query_hierarchy;))>
<!ATTLIST PP_ADD_HIERARCHY xml_dest %string; #IMPLIED>
<!ATTLIST PP_ADD_HIERARCHY attribute_name %string; #REQUIRED>
```

Description

The operator assigns an hierarchy object to a table column as meta-data information. This operator works only on data schemata: neither the physical records, nor the preprocessing section are affected.

KDD phase

Preprocessing.

Signature

$f_{\texttt{<PP\_ADD\_HIERARCHY>}} : \texttt{PPtable} \times \texttt{hierarchy} \rightarrow \texttt{PPtable}$.

Required attributes

- `attribute_name`: the name of the input table attribute. The logical type of the attribute must be nominal or string.

Optional Attributes

- `xml_dest`.

3.2.22 PP_CHANGE_TYPE

DTD

```
<!ELEMENT PP_CHANGE_TYPE ((%kdd_query_PPtable;))>
<!ATTLIST PP_CHANGE_TYPE xml_dest %string; #IMPLIED>
<!ATTLIST PP_CHANGE_TYPE attributes_list %string; #REQUIRED>
<!ATTLIST PP_CHANGE_TYPE new_type (numeric|string|nominal) #REQUIRED>
```

Description

The operator changes the logical type of one or more attributes. The following conversions are allowed:

- from numeric attributes to nominal or string attributes;
- from nominal attributes to numeric or string attributes;
- from string attributes to numeric or nominal attributes.

Notice that the conversion from nominal or string to a numeric type succeeds only if the input attribute values are effectively numbers (see also the PP_NUMERIC_LABELING in section 3.2.33). This operator works only on data schemata: neither the physical records, nor the preprocessing section are affected.

KDD phase

Preprocessing.

Signature

$f_{<\mathrm{PP\_CHANGE\_TYPE}>}$: `PPtable` $\longrightarrow$ `PPtable`.

Required attributes

- `attributes_list`: the list of attributes of the input table whose type must be changed. The list is given in a comma separated format (e.g. outlook, play).
- `new_type`: the conversion type. Can be *numeric*, *string* or *nominal*.

Optional Attributes

- `xml_dest`.

3.2.23 PP_DIVIDING_ATTRIBUTE

DTD

```
<!ELEMENT PP_DIVIDING_ATTRIBUTE (%kdd_query_PPtable;)>
<!ATTLIST PP_DIVIDING_ATTRIBUTE xml_dest %string; #IMPLIED>
<!ATTLIST PP_DIVIDING_ATTRIBUTE source_attribute_name %string; #REQUIRED>
<!ATTLIST PP_DIVIDING_ATTRIBUTE
                         destination_attribute_name_1 %string; #REQUIRED>
<!ATTLIST PP_DIVIDING_ATTRIBUTE
                         destination_attribute_name_2 %string; #REQUIRED>
<!ATTLIST PP_DIVIDING_ATTRIBUTE regular_expression %string; #REQUIRED>
<!ATTLIST PP_DIVIDING_ATTRIBUTE remove_source_attribute (true|false) "false">
<!ATTLIST PP_DIVIDING_ATTRIBUTE mark_exception %string; #FIXED "no_RE_match">
```

Description

Given a preprocessing table, the operator partitions the values of an input attribute into two new string attributes. The splitting point is computed on the basis of a regular expression. A regular expression is a set of characters that determines a pattern or a template used to match a string.
For each instance, the operator attempts to match the entire attribute value against the pattern.
If the pattern matching succeeds, then the first occurrence matching the pattern is the sub-string to be assigned to the first output attribute. The rest of the input value becomes the sub-string to be assigned to the second output attribute.
If the attribute value is missing for an instance, or if the pattern matching fails, the correspondent preprocessing value for that instance is marked with the exception string `no_RE_match`. In this case, output destination attributes will be assigned to a missing value for that instance.
Destination attributes are added by the operator at the end of the data schema with a string type.

KDD phase

Preprocessing.

Signature

$f_{<\texttt{PP\_DIVIDING\_ATTRIBUTE}>} : \texttt{PPtable} \longrightarrow \texttt{PPtable}.$

Required attributes

- `source_attribute_name`: the source attribute name.
- `destination_attribute_name_1`: the first destination attribute name.
- `destination_attribute_name_2`: the second destination attribute name.
- `regular_expression`: the regular expression to be used to find the splitting point.

Optional Attributes

- `xml_dest`.
- `remove_source_attribute`: can be *true* or *false*. If the first case, the input source attribute will be removed from the output table. By default, the attribute value is *false*.

3.2.24 PP_FILTER_ATTRIBUTES

DTD

```
<!ELEMENT PP_FILTER_ATTRIBUTES ((%kdd_query_PPtable;))>
<!ATTLIST PP_FILTER_ATTRIBUTES attributes_list %string_list; #REQUIRED>
<!ATTLIST PP_FILTER_ATTRIBUTES take_or_remove (take|remove) "take">
<!ATTLIST PP_FILTER_ATTRIBUTES xml_dest %string; #IMPLIED>
```

Description

Select (or remove) a list of attributes from the input preprocessing table.

KDD phase

Preprocessing.

Signature

$f_{<\texttt{PP\_FILTER\_ATTRIBUTE}>} : \texttt{PPtable} \longrightarrow \texttt{PPtable}$.

Required attributes

- `attributes_list`: the set of attributes to be selected (or removed). The list of attributes is given in a comma separated format (e.g. *outlook, play, windy*).

Optional Attributes

- `xml_dest`.
- `take_or_remove`: can be *take* or *remove*. In the first case, the resultant table will be composed only by the list of provided attributes. Otherwise, the specified attributes will be removed from the input data source. By default, the attribute assumes a *take* value.

3.2.25 PP_FOLDING

DTD

```
<!ELEMENT PP_FOLDING ((%kdd_query_PPtable;))>
<!ATTLIST PP_FOLDING xml_dest %string; #IMPLIED>
<!ATTLIST PP_FOLDING source_attribute_name %string; #REQUIRED>
<!ATTLIST PP_FOLDING destination_attribute_name %string; #REQUIRED>
```

Description

The operator removes from the input preprocessing source the column of a specified attribute, and it copies its values into a destination attribute. Every instance is replicated and every pair of instances can be distinguished only for the destination attribute values. As a consequence, if the input data source is an *N x M* table, the operator returns a new preprocessing table with size *2N X M-1*. Also the preprocessing section is replicated.
The input source attribute and the destination attribute must share the same type in the input table.

KDD phase

Preprocessing.

Signature

$f_{\texttt{<PP\_FOLDING>}}$: `PPtable` $\longrightarrow$ `PPtable`.

Required attributes

- `source_attribute_name`: the source attribute name of the input table.
- `destination_attribute_name`: the destination attribute name of the input table containing the folding.

Optional Attributes

- `xml_dest`.

3.2.26 PP_HIERARCHICAL_DISCRETIZATION

DTD

```
<!ELEMENT PP_HIERARCHICAL_DISCRETIZATION ((%kdd_query_PPtable;),
                                          (%kdd_query_hierarchy;))>
<!ATTLIST PP_HIERARCHICAL_DISCRETIZATION xml_dest %string; #IMPLIED>
<!ATTLIST PP_HIERARCHICAL_DISCRETIZATION attribute_name %string; #REQUIRED>
<!ATTLIST PP_HIERARCHICAL_DISCRETIZATION level %integer; #REQUIRED>
<!ATTLIST PP_HIERARCHICAL_DISCRETIZATION mark_exception %string;
                                      #FIXED "no_hierarchy_generalization">
```

Description

A categoric discretization of an attribute with respect to a hierarchy is performed. Given a non-numeric attribute of the input preprocessing table, a concept hierarchy related to the attribute values, and an integer representing the hierarchy level, the operator returns a new table containing generalized values for that attribute.
The mapping between attribute values and leaf values of the hierarchy is by name.
If the attribute value is found in a leaf of the hierarchy, then the generalized node value is computed at the specified level. In this case, the old attribute value is replaced with the generalized item that has been found.
If some value cannot be found in the hierarchy leaves or if the generalization cannot be computed, then the attribute is set with a missing value for that instance. Also the correspondent preprocessing information is marked with the exception `no_hierarchy_generalization`.
Obviously, the level of the hierarchy must be compatible with the depth of the hierarchy in order to compute the generalization.

KDD phase

Preprocessing.

Signature

$f_{<\texttt{PP\_HIERARCHICAL\_DISCRETIZATION}>} : \texttt{PPtable} \times \texttt{hierarchy} \rightarrow \texttt{PPtable}$.

Required attributes

- `attribute_name`: the attribute to which the categoric discretization is applied. The attribute must be string or nominal.
- `level`: a positive integer representing the level of generalization. The level must be less than the depth of the hierarchy. We assume that the leaves of the hierarchy have a level 0, while the root has a level equal to the depth of the hierarchy.

Optional Attributes

- `xml_dest`.

3.2.27 PP_MARKING

DTD

```
<!ELEMENT PP_MARKING ((%kdd_query_PPtable;), CONDITION)>
<!ATTLIST PP_MARKING xml_dest %string; #IMPLIED>
<!ATTLIST PP_MARKING attribute_name %string; #REQUIRED>
<!ATTLIST PP_MARKING mark %string; #REQUIRED>
```

Description

The operator marks the preprocessing values of an attribute according to a condition expressed on table attributes and/or constants.
For each instance of the input table, the condition is evaluated.
If the condition is satisfied, then the input attribute of that instance is marked (i.e. a value is added to preprocessing information) with a specified value.
If the condition evaluation returns false, no operation is performed.
This operator works only on the preprocessing section of the input table. In other terms, the physical data will not be changed.

KDD phase

Preprocessing

Signature

$f_{\texttt{<PP\_MARKING>}}$: $\texttt{PPtable} \times \texttt{condition} \longrightarrow \texttt{PPtable}$.

Required attributes

- `attribute_name`: the name of marking attribute.
- `mark`: the string value to be added to preprocessing section of the attribute.

Optional attributes

- `xml_dest`.

Condition specification

In reference to the figure 2.36, legal values for the XML attribute `op_type` are reported below:

- **equal**, **not_equal**, **greater**, **greater_or_equal**, **less**, **less_or_equal**: used for relational expressions on attribute values and/or constants (e.g. *"to mark all instances with the attribute* `temperature` *less than 80"*).
- **is_missing**: unary boolean expression evaluated on a single table attribute. Its evaluation is satisfied only if the attribute value for an instance is missing (e.g. *"to mark all instances where the attribute* `outlook` *is missing"*).

The table 3.8 contains the types of primitive terms (XML attributes `term1, term2, term3`) to be used for each `op_type` legal value.

OpType	Term 1 type	Term 2 type	Term 3 type
equal, **not_equal**	@attribute	@attribute numeric or string constant	-
greater, **greater_or_equal** **less**, **less_or_equal**	@numeric_attribute	@numeric_attribute numeric constant	-
is_missing	@attribute	-	-

Table 3.8: Element BASE_COND for PP_MARKING operator

3.2.28 PP_MARK_DUPLICATES

DTD

```
<!ELEMENT PP_MARK_DUPLICATES ((%kdd_query_PPtable;))>
<!ATTLIST PP_MARK_DUPLICATES attributes_list %string_list; #IMPLIED>
<!ATTLIST PP_MARK_DUPLICATES xml_dest %string; #IMPLIED>
<!ATTLIST PP_MARK_DUPLICATES mark %string; "duplicate">
```

Description

The operator marks duplicated instances. Two instances are considered duplicates on the basis of a key composed by a list of attributes. As an example, consider the attributes `temperature` and `outlook` as keys. In this case, two instances are duplicates if they have the same values for those attributes.
When two instances are selected as duplicates, all key attributes are marked (i.e. a string is added to the preprocessing information) with a specified value.
Notice that the operator only marks duplicates and the value of the mark is inserted as preprocessing information of the output table. In other words, no physical data are affected by the operator.
Duplicated instances can be joined using the PP_MERGE_DUPLICATES operator (see sect. 3.2.29).

KDD phase

Preprocessing.

Signature

$f_{<\texttt{PP\_MARK\_DUPLICATES}>} : \texttt{PPtable} \rightarrow \texttt{PPtable}$.

Required attributes

None.

Optional Attributes

- `xml_dest`.

- `attributes_list`: the list of attributes of the input table representing the key. The list of attributes is given in a comma separated format (e.g. *outlook*, *temperature*). The attribute is optional. If it is omitted, then all the columns of the input preprocessing table will be considered as composing the key.
- `mark`: the mark that must be added to the preprocessing section of key attributes when two instances are duplicates. By default, it assumes the value `duplicate`.

3.2.29 PP_MERGE_DUPLICATES

DTD

```
<!ELEMENT PP_MERGE_DUPLICATES ((%kdd_query_PPtable;))>
<!ATTLIST PP_MERGE_DUPLICATES attributes_list %string_list; #IMPLIED>
<!ATTLIST PP_MERGE_DUPLICATES xml_dest %string; #IMPLIED>
<!ATTLIST PP_MERGE_DUPLICATES mark %string; "merged_duplicates">
```

Description

The operator finds and unifies instances that are duplicates. Two instances are duplicates on the basis of a key composed by a list of attributes. As an example, consider the attributes `temperature` and `outlook` as the key. In this case, two instances are duplicates if they have the same values for those attributes.
When two or more instances have been selected as duplicates, the operator chooses[4] only one instance as the representant. All the other instances are removed from the input preprocessing table. Finally, all key attributes of this instance are marked (i.e. a string is added to preprocessing information) with a specified value.

KDD phase

Preprocessing.

Signature

$f_{\texttt{<PP\_MERGE\_DUPLICATES>}}$: `PPtable` $\longrightarrow$ `PPtable`.

Required attributes

None.

[4]In the current version, the operator uses a random policy to choose the representant of a set of duplicates.

Optional Attributes

- `xml_dest`.
- `attributes_list`: the list of attributes of the input table representing the key. The list of attributes is given in a comma separated format (e.g. *outlook*, *temperature*). The attribute is optional. If it is omitted, then all the columns of the input preprocessing table will be considered as forming the key.
- `mark`: the mark that must be added to the preprocessing section of key attributes when an instance is selected as representant. By default, it assumes the value `merged_duplicate`.

3.2.30 PP_NEW_ATTRIBUTE

DTD

```
<!ELEMENT PP_NEW_ATTRIBUTE ((%kdd_query_PPtable;),(EXPRESSION)?)>
<!ATTLIST PP_NEW_ATTRIBUTE xml_dest %string; #IMPLIED>
<!ATTLIST PP_NEW_ATTRIBUTE attribute_name %string; #REQUIRED>
<!ATTLIST PP_NEW_ATTRIBUTE position %integer; #IMPLIED>
<!ATTLIST PP_NEW_ATTRIBUTE attribute_type (string|numeric)
#REQUIRED>
```

Description

The operator adds a new attribute to the input preprocessing table. The attribute values can be derived from existing ones by means of a simple expression language, as reported in figure 2.37. Also the type of the derived attribute and his position in the data schema can be specified.
The sub-element `EXPRESSION` is optional. If no expression is provided, all the values become missing for the new attribute.

KDD phase

Preprocessing.

Signature

$f_{<\texttt{PP\_NEW\_ATTRIBUTE}>} : \texttt{PPtable} \times \texttt{expr} \longrightarrow \texttt{PPtable}.$

Required attributes

- `attribute_name` the attribute to be added to the preprocessing table.
- `attribute_type` the type of derived attribute. The type can be `numeric` or `string`.

Optional Attributes

- `xml_dest`.
- `position`: a non-negative integer representing the position of the new attribute in the resulting table. If it is omitted, the column will be added to the end of the schema. The first attribute starts from 0 in the schema section.

3.2.31 PP_NORMALIZATION

DTD

```
<!ELEMENT PP_NORMALIZATION ((%kdd_query_PPtable;),ALGORITHM)>
<!ATTLIST PP_NORMALIZATION xml_dest %string; #IMPLIED>
<!ATTLIST PP_NORMALIZATION attributes_list %string_list; #REQUIRED>
```

Description

Normalization is the process of scaling data values of a numeric attribute to fit in a range such as [-1, 1] or [0,1]. Normalization is particularly useful for classification algorithms and clustering. The operator takes a preprocessing table, a list of numeric attributes, and a normalization method and returns a new preprocessing table with normalized values for each specified attribute. There are many methods for data normalization, each of them expressed by using the XML element `ALGORITHM` (see figure 2.35). In section 4.1.2, the list of supported normalization algorithms is reported.

KDD phase

Preprocessing.

Signature

$f_{<\texttt{PP\_NORMALIZATION}>} : \texttt{PPtable} \times \texttt{alg} \longrightarrow \texttt{PPtable}$.

Required attributes

- `attributes_list`: the list of attributes of the input table to normalize. The list of attributes is given in a comma separated format (e.g. *humidity, temperature*). Normalization methods are allowed only on numeric attributes.

Optional attributes

- `xml_dest`.

3.2.32 PP_NUMERIC_DISCRETIZATION

DTD

```
<!ELEMENT PP_NUMERIC_DISCRETIZATION ((%kdd_query_PPtable;),ALGORITHM)>
<!ATTLIST PP_NUMERIC_DISCRETIZATION xml_dest %string; #IMPLIED>
<!ATTLIST PP_NUMERIC_DISCRETIZATION attribute_name %string; #REQUIRED>
```

Description

Discretization techniques can be used to reduce the number of values for a given continuous attribute, by dividing the range of the attribute into intervals.
The operator takes a preprocessing table, a numeric attribute and a discretization method and returns a new preprocessing table with discretized values for the specified attribute. There are many methods for data discretization, each of them expressed by using the XML element `ALGORITHM` (see figure 2.35). In section 4.1.1, the list of supported discretization algorithms is reported.

KDD phase

Preprocessing.

Signature

$f_{\texttt{<PP\_NUMERIC\_DISCRETIZATION>}} : \texttt{PPtable} \times \texttt{alg} \longrightarrow \texttt{PPtable}$.

Required attributes

- `attribute_name`: the name of the attribute to discretize. Numeric discretization can be applied only on continuous attributes.

Optional Attributes

- `xml_dest`.

3.2.33 PP_NUMERIC_LABELING

DTD

```
<!ELEMENT PP_NUMERIC_LABELING ((%kdd_query_PPtable;))>
<!ATTLIST PP_NUMERIC_LABELING xml_dest %string; #IMPLIED>
<!ATTLIST PP_NUMERIC_LABELING attribute_name %string; #REQUIRED>
<!ATTLIST PP_NUMERIC_LABELING numeric_values_list %real_list; #IMPLIED>
```

Description

The operator is used to convert a nominal attribute into a numeric attribute by using its natural rank, as implied by the header specification, or by using the list of numeric labels provided as XML attribute.
The natural rank is given by the position in which the single categories occur in the nominal definition contained in the data schema. In this case, the labeling technique assigns 0 to the first category, 1 to the second category and so on. As an example, consider the nominal attribute `outlook` defined as:

@attribute outlook $\{$*sunny, overcast, rainy*$\}$.

When a *sunny* value is found in an instance, it is replaced with the number 0. Analogously, numbers 1 and 2 are used for *overcast* and *rainy* values respectively.
As an alternative to the natural rank, a list of continuous values can be specified as XML attribute. This operator is usually applied to discretize nominal attributes, converting they into numeric attributes and applying then the PP_NUMERIC_DISCRETIZATION operator (see sect. 3.2.32).

KDD phase

Preprocessing.

Signature

$f_{\texttt{<PP\_NUMERIC\_LABELING>}} : \texttt{PPtable} \rightarrow \texttt{PPtable}$.

Required attributes

- `attribute_name`: the nominal attribute of the input preprocessing table.

Optional Attributes

- `xml_dest`.
- `numeric_value_list`: the list of numbers used to map the categories of the nominal attribute. The list is given in a comma separated format and the number of values must be equal to the number of categories for that attribute. The order of labeling is given by the order in which the categories appears in the nominal definition. If the attribute is omitted, the operator will assign 0 to the first category, 1 to the second category, and so on.

3.2.34 PP_REMOVE_ROWS

DTD

```
<!ELEMENT PP_REMOVE_ROWS ((%kdd_query_PPtable;), CONDITION)>
<!ATTLIST PP_REMOVE_ROWS xml_dest %string; #IMPLIED>
```

Description

The operator deletes rows (physical records and preprocessing information) from an input preprocessing table according to a specific condition.

KDD phase

Preprocessing.

Signature

$f_{<\mathtt{PP\_REMOVE\_ROWS}>} : \mathtt{PPtable} \times \mathtt{cond} \longrightarrow \mathtt{PPtable}$.

Required attributes

None.

Optional Attributes

- `xml_dest`.

Condition specification

In reference to the figure 2.36, legal values for the XML attribute `op_type` are reported below:

- **equal**, **not_equal**, **greater**, **greater_or_equal**, **less**, **less_or_equal**: used for relational expressions on attributes values and/or constants (e.g. *"to remove all instances with the attribute* `temperature` *less than 80"*).
- **is_missing**: unary boolean expression evaluated on a single table attribute. Its evaluation is satisfied only if the attribute value for an instance is missing (e.g. *"to remove all instances where the attribute* `outlook` *is missing"*).
- **has_exception**, **has_mark**: binary expressions evaluated on preprocessing section of input table. In particular, they test if an attribute has a mark (exception) equal to a specified value (e.g. *"to remove rows if the attribute* `temperature` *contains the mark out_of_range"*).

The table 3.9 contains the types of primitive terms (XML attributes `term1, term2, term3`) to be used for each `op_type` legal value.

3.2.35 PP_RENAME_ATTRIBUTES

DTD

```
<!ELEMENT PP_RENAME_ATTRIBUTES ((%kdd_query_PPtable;))>
<!ATTLIST PP_RENAME_ATTRIBUTES xml_dest %string; #IMPLIED>
<!ATTLIST PP_RENAME_ATTRIBUTES old_attributes_list %string_list; #REQUIRED>
<!ATTLIST PP_RENAME_ATTRIBUTES new_attributes_list %string_list; #REQUIRED>
```

OpType	Term 1 type	Term 2 type	Term 3 type
equal, **not_equal**	@attribute	@attribute numeric or string constant	-
greater, **greater_or_equal** **less**, **less_or_equal**	@numeric_attribute	@numeric_attribute numeric constant	-
is_missing	@attribute	-	-
has_mark, **has_exception**	@attribute	constant	-

Table 3.9: Element BASE_COND for the PP_REMOVE_ROWS operator

Description

The operator renames a list of attributes of the input data source. The list of old attributes and the list of new attributes can be given via XML attributes; the operator performs a position mapping between values. This operator works only on data schema: neither the physical records, nor the preprocessing section are affected.

KDD phase

Preprocessing.

Signature

$f_{\texttt{<PP\_RENAME\_ATTRIBUTES>}}$: `PPtable` $\longrightarrow$ `PPtable`.

Required attributes

- `old_attributes_list` the list of attributes of the input table to change. The list of attributes is in a comma separated format (e.g. *outlook*, *temperature*, *play*).
- `new_attributes_list` the list of strings containing the new attributes names. The list is given in a comma separated format (e.g. *new_outlook*, *new_temperature*, *new_play*) and a position mapping with the values of `old_attributes_list` is performed.

Optional Attributes

- `xml_dest`.

3.2.36 PP_REWRITING

DTD

```
<!ELEMENT PP_REWRITING ((%kdd_query_PPtable;),(ALGORITHM, CONDITION)+)>
<!ATTLIST PP_REWRITING xml_dest %string; #IMPLIED>
<!ATTLIST PP_REWRITING attribute_name %string; #REQUIRED>
```

Description

The operator rewrites the values of an input table attribute. It is a construct based on regular expressions that are a powerful way to specify string matching and the substitution of some pattern with new values.
The operator takes a preprocessing table, a table attribute and a set of pairs containing a condition (element `CONDITION`) and a rewriting method (element `ALGORITHM`). It returns a new preprocessing table with rewritten values for the specified attribute.
There are many methods for data rewriting, each of them expressed by using the XML element `ALGORITHM` (see figure 2.35). In section 4.1.3, the list of supported rewriting algorithms will be reported.
The condition regards restrictions on the rows to be rewritten. In other words, the corresponding rewriting rule is applied only on instances on which the condition is satisfied. By using this strategy, more rewriting algorithm can be used on a single record. Each condition provides a constraint on these algorithms.
In general, every rewriting algorithm is characterized by three features:

1. the regular expression used to match the input value;
2. the replacement policy to apply when the matching succeeds (for example, it is possible to replace all string value or only some occurrences satisfying the pattern matching);
3. the policy to apply when the matching procedure fails (typically, a preprocessing marking procedure is applied in this case).

The core procedure of the operator is reported in Alg. 1.

Algorithm 1 The core procedure of the `PP_REWRITING` operator

```
Require: Instances: instances, a list of couples (Algorithm: algorithm, Condition: condition)
  for all inst in instances do
    for all (alg, cond) in (algorithm, condition) do
      if (cond.evaluateCond(inst)) then
        if alg.match(inst.getValue(attribute_name), alg.regular_expression)) then
          inst.setValue(attribute_name, alg.applyReplacementPolicy())
        else
          alg.applyMarkingPolicy(inst, attribute_name)
        end if
      end if
    end for
  end for
```

The type of the rewriting attribute is preserved. Run-time checking is needed on new values for numeric or nominal attributes. In particular, nominal categories for an enumerated attribute must be preserved after rewriting.

KDD phase

Preprocessing.

Signature

$f_{\texttt{<PP\_REWRITING>}} : \texttt{PPtable} \times (\texttt{alg} \times \texttt{cond})+ \longrightarrow \texttt{PPtable}.$

Required attributes

- `attribute_name`: the rewriting attribute of the input preprocessing table.

Optional Attributes

- `xml_dest`.

Condition specification

In reference to the figure 2.36, legal values for the XML attribute `op_type` are reported below:

- **equal**, **not_equal**, **greater**, **greater_or_equal**, **less**, **less_or_equal**: used for relational expressions on attribute values and/or constants (e.g. *"to mark all instances with the attribute* `temperature` *less than 80"*).
- **is_missing**: unary boolean expression evaluated on a single table attribute. Its evaluation is satisfied only if the attribute value for an instance is missing (e.g. *"to mark all instances where the attribute* `outlook` *is missing"*).
- **group**: given an attribute value previously matched with a regular expression, a positive integer N representing the index of a sub-sequence (group) related to pattern matching, and a string constant, it checks if the group N of the attribute value is equal to the input string. As a matter of notation, sub-sequences can be identified in the regular expression by grouping them within round parentheses.

The table 3.10 contains the types of primitive terms (XML attributes `term1, term2, term3`) to be used for each `op_type` legal value.

OpType	Term 1 type	Term 2 type	Term 3 type
equal, **not_equal**	@attribute	@attribute numeric or string constant	-
greater, **greater_or_equal** **less**, **less_or_equal**	@numeric_attribute	@numeric_attribute numeric constant	-
is_missing	@attribute	-	-
group	positive integer	string constant	-

Table 3.10: Element BASE_COND for the PP_REWRITING operator

3.2.37 PP_SAMPLING

DTD

```
<!ELEMENT PP_SAMPLING ((%kdd_query_PPtable;), ALGORITHM)>
<!ATTLIST PP_SAMPLING xml_dest %string; #IMPLIED>
```

Description

Data sampling techniques can be used to obtain a reduced representation of the data set, such that it is a much smaller random sample, yet closely maintaining the integrity of the original data.
The operator takes a preprocessing table and a sampling method, and returns a new preprocessing table whose instances have been selected according to the specified procedure. There are many methods for data sampling, each of them expressed using the XML element `ALGORITHM` (see figure 2.35). In section 4.1.4, the list of supported sampling algorithms is reported.

KDD phase

Preprocessing.

Signature

$f_{\texttt{<PP\_SAMPLING>}} : \texttt{PPtable} \times \texttt{alg} \longrightarrow \texttt{PPtable}.$

Required attributes

None.

Optional Attributes

- `xml_dest`.

3.2.38 PP_SORTING_ATTRIBUTE

DTD

```
<!ELEMENT PP_SORTING_ATTRIBUTE ((%kdd_query_PPtable;))>
<!ATTLIST PP_SORTING_ATTRIBUTE xml_dest %string; #IMPLIED>
<!ATTLIST PP_SORTING_ATTRIBUTE attribute_name %string; #REQUIRED>
<!ATTLIST PP_SORTING_ATTRIBUTE sorting_type
                     (by_frequency|ascending|descending) "by_frequency">
```

Description

The operator sorts the values of a given attribute according to a sorting method, as reported below:

- *ascending* (*descending*) ordering of a numeric, string or nominal attribute;

- *by frequency* ordering used on nominal attributes only. By using this strategy, the attribute is ordered with respect to the frequency of the categories (i.e. the number of elements for that category occurring in the input table). In this case, the operator adds a new numeric attribute with the name of the input attribute followed by `_frequency` at the end of the data schema. This new attribute will contain the frequency values for each nominal category.

KDD phase

Preprocessing.

Signature

$f_{\texttt{<PP\_SORTING\_ATTRIBUTE>}}$: `PPtable` $\rightarrow$ `PPtable`.

Required attributes

- `attribute_name`: the name of the sorting attribute. If the sorting method is *by_frequency*, then the attribute must be nominal.

Optional Attributes

- `xml_dest`.
- `sorting_type`: the sorting procedure to be used. Possible values are *ascending*, *descending* or *by_frequency*. By default, the operator uses the *by_frequency* ordering.

3.2.39 PP_TABLE_2_TABLE

DTD

```
<!ELEMENT PP_TABLE_2_TABLE (%kdd_query_PPtable;)>
<!ATTLIST PP_TABLE_2_TABLE xml_dest %string; #IMPLIED>
```

Description

The operator finalizes the preprocessing KDD phase by mapping the input preprocessing table into a relational table. Input and output tables share the same data schema and the same physical instances.

KDD phase

Preprocessing.

Signature

$f_{\texttt{PP\_TABLE\_TO\_TABLE}}$: `PPtable` $\rightarrow$ `table`

Required attributes

None.

Optional Attributes

- `xml_dest`.

3.2.40 PP_TABLE_LOADER

DTD

```
<!ELEMENT PP_TABLE_LOADER EMPTY>
<!ATTLIST PP_TABLE_LOADER xml_source %string; #REQUIRED>
```

Description

It loads a preprocessing table from the system repository.

KDD phase

Resource loading.

Signature

$f_{<\texttt{PP\_TABLE\_LOADER}>} : empty \rightarrow \texttt{PPtable}$.

Required attributes

- `xml_source`: the XML file source contained in the data repository.

Optional attributes

None.

3.2.41 RDA_EXCEPTION

DTD

```
<!ELEMENT RDA_EXCEPTION ((%kdd_query_rules;),
                         (%kdd_query_table;))>
<!ATTLIST RDA_EXCEPTION xml_dest %string; #IMPLIED>
<!ATTLIST RDA_EXCEPTION itemsets_or_rules (itemsets|rules) "rules">
```

Description

Given a set of association rules and a table, the operator extracts the transactions in the table that are exceptions to all the association rules.
A transaction is an exception to an association rule $I_1, \ldots, I_n \rightarrow I_{n+1}, \ldots, I_m$ if some item I_i for $i \in [1, m]$ does not occur in the transaction. For example, the transaction *T* = $\{$*bread, milk, wine*$\}$ satisfies the rule *bread* → *wine*, but it is an exception to the rule *bread, milk* → *water*.
The operator can be restricted to itemsets only. In this case, we say that a transaction is an exception to an itemset $I_1, \ldots, I_n$ if some item I_i for $i \in [1, n]$ does not occur in the transaction.
In both cases, the input relational table must be compatible with the association rules. In particular, a table is compatible with a model if for each active mining field, belonging to the model, there is an attribute in the table with the same name and type.

KDD phase

Model application.

Signature

$f_{\texttt{<RDA\_EXCEPTION>}}$: `rda` × `table` → `table`.

Required attributes

None.

Optional Attributes

- `xml_dest`.
- `itemsets_or_rules`: can be *itemsets* or *rules*. In the first case, the operator uses the itemsets to evaluate the transaction exceptions. In the second case, the operator uses the association rules for the same purpose. By default, the value is *rules*.

3.2.42 RDA_FILTER

DTD

```
<!ELEMENT RDA_FILTER ((%kdd_query_rules;), CONDITION)>
<!ATTLIST RDA_FILTER  xml_dest %string; #IMPLIED>
```

Description

Given a set of association rules, the operator returns the rules satisfying a specified condition. The condition is expressed by using an `AND/OR/NOT` combination between a set of primitive filters, concerning:

- the support or the cardinality of the itemsets;
- the support, the confidence or the cardinality of body/head of the association rules;
- the single items contained in the itemset, body or head elements.

KDD phase

Model filtering.

Signature

$f_{\texttt{<RDA\_FILTER>}}$: `rda` $\times$ `condition` $\longrightarrow$ `rda`.

Required attributes

None.

Optional attributes

- `xml_dest`.

Condition specification

In reference to the figure 2.36, legal values for the XML attribute `op_type` are reported below:

- **equal**, **not_equal**, **greater**, **greater_or_equal**, **less**, **less_or_equal**: used for relational expressions on support/confidence of the rules or itemsets (e.g. *"to filter association rules with a confidence less than 1"*) or on cardinality (i.e. the number of items) of itemsets, body or head elements (e.g. *"to filter rules with exactly 2 items in the body"*).
- **is_in**: it checks if an item belongs to an itemset, or to either the body or the head elements of an association rule (e.g. *"filter rules with the item 'milk' in the body"*).
- **is_not_in**: it checks if an item does not belong to itemset or to either the body or the head elements of an association rule.

The table 3.11 contains the types of primitive terms (XML attributes `term1, term2, term3`) to be used for each `op_type` legal value.

3.2.43 RDA_LOADER

DTD

```
<!ELEMENT RDA_LOADER EMPTY>
<!ATTLIST RDA_LOADER xml_source %string; #REQUIRED>
```

OpType	Term 1 type	Term 2 type	Term 3 type
equal, **not_equal** **greater**, **greater_or_equal** **less**, **less_or_equal**	@itemset_support @rda_support @rda_confidence	real in [0,1]	-
	@itemset_cardinality @body_cardinality @head_cardinality	positive integer	-
is_in, **is_not_in**	@itemset @body @head	string	-

Table 3.11: Element BASE_COND for the RDA_FILTER operator

Description

It loads a set of association rules from the system repository.

KDD phase

Resource loading.

Signature

$f_{<\texttt{RDA\_LOADER}>} : empty \longrightarrow \texttt{rda}$.

Required attributes

- `xml_source`: the XML file source contained in the models repository.

Optional attributes

None.

3.2.44 RDA_MINER

DTD

```
<!ELEMENT RDA_MINER ((%kdd_query_table;), ALGORITHM)>
<!ATTLIST RDA_MINER xml_dest %string; #IMPLIED>
```

Description

It extracts a set of association rules by using a mining algorithm. The operator takes a data source and an association rules miner algorithm, and it returns an association model as output.

The algorithm specification (i.e. the algorithm name and the list of expected parameters) is expressed by using the XML element `ALGORITHM` (see figure 2.35). In section 4.1.5, the list of supported association rules algorithms is reported.

KDD phase

Data mining.

Signature

$f_{\texttt{<RDA\_MINER>}} : \texttt{table} \times \texttt{alg} \rightarrow \texttt{rda}$.

Required attributes

None.

Optional attributes

- `xml_dest`.

3.2.45 RDA_PRESERVED

DTD

```
<!ELEMENT RDA_PRESERVED ((%kdd_query_hierarchy;),
                         (%kdd_query_rules;),
                         (%kdd_query_rules;)+)>
<!ATTLIST RDA_PRESERVED xml_dest %string; #IMPLIED>
```

Description

Given a hierarchy of items and two sets of association rules $\mathcal{R}_1, \mathcal{R}_2$ over items in the hierarchy, this operator selects those rules in $\mathcal{R}_1$ such that, by generalizing the items in the rule to the parent level, yields a rule that belongs to $\mathcal{R}_2$. For example, the two association rules reported below,

$R_1 =$*“LongBeach AND SanJose* $\rightarrow$ *Chicago“*,
$R_2 =$*“California* $\rightarrow$ *Illinois“*

are preserved over the *cities-states* hierarchy.
The first set of rules ($\mathcal{R}_1$) must have been extracted at the bottom level of the hierarchy (i.e. the leaves level). The second set of rules ($\mathcal{R}_1$) is a generalization at parent levels. Obviously, the two sets of association rules must share the same signature, i.e. the same mining schema.

KDD phase

Model meta-reasoning.

Signature

$f_{<\texttt{RDA\_PRESERVED}>}$: `hierarchy` $\times$ `rda` $\times$ `rda` $\longrightarrow$ `rda`.

Required attributes

None.

Optional attributes

- `xml_dest`.

3.2.46 RDA_SATISFY

DTD

```
<!ELEMENT RDA_SATISFY ((%kdd_query_rules;),
                        (%kdd_query_table;))>
<!ATTLIST RDA_SATISFY xml_dest %string; #IMPLIED>
<!ATTLIST RDA_SATISFY itemsets_or_rules (itemsets|rules) "rules">
```

Description

Given a set of association rules and a table, the operator extracts the transactions in the table that satisfy at least one association rule. A transaction satisfies an association rule $I_1, \ldots, I_n \rightarrow I_{n+1}, \ldots, I_m$ if every item I_i for $i \in [1, m]$ occurs in the transaction.
As for RDA_EXCEPTION, the input relational table must be compatible with the association model (see sect. 3.2.41).

KDD phase

Model application.

Signature

$f_{<\texttt{RDA\_SATISFY}>}$: `rda` $\times$ `table` $\longrightarrow$ `table`.

Required attributes

None.

Optional Attributes

- `xml_dest`.
- `itemsets_or_rules`: can be *itemsets* or *rules*. In the first case, the operator uses itemsets to evaluate the transaction exceptions. In the second case, the operator uses the association rules. By default, the value is *rules*.

3.2.47 SCALAR

DTD

```
<!ELEMENT SCALAR EMPTY>
<!ATTLIST SCALAR value %any_type; #REQUIRED>
<!ATTLIST SCALAR xml_dest %string; #IMPLIED>
```

Description

The operator returns a scalar value expressed through an XML attribute.

KDD phase

None.

Signature

$f_{\texttt{SCALAR}} : empty \rightarrow \texttt{scalar}$

Required attributes

- `value`: the scalar value to return.

Optional Attributes

- `xml_dest`.

3.2.48 SEQUENCE_AGGREGATE_FILTER

DTD

```
<!ELEMENT SEQUENCE_AGGREGATE_FILTER ((%kdd_query_sequence;),
                                     (%kdd_query_table;), CONDITION)>
<!ATTLIST SEQUENCE_AGGREGATE_FILTER  xml_dest %string; #IMPLIED>
```

Description

Given a set of sequential patterns and a time-stamp table, the operator returns the patterns satisfying a specific condition.
The condition concerns an *aggregate constraint*, that is a constraint on an aggregate of items in a pattern, where the aggregate function can be *sum, average, max, min, standard deviation* (e.g., *"return all sequences in which the average price of all items is over 100$"*). The values of the items of the sequence (e.g., the price of item in the example above) are contained in the input time-stamp table.
This operator is *support-related*, i.e. given a condition it is applied to check whether a sequence matches a transaction. To find whether a sequential pattern satisfies these constraints, one needs

to examine the sequence database. In other terms, a pattern keeps being frequent only if the number of data sequences (transactions) supporting it (see the SEQUENCE_SATISFY operator in sect. 3.2.55) and satisfying the condition is greater than the minimum support. The support threshold is commonly expressed in the model and it can be specified in terms of absolute count or percentage. The core procedure of the operator is reported in alg. 2.

Algorithm 2 The core procedure of the `SEQUENCE_AGGREGATE_FILTER` operator

```
Require: Sequences: sequences, Transactions: transactions, Condition: condition
  int absolute_supp = Transactions.num_trans * Sequences.min_supp;
  for all seq in sequences do
    seq.absolute_supp = 0;
    for all trans in transactions do
      if ((trans.satisfied(seq)) and (condition.evalCond(trans))) then
        seq.absolute_supp = seq.absolute_supp + 1;
      end if
    end for
    if seq.absolute_supp $<$ absolute_supp then
      sequences.remove(seq);
    end if
  end for
  return sequences;
```

As an example, consider the basket database reported in table 3.12 with three transactions. Item properties as the `price` and the `quantity` of purchased products are expressed as additional columns. Moreover, consider the sequential pattern $\{spaghetti\} \rightarrow \{spaghetti, wine\}$ and sup-

transaction	timestamp	event	price	quantity
id_1	10	spaghetti	3	5
id_1	15	tomato	2	2
id_1	15	wine	4	1
id_1	21	spaghetti	6	4
id_1	21	tomato	3	2
id_1	21	wine	7	1
id_2	12	mais	6	10
id_2	15	wine	8	1
id_3	15	spaghetti	4	3
id_3	15	milk	2	1
id_3	18	spaghetti	4	3
id_3	18	wine	2	1

Table 3.12: An example of market sequence dataset

pose we are interested only in the patterns in which the sum of prices of all items is greater or equal than 11\$. Transactions `id_1` and `id_3` satisfy the input sequence, while the transaction `id_2` does not. With respect to satisfied transactions, the condition $sum(@price) \geqslant 11$ is true only for transaction `id_1` ($sum(3,6,7) \geqslant 11$ but $sum(4,4,2) \not\geqslant 11$). So, if the minimum support of the input sequence model is less than 0.33, then the sequence is returned; otherwise, the sequence is filtered out from the model.

KDD phase

Model Filtering.

Signature

$f_{\texttt{<SEQUENCE\_AGGREGATE\_FILTER>}}$: `sequence` $\times$ `table` $\times$ `condition` $\rightarrow$ `sequence`.

Required attributes

None.

Optional attributes

- `xml_dest`.

Condition specification

In reference to the figure 2.36, legal values for the XML attribute `op_type` are reported below:

- **equal**, **not_equal**, **greater**, **greater_or_equal**, **less**, **less_or_equal**: they are used for relational expressions on aggregates of items in the patterns (e.g., *"return all the sequences in which the sum of quantities of all items is equal to 10"*). Values for aggregates of items of the sequence are contained in the columns of the input time-stamp table. They can be referred to by using the `term2` XML attribute.

The table 3.13 contains the types of primitive terms (XML attributes `term1, term2, term3`) to be used for each `op_type` legal value.

OpType	Term 1 type	Term 2 type	Term 3 type
equal, **not_equal** **greater**, **greater_or_equal** **less**, **less_or_equal**	$< sum >$ $< avg >$ $< stddev >$ $< max >$ $< min >$	@numeric_attribute_name	numeric constant

Table 3.13: Element BASE_COND for the SEQUENCE_AGGREGATE_FILTER operator

3.2.49 SEQUENCE_EXCEPTION

DTD

```
<!ELEMENT SEQUENCE_EXCEPTION ((%kdd_query_sequence;),
                              (%kdd_query_table;))>
<!ATTLIST SEQUENCE_EXCEPTION xml_dest %string; #IMPLIED>
```

Description

Given a set of sequences and a table, the operator extracts the transactions in the table that are exceptions to all sequences. A transaction is an exception to a sequence if it does not satisfy it (see the section 3.2.49 to see when a transaction satisfy a sequence).
The input relational table must be in a time-stamp format, as illustrated in section 2.2.3.

KDD phase

Model application.

Signature

$f_{<\texttt{SEQUENCE\_EXCEPTION}>} : \texttt{sequence} \times \texttt{table} \longrightarrow \texttt{table}$.

Required attributes

None.

Optional Attributes

- `xml_dest`.

3.2.50 SEQUENCE_FILTER

DTD

```
<!ELEMENT SEQUENCE_FILTER ((%kdd_query_sequence;), CONDITION)>
<!ATTLIST SEQUENCE_FILTER xml_dest %string; #IMPLIED>
```

Description

Given a set of sequential patterns, it returns the patterns satisfying a specified condition. The condition is expressed using an `AND/OR/NOT` combination between a set of primitive filters, concerning *support constraint*, *item constraint*, *length constraint* or *super-pattern constraint*.
The first one specifies a requirement on the support of the sequence (e.g. *"to filter sequences with a support less than 1"*).
The *item constraint* specifies what are the particular individual or groups of items that should or

should not be present in the pattern (e.g. *"to filter sequences with the item "milk" in the sequence"*).
The *length constraint* specifies requirements on the length of the patterns, where the length can be either the number of distinct items, or the number of sets, or the maximal number of items per set (e.g. *"to filter sequences with exactly 3 distinct items"*).
Finally, the *super-pattern constraint* finds patterns that contain a particular list of sets as sub-patterns (e.g. *"to filter sequences that contain the item spaghetti first and then the item milk"*).

KDD phase

Model filtering.

Signature

$f_{<\mathtt{SEQUENCE\_FILTER}>}$: `sequence` $\times$ `condition` $\longrightarrow$ `sequence`.

Required attributes

None.

Optional attributes

- `xml_dest`.

Condition specification

In reference to the figure 2.36, legal values for the XML attribute `op_type` are reported below:

- **equal**, **not_equal**, **greater**, **greater_or_equal**, **less**, **less_or_equal**: they are used for relational expressions concerning
  - the support of the sequence (`term1=@sequence_support`);
  - the number of sets of the sequence (`term1=@sequence_cardinality`);
  - the number of distinct items in the sequence (`term1=@sequence_distinct_-items_cardinality`);
  - the maximum number of items per set (`term1=@max_number_of_items_per_set`).
- **is_in**: it checks if an item belongs to the pattern;
- **is_in_all**: it checks if an item belongs to all sets of the pattern;
- **is_not_in**: it checks if an item does not belong to the pattern;
- **is_not_in_all**: it checks if an item does not belong to all sets of the pattern;

- **sub_sequence**: it checks if a given input pattern is contained in the pattern related to the condition. The input pattern is given as a string in the `term2` attribute; the sets belonging to the pattern are separated using the symbol ";"; each set is expressed using a comma separated format (e.g. *"milk, spaghetti; water; milk"* represents the sequence $\{milk, spaghetti\} \rightarrow \{water\} \rightarrow \{milk\}$).

The table 3.14 contains the types of primitive terms (XML attributes `term1, term2, term3`) to be used for each `op_type` legal value.

OpType	Term 1 type	Term 2 type	Term 3 type
equal, **not_equal**	@sequence_support	real in (0,1]	-
greater, **greater_or_equal** **less**, **less_or_equal**	@sequence_cardinality @sequence_distinct_items_cardinality @max_number_of_items_per_set	positive integer	-
is_in, **is_in_all** **is_not_in**, **is_not_in_all**	@sequence	string	-
sub_sequence	@sequence	list of sets in dot-comma separated format	-

Table 3.14: Element BASE_COND for the SEQUENCE_FILTER operator

3.2.51 SEQUENCE_LOADER

DTD

```
<!ELEMENT SEQUENCE_LOADER EMPTY>
<!ATTLIST SEQUENCE_LOADER xml_source %string; #REQUIRED>
```

Description

It loads a sequence model from the system repository.

KDD phase

Resource loading.

Signature

$f_{<\texttt{SEQUENCE\_LOADER}>} : empty \rightarrow \texttt{sequence}$.

Required attributes

- `xml_source`: the XML file source contained in the models repository.

Optional attributes

None.

3.2.52 SEQUENCE_MAXIMAL_FILTER

DTD

```
<!ELEMENT SEQUENCE_MAXIMAL_FILTER ((%kdd_query_sequence;))>
<!ATTLIST SEQUENCE_MAXIMAL_FILTER xml_dest %string; #IMPLIED>
```

Description

Given a set of sequential patterns it returns only the patterns that are maximal, i.e. that are not contained in any other pattern.

KDD phase

Model Filtering.

Signature

$f_{<\texttt{SEQUENCE\_MAXIMAL\_FILTER}>} : \texttt{sequence} \longrightarrow \texttt{sequence}$.

Required attributes

None.

Optional attributes

- `xml_dest`.

3.2.53 SEQUENCE_MINER

DTD

```
<!ELEMENT SEQUENCE_MINER ((%kdd_query_table;), ALGORITHM)>
<!ATTLIST SEQUENCE_MINER xml_dest %string; #IMPLIED>
```

Description

It extracts a set of sequential patterns by using a mining algorithm. The operator takes a data source and a sequence miner algorithm and it returns a sequence model as output.
The algorithm specification (i.e. the algorithm name and the list of expected parameters) is expressed by using the XML element `ALGORITHM` (see figure 2.35). In section 4.1.8, the list of supported sequence algorithms is reported.

KDD phase

Data mining.

Signature

$f_{<\mathtt{SEQUENCE\_MINER}>}$: `table` $\times$ `alg` $\rightarrow$ `sequence`.

Required attributes

None.

Optional attributes

- `xml_dest`.

3.2.54 SEQUENCE_RULE

DTD

```
<!ELEMENT SEQUENCE_RULE ((%kdd_query_sequence;))>
<!ATTLIST SEQUENCE_RULE xml_dest %string; #IMPLIED>
<!ATTLIST SEQUENCE_RULE min_confidence %prob_number; #REQUIRED>
<!ATTLIST SEQUENCE_RULE max_number_of_rules %integer; #IMPLIED>
```

Description

Given a set of sequential patterns, it returns all the corresponding sequential rules that satisfy a minimum confidence.
A sequence rule describes the relationship between two sequences and it consists of an antecedent and a consequent, separated by a delimiter. More in detail, a sequence rule is an implication of the form $X \rightarrow Y$, where X, Y and $Z = X(Y)$ are sequential patterns. The support and the confidence of the sequence rule are computed as reported in 3.1 and 3.2 respectively:

$$support(X \rightarrow Y) = support(Z) \tag{3.1}$$

$$confidence(X \rightarrow Y) = \frac{support(Z)}{support(X)} \tag{3.2}$$

The core procedure of the operator is shown in alg. 3, where `seq.sub_sequence(i, j)` returns the sub-sequence composed by sets from i to j of the sequence `seq`.

Algorithm 3 The core procedure of the `SEQUENCE_RULE` operator

```
Require: Sequences: sequences, real: min_confidence
  Rules rules = new Rules(min_confidence);
  for all seq in sequences do
    for i=1 to seq.lenght-1 do
      Rule rule = new Rule();
      rule.antecedent = seq.sub_sequence(0, i);
      rule.consequent = seq.sub_sequence(i+1, seq.length);
      rule.support = seq.support;
      rule.confidence = seq.support / rule.antecedent.support;
      $i = i + 1$;
      if rule.confidence $\geq$ min_confidence then
        rules.add(rule);
      end if
    end for
  end for
  return rules;
```

Notice that this operator needs the entire set of sequential patterns in order to compute all sequential rules, because the support cannot be computed for some sub-sequences (for example, applying the `SEQUENCE_RULE` operator on sequences returned by the `SEQUENCE_MAXIMAL_FILTER` operator).

KDD phase

Model meta-reasoning.

Signature

$f_{<\texttt{SEQUENCE\_RULE}>}$: `sequence` $\rightarrow$ `sequence`.

Required attributes

- `min_confidence`: a real value $c \in (0, 1]$ representing the minimum confidence of the sequential rules.

Optional attributes

- `xml_dest`.

- `max_number_of_rules`: a positive integer representing the maximum number of rules to extract. Sequential rules in the model are ordered with respect to the confidence value. If the number of sequential rules is greater than the parameter value, then rules with a low confidence are filtered out from the model. If this parameter is omitted, all computed rules are returned.

3.2.55 SEQUENCE_SATISFY

DTD

```
<!ELEMENT SEQUENCE_SATISFY ((%kdd_query_sequence;),
                            (%kdd_query_table;))>
<!ATTLIST SEQUENCE_SATISFY xml_dest %string; #IMPLIED>
```

Description

Given a set of sequences and a table, the operator extracts the transactions in the table that satisfy at least one sequence.
Informally, a transaction T is a list of pairs $T = [(t_1, I_1); \ldots ; (t_m, I_m)]$ where t_i is an identifier representing the *time-stamp* and I_i is a set of items. Pairs are ordered by *time-stamp*: $\forall i, j \in [1, m]$ if $i < j$ then $t_i < t_j$. A transaction T satisfies a sequence $S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_n$, where S_i are sets of items, if $\forall i, j \in [1, n]$ with $i < j$, $\exists h, k \in [1, m]$ such that $t_h < t_k$ and $S_i \subseteq I_h$ and $S_j \subseteq I_k$.
For example, given the transaction

$$T = [(1, \{bread, milk, mais\}); (2, \{milk, bread, wine\}); (3, \{mais\})]$$

the sequence {*milk*, *bread*} $\rightarrow$ {*mais*} is satisfied by the transaction, but the sequence {*milk*, *bread*} $\rightarrow$ {*wine*} $\rightarrow$ {*milk*} is not.
The input relational table must be in a time-stamp format, as illustrated in section 2.2.3.

KDD phase

Model application.

Signature

$f_{<\texttt{SEQUENCE\_EXCEPTION}>}$: `sequence` $\times$ `table` $\rightarrow$ `table`.

Required attributes

None.

Optional Attributes

- `xml_dest`.

3.2.56 SEQUENCE_TIMESTAMP_FILTER

DTD

```
<!ELEMENT SEQUENCE_TIMESTAMP_FILTER ((%kdd_query_sequence;),
                                     (%kdd_query_table;))>
<!ATTLIST SEQUENCE_TIMESTAMP_FILTER xml_dest %string; #IMPLIED>
<!ATTLIST SEQUENCE_TIMESTAMP_FILTER constr_type (gap|duration)
#REQUIRED> <!ATTLIST SEQUENCE_TIMESTAMP_FILTER interval_closure
             (open_closed|closed_closed|open_open|closed_open) #REQUIRED>
<!ATTLIST SEQUENCE_TIMESTAMP_FILTER lower_bound %integer; #IMPLIED>
<!ATTLIST SEQUENCE_TIMESTAMP_FILTER upper_bound %integer; #IMPLIED>
```

Description

Given a set of sequential patterns and a time-stamp table, the operator returns the patterns satisfying a specific condition.
Condition concerns a *time-stamp constraint* that is defined only in sequence databases, where each transaction in every sequence has a continuous time-stamp, i.e., the `timestamp` column of the input data source must be numeric. The constraint can be a *duration constraint* or a *gap constraint*. The first one requires that the pattern appears frequently in the sequence database in such a way that the time-stamp difference between the first and last transactions in the pattern are either longer or shorter than a given period (e.g., *"only patterns in which every event occurred within a month from the first one"*).
The *gap constraint* requires that the pattern occurs frequently in the sequence database such that the time-stamp difference between every two adjacent transactions must be either longer or shorter than a given gap (e.g., *"only patterns in which the time gap between adjacent events occurred within one day*). In both cases, the period is given as interval expressed via XML attributes.
This operator is *support-related*, i.e. the given condition is applied to check whether a sequence matches a transaction. In order to find whether a sequential pattern satisfies these constraints, one needs to examine the sequence database. In other terms, a pattern keeps being frequent only if the number of data sequences (transactions) supporting it (see the SEQUENCE_SATISFY operator in sect. 3.2.55) and satisfying the condition is greater than the minimum support. The support threshold is commonly expressed in the model and it can be specified in terms of absolute count or percentage.
The core procedure of the operator is reported in alg. 4 and alg. 5, 6 for duration and gap filter respectively, where `trans.getTimestamp(i)` returns the timestamp value of the ith set of items belonging to the transaction `trans`.

KDD phase

Model Filtering.

Signature

$f_{<\texttt{SEQUENCE\_TIMESTAMP\_FILTER}>} : \texttt{sequence} \times \texttt{table} \longrightarrow \texttt{sequence}.$

Algorithm 4 The core procedure of the `SEQUENCE_TIMESTAMP_FILTER` operator

```
Require: Sequences: sequences, Transactions: transactions, Interval: int
Ensure: Sequences
  int absolute_supp = Transactions.num_trans * Sequences.min_supp;
  for all seq in sequences do
    seq.absolute_support = 0;
    for all trans in transactions do
      if trans.satisfied(seq) and evalConstraint(trans, int) then
        seq.absolute_supp = seq.absolute_supp + 1;
      end if
    end for
    if seq.absolute_supp < absolute_supp then
      sequences.remove(seq);
    end if
  end for
  return sequences;
```

Algorithm 5 The procedure `evalConstraint(trans, int)` for the duration filter

```
Require: Transactions: trans, Interval: int
Ensure: boolean
  int value = trans.getTimestamp(trans.lenght - 1) - trans.getTimestamp(0);
  return int.contains(value);
```

Algorithm 6 The procedure `evalConstraint(trans, int)` for the gap filter

```
Require: Transactions: trans, Interval: int
Ensure: boolean
  for i=1 to trans.lenght do
    int value = trans.getTimestamp(i) - trans.getTimestamp(i-1);
    if not int.contains(value) then
      return false;
    end if
  end for
  return true;
```

Required attributes

- `constr_type`: the filter constraint to be used. Possible values are *gap* or *duration*;
- `interval_closure`: it defines a range of numeric values. Possible values are *open_closed, closed_closed, open_open,* and *closed_open.* `lower_bound` and `upper_bound` attributes define the bound values.

Optional attributes

- `xml_dest`.
- `lower_bound`: an integer representing the lower bound of the interval. The attribute is optional. If it is omitted, then - infinity is assumed.
- `upper_bound`: an integer representing the upper bound of the interval. The attribute is optional. If it is omitted, then + infinity is assumed. The system guarantees that `lower_bound` $\leq$ `upper_bound`.

3.2.57 SEQ_QUERY

DTD

```
<!ELEMENT SEQ_QUERY (%kdd_operator;,(%kdd_operator;)+)>
```

Description

The SEQ_QUERY element models sequentialization between KDDML operators. The return value of the SEQ_QUERY is assumed to be the last operator in the sequence of their arguments.

KDD phase

Control flow.

Signature

$f_{<\texttt{SEQ\_QUERY}>} : \texttt{any} \times \cdots \times \texttt{any} \rightarrow \texttt{any}.$

Required attributes

None.

Optional attributes

None.

3.2.58 TABLE_2_HIERARCHY

DTD

```
<!ELEMENT TABLE_2_HIERARCHY (%kdd_query_table;)>
<!ATTLIST TABLE_2_HIERARCHY xml_dest %string; #IMPLIED>
```

Description

The operator loads an item hierarchy from a data source. The input data source is a table with exactly two string columns:

- the first one named `child` is the attribute identifying the children of an item;
- the second one named `parent` is the father.

In the table 3.15, the three-level hierarchy *cities-states-countries* of figure 2.10 is reported.

Child	**Parent**
Long Beach	California
San Jose	California
Chicago	Illinois
California	USA
Illinois	USA

Table 3.15: An example of item hierarchy as relational table

Notice that hierarchies are not yet related to a table column as meta-data but they can be built from any table. Assigning the hierarchy to a table column as meta-data information can be achieved by the preprocessing operator `<PP_ADD_HIERARCHY>` (see section 3.2.21).

KDD phase

Resources loading.

Signature

$f_{\texttt{<TABLE\_2\_HIERARCHY>}} : \texttt{table} \longrightarrow \texttt{hierarchy}$.

Required attributes

None.

Optional attributes

- `xml_dest`.

3.2.59 TABLE_2_PP_TABLE

DTD

```
<!ELEMENT TABLE_2_PP_TABLE (%kdd_query_table;)>
<!ATTLIST TABLE_2_PP_TABLE xml_dest %string; #IMPLIED>
```

Description

The operator starts the preprocessing phase by mapping the input relation table into a preprocessing table. An empty preprocessing section is added to the output table, while the data schema and the physical instances do not change.

KDD phase

Preprocessing.

Signature

$f_{\texttt{TABLE\_2\_PP\_TABLE}} : \texttt{table} \rightarrow \texttt{PPtable}$

Required attributes

None.

Optional Attributes

- `xml_dest`.

3.2.60 TABLE_LOADER

DTD

```
<!ELEMENT TABLE_LOADER EMPTY>
<!ATTLIST TABLE_LOADER xml_source %string; #REQUIRED>
```

Description

The operator loads a relational table from the system repository.

KDD phase

Resource loading.

Signature

$f_{<\texttt{TABLE\_LOADER}>} : empty \rightarrow \texttt{table}$.

Required attributes

- `xml_source`: the XML file source contained in the data repository.

Optional attributes

None.

3.2.61 TREE_CLASSIFY

DTD

```
<!ELEMENT TREE_CLASSIFY ((%kdd_query_trees;),
                         (%kdd_query_table;))>
<!ATTLIST TREE_CLASSIFY xml_dest %string; #IMPLIED>
```

Description

Given a classification tree and a table, the operator yields a new table with an additional column at the end of the schema consisting of the class predicted by the decision tree. The name of the new column is the name of the classification attribute of the tree followed by the extension `_predicted`. For example, if the target attribute is *"play_tennis"*, then the output column is named as *"play_tennis_predicted"*. The type of the predicted attribute become nominal.
The procedure used to determine the class predicted is the one adopted in the C4.5 algorithm. The mapping between attributes used in the classification tree and attributes in the dataset is by name. Therefore, the input relational table must be compatible with the classification tree. In particular, a table is compatible with a model if for each active mining field belonging to the model, there is an attribute in the table with the same name and type.

KDD phase

Model application.

Signature

$f_{\texttt{<TREE\_CLASSIFY>}} : \texttt{tree} \times \texttt{table} \rightarrow \texttt{table}$.

Required attributes

None.

Optional Attributes

- `xml_dest`.

3.2.62 TREE_LOADER

DTD

```
<!ELEMENT TREE_LOADER EMPTY>
<!ATTLIST TREE_LOADER xml_source %string; #REQUIRED>
```

Description

It loads a classification tree from the system repository.

KDD phase

Resource loading.

Signature

$f_{\texttt{<TREE\_LOADER>}} : empty \rightarrow$ `tree`.

Required attributes

- `xml_source`: the XML file source contained in the models repository.

Optional attributes

None.

3.2.63 TREE_META_CLASSIFIER

DTD

```
<!ELEMENT TREE_META_CLASSIFIER ((%kdd_query_trees;),
                                 (%kdd_query_trees;)+)>
<!ATTLIST TREE_META_CLASSIFIER xml_dest %string; #IMPLIED>
<!ATTLIST TREE_META_CLASSIFIER combination_type (and|or|committee) #REQUIRED>
<!ATTLIST TREE_META_CLASSIFIER positive_class %string; #IMPLIED>
```

Description

Models extracted by data mining algorithms very often need to be further processed, e.g., combined with other models. The operator takes a set of classification trees and returns a voting meta-classifier among the input trees.
The output meta-classifier is built according to a combination method, as reported below:

- *committee*: constructs a *voting classifier* on input trees. More precisely, given n distinct classifiers $C_1, \ldots, C_n$, a *voting classifier* assigns to a tuple the class mostly assigned by $C_1, \ldots, C_n$.
- *and* (*or*): builds a boolean *AND tree (OR tree)* on a set of decision trees[5]. More precisely, given n distinct decision trees $D_1, \ldots, D_n$, the *AND tree* (*OR tree*) assigns to a tuple the positive class (negative class) if all the trees $D_1, \ldots, D_n$ classify the instance with the positive class (negative class); otherwise, the negative class (positive class) is returned.

The operator performs a run-time checking that the three classifiers have been extracted from the same meta-data. In particular, input trees must share the same mining schema and data dictionary.

[5]In a decision tree, the target attribute is binary with a positive class (e.g. *true, yes*) and a negative class (e.g. *false, no*) as values.

KDD phase

Model meta-reasoning.

Signature

$f_{\texttt{<TREE\_META\_CLASSIFIER>}} : \texttt{tree} \times \cdots \times \texttt{tree} \rightarrow \texttt{tree}$.

Required attributes

- `combination_type`: the combination procedure to be used for trees. Possible values are *committee* or *and, or* for boolean classifiers. In the last case, the XML attribute `positive_class` identifies the positive class to be used.

Optional Attributes

- `xml_dest`.
- `positive_class`: is the positive class used when the target attribute is binary. The XML attribute is required if a boolean combination strategy is used.

3.2.64 TREE_MINER

DTD

```
<!ELEMENT TREE_MINER ((%kdd_query_table;), ALGORITHM)>
<!ATTLIST TREE_MINER xml_dest %string; #IMPLIED>
<!ATTLIST TREE_MINER target_attribute %string; #REQUIRED>
```

Description

It extracts a classification tree by using a mining algorithm. The operator takes a table representing the training set, the name of the target attribute and a tree miner algorithm and returns a tree model as output.
The algorithm specification (i.e. the algorithm name and the list of expected parameters) is expressed by using the XML element `ALGORITHM` (see figure 2.35). In section 4.1.6, the list of supported classification algorithms is reported. The classification attribute is given as XML attribute.
The data schema of the input data source depends on the algorithm specification. In other words, some attributes can be ignored during the mining if their types are not supported by the algorithm. For example, ID3 cannot work on continuous attributes. However, preprocessing operators can be used to adapt the input table to specific data mining algorithms.

KDD phase

Data mining.

Signature

$f_{\texttt{<TREE\_MINER>}}$: $\texttt{table} \times \texttt{alg} \rightarrow \texttt{tree}$.

Required attributes

- `target_attribute`: the attribute to be used for classification. Target attribute must be nominal.

Optional attributes

- `xml_dest`.

CHAPTER 4

KDDML algorithms

This chapter contains all the KDDML supported algorithms for the preprocessing and mining phases. With reference to figure 2.35, for each algorithm it reports:

1. *the algorithm name to be used in the* `algorithm_name` *XML attribute with a description of the algorithm;*
2. *the format of input data source required by the algorithm*
3. *the list of supported parameters with an explicative usage description;*
4. *a table with a row for each parameter containing the parameter type and the parameter usage. More in detail, the table contains:*
   - *the* parameter name *to be used in the attribute* `name` *of the XML element* `PARAM`*;*
   - *the* parameter type *containing the expected type to be used in the XML attribute* `value` *of the element* `PARAM`.
   - *the usage of the parameter: it can be* optional, required *or* fixed.
   - *the default value to be used when the parameter is omitted (for optional parameters only) in the XML definition.*

Language algorithms will be presented according to a lexicographic ordering.

4.1 Classification of algorithms

KDDML algorithms can be classified according to the type of knowledge they extract.

4.1.1 Discretization algorithms

Discretization techniques can be used to reduce the number of values for a given continuous attribute, by dividing the range of the attribute into intervals. Discretization algorithms are used by the `PP_NUMERIC_DISCRETIZATION` operator that takes a preprocessing table as first argument. Currently, KDDML supports the discretization algorithms shown on table 4.1.

Algorithm Name	Description	Sect.
equal_frequency_discretization	It divides the range of an attribute into k intervals of equal cardinality	4.2.5
natural_binning_discretization	It divides the range of an attribute into k intervals of equal width	4.2.8

Table 4.1: Discretization algorithms

4.1.2 Normalization algorithms

Normalization is the process of scaling data values of a numeric attribute into a range such as [-1, 1] or [0,1]. Normalization algorithms reported in table 4.2 are used by the `PP_NORMALIZATION` operator (see sect. 3.2.31) that takes a preprocessing table as first argument.

Algorithm Name	Description	Sect.
min_max_normalization	It implements a linear transformation based on the min-max method	4.2.7
z_score_normalization	The values of an attribute are normalized on the basis of his mean and standard deviation	4.2.16

Table 4.2: Normalization algorithms

4.1.3 Rewriting algorithms

Rewrite algorithms match the input attribute value against a pattern, and if a match is found, they rewrite the value with a new format by using the rules defined in a transformation rule. Each rewriting algorithm is defined by three features:

1. a regular expression (*regex*) is used to perform the pattern matching. Basically, a *regex* is a string that describes or matches against a set of strings, according to certain syntax rules.

As an instance, $(a|b)*$ denotes the set of all strings consisting of a and b, including the empty string. KDDML uses the basic `java.util.regex` Java package [13] for regular expressions.

2. the transformation rule to be applied on attribute values when the pattern matching is satisfied;

3. the policy to be applied when the pattern matching fails for a given instance value.

In general, the last two items distinguish the rewrite algorithms. Rewriting methods are used by the `PP_REWRITING` operator (see section 3.2.36) that takes a preprocessing table as first argument. The table 4.3 lists the KDDML rewriting algorithms.

Algorithm Name	**Description**	**Sect.**
rule_m	As rule-S, but it works only on preprocessing section marking meta-data that match a regular expression	4.2.10
rule_s	It replaces the entire attribute value that matches a regular expression	4.2.11
rule_t	It replaces every sub-sequence of the attribute value that matches a regular expression	4.2.12

Table 4.3: Rewriting algorithms

4.1.4 Sampling algorithms

Sampling can be used as a data reduction technique, since it allows a large data set to be represented by a much smaller random sample (or subset) of the data. The sampling algorithms reported in table 4.4 are used by the `PP_SAMPLING` operator (see sect. 3.2.37) that takes a preprocessing table as first argument.

Algorithm Name	**Description**	**Sect.**
by_cluster_sampling	It selects a random set of "stratum" of a nominal attribute belonging to the input table	4.2.1
simple_sampling	It returns a random subset of fixed cardinality from the input preprocessing table	4.2.13
stratified_sampling	It performs a simple sampling at each "stratum" of a nominal attribute belonging to the input table	4.2.14

Table 4.4: Data sampling algorithms

4.1.5 RdA miner algorithms

Mining models are extracted from a data source by using a data mining algorithm. Each miner operator expects a sub-element with input data and a second sub-element with the algorithm name and parameters (name and value). In table 4.7 all the supported association rules algorithms are reported; they are used by the RDA_MINER operator (see sect. 3.2.44).

Algorithm Name	Description	Sect.
DCI	It finds frequent itemsets and can be considered an enhancement of the well known Apriori. It is implemented in C.	4.2.3

Table 4.5: RdA miner algorithms

4.1.6 Tree miner algorithms

In table 4.6 all the supported classification algorithms are reported; they are used by the TREE_MINER operator (see sect. 3.2.64).

Algorithm Name	Description	Sect.
CAGE	It is a tool for parallel genetic programming that realizes a parallel implementation on distribuited-memory computers. It is implemented on linux platform using the MPI libraries	4.2.2
YaDT	C++ implementation of the C4.5 algorithm	4.2.15

Table 4.6: Tree miner algorithms

4.1.7 Cluster miner algorithms

KDDML supports two types of clustering: the *distribution-based* clustering and the *centroid-based* clustering; as reported in table 4.7 all KDDML clustering algorithms used by the CLUSTER_MINER operator (see sect. 3.2.6) belong to one or to other class.

Algorithm Name	Description	Sect.	Clustering Type
EM	Implementation of the EM algorithm. It uses the WEKA system library.	4.2.4	Distribution based
KMeans	Implementation of the KMeans algorithm It uses the WEKA system library.	4.2.6	Centroid based

Table 4.7: Cluster miner algorithms

4.1.8 Sequence miner algorithms

In table 4.8 all the supported sequence algorithms are reported; they are used by the SEQUENCE_MINER operator (see sect. 3.2.53).

Algorithm Name	Description	Sect.
prefix_span	Java proprietary implementation of the PrefixSpan algorithm	4.2.9

Table 4.8: Sequence miner algorithms

4.2 Algorithms specification

4.2.1 By cluster sampling

Algorithm Name

`by_cluster_sampling`

Description

Suppose that the tuples of the input database are grouped into M mutually disjoint *clusters* (according to the values of a nominal attribute), then a simple random sampling of m clusters can be obtained, where $m \leq M$. The clusters are selected according to a *SRSWR* policy or to a *SRSWOR* policy (see sect. 4.2.13).
The number of output clusters can be given either in absolute form (using the parameter `number-_of_categories`) or as percentage (using the parameter `percentage`) with respect to the number of clusters, M. It is important to notice that the schema of the data does not change when applying the algorithm, and that the categories are not removed from the by-cluster sampling attribute.
Currently, the algorithm uses a proprietary Java implementation.

Input data format

The algorithm takes as input a preprocessing table containing at least a nominal field, representing the clustering attribute.

Parameters description

- `attribute_name`: it is the name of the by-cluster sampling attribute. By-cluster sampling can be applied only to nominal attributes.
- `percentage`: it is the percentage of output clusters (i.e., categories) with respect to the total number of clusters, M. The parameter `percentage` and the parameter `number-_of_clusters` are mutually exclusive; i.e. if this parameter is specified by the user, the other one must be omitted;

- `number_of_categories`: it is the absolute number of output clusters (i.e., categories), m. If $m > M$ and the *SRSWOR* technique is selected, then all instances belonging to the input table will be returned by the algorithm: no tuple replication is performed. Otherwise, if $m > M$ and the *SRSWR* method is selected, then the output table will contain more instances that the input table and some instances (belonging to the same cluster) will be replicated. If this parameter is specified by the user, the parameter `percentage` must be omitted.
- `with_replacement`: it selects the replacement strategy to be applied to the M clusters. The parameter can be *true* or *false*. In the first case, the *SRSWR* method is selected and the instances belonging to the same cluster can be replicated. Otherwise, the algorithm uses the *SRSWOR* method for simple sampling.

Parameters specification

Table 4.9 contains the parameters specification for the by-cluster sampling technique to be used in the `PARAM` XML element of figure 2.35.

Parameter Name	**Parameter Value**	**Usage**	**Default Value**
`attribute_name`	string	required	-
`percentage`	real in $(0, 1]$	optional	-
`number_of_categories`	positive integer	optional	-
`with_replacement`	$< true >$ $< false >$	optional	$< false >$

Table 4.9: `by_cluster_sampling` parameters.

4.2.2 CAGE (CellulAr GEneting programming tool)

Algorithm Name

`CAGE`

Description

CAGE (*CellulAr GEneting programming tool*) [14] is a tool for parallel genetic programming applications, that realizes a fine-grained parallel implementation of genetic programming on distributed-memory parallel computers. Experimental results on some classical test problems shows that the cellular model outperforms both the sequential canonical implementation of GP and the parallel island model. Furthermore parallel cellular GP has a nearly linear speed-up and a good scale-up behavior. CAGE implements the cellular GP model using a one-dimensional domain decomposition (in the x direction) of the grid and an explicit message passing to exchange information among the domains. All the communications are performed using the MPI (Message Passing Interface)

portable message passing system so that CAGE can be executed across different hardware platforms. Since the processes are connected according to a ring architecture and each process has a limited buffer for storing boundary data, asynchronous communication are used in order to avoid processors to idle. CAGE uses the standard tool for genetic programming sgpc1.1, a simple GP in the C language to apply the GP algorithm to each grid point.
Important note: the CAGE algorithm works on linux platform only and needed of the MPI libraries installed on operative system.

Input data format

In currently implementation, the CAGE algorithm supports nominal attributes only: no string or numeric attributes are allowed in the input table.

Parameters description

- `num_processors`: a positive integer representing the number of processors used. Default: 1.
- `num_iterations`: a positive integer representing the maximum number of iterations needed. Default: 100.
- `parameter_file`: the path of a configuration file used by the algorithm.
- `classification_type`: it specifies if the algorithm uses the boosting classification or not.
- `perc_data`: a double in (0,1] containing the percentage of input instances used by CAGE as training set. This parameter is used only with boosting classification technique.

Parameters specification

Table 4.10 contains the parameters specification for the CAGE algorithm to be used in the `PARAM` XML element of figure 2.35.

Parameter Name	**Parameter Value**	**Usage**	**Default Value**
`num_processors`	positive integer	optional	1
`num_iterations`	positive integer	optional	100
`parameter_file`	string	optional	-
`classification_type`	$< boost >$ $< no\_boost >$	optional	$< no\_boost >$
`perc_data`	real in $(0, 1]$	optional	-

Table 4.10: CAGE parameters

4.2.3 DCI (Direct Count & Intersect)

Algorithm Name

`DCI`

Description

DCI (Direct Count & Intersect) [15] is an algorithm for finding frequent sets of items from a transactional database. It can be considered an enhancement of the well known Apriori, because they both share the same level-wise approach and use the same candidate generation technique. Nevertheless, DCI improves considerably the performance of Apriori by introducing a new pruning technique and the automatic switching to a vertical database representation, during execution. DCI adopts a direct counting based approach for the first iterations and an intersection based approach for the vertical dataset iterations. Moreover DCI adopts a new counting inference strategy based on the notion of key-patterns.
DCI is used by KDDML to generate frequent itemsets and it has been extended through a proprietary Java implementation, in order to extract association rules also. Both the number of requested output itemsets and the number of output association rules are (optional) input parameters of the algorithm. In order to use it, the user must specify the minimum support and the minimum confidence of the rules.

Input data format

According to section 2.2.3, the input data source can be in a *relational format*, i.e. a row for each single transaction and an attribute for every item. This format allows for deriving inter-attribute association rules such as *"carType=racing AND homeInsurance=false* $\rightarrow$ *married=false"*. In a relational table, numeric attributes are ignored by the DCI algorithm.
Also the *transactional format* is recognized. The last one is used to deriving intra-attribute association rules such as *"spaghetti AND tomato* $\rightarrow$ *parmesan"*. In a transaction format, the DCI algorithm uses only the attributes `item` and `transaction` during the extraction process. Other attributes are allowed in the table, but they are ignored by the mining algorithm.

Parameters description

- `min_support`: the minimum support of a rule or itemset;
- `min_confidence`: the minimum confidence of a rule;
- `max_number_of_itemsets`: the maximum number of itemsets to extract;
- `max_number_of_rules`: the maximum number of rules to extract.

Parameters specification

Table 4.11 contains the parameters specification for the DCI algorithm to be used in the `PARAM` XML element of figure 2.35.

Parameter Name	Parameter Value	Usage	Default Value
`min_support`	real value in (0,1]	required	-
`min_confidence`	real value in (0,1]	required	-
`max_number_of_itemsets`	positive integer	optional	-
`max_number_of_rules`	positive integer	optional	-

Table 4.11: DCI parameters

4.2.4 EM (Expectation Maximization)

Algorithm Name

`EM`

Description

The EM (Expectation Maximization) [9] algorithm is an effective, popular technique for estimating mixture model parameters (cluster parameters and their mixture weights). The EM algorithm iteratively refines initial mixture model parameter estimates to better fit the data and it terminates at a locally optimal solution. EM is a distribution-based technique.
In the current version, the EM algorithm is implemented using the WEKA [16] library[1].

Input data format

The EM algorithm supports nominal or continuous attributes only: no string attributes are allowed in the input table.

Parameters description

- `number_of_clusters`: it specifies the number of clusters to generate. If it is omitted, EM will use cross validation to select the optimal number of clusters.
- `max_iterations`: terminate after this many iterations if EM has not converged.
- `min_std_dev`: set the minimum allowable standard deviation for normal density calculation.

Parameters specification

Table 4.12 contains the parameters specification for the EM algorithm to be used in the `PARAM` XML element of figure 2.35.

[1]Notice that the algorithm is based on a main-memory Java implementation.

Parameter Name	Parameter Value	Usage	Default Value
`number_of_clusters`	positive integer	optional	-
`max_iterations`	positive integer	optional	100
`min_std_dev`	real in $(0, 1]$	optional	1

Table 4.12: EM parameters

4.2.5 Equal frequency discretization

Algorithm Name

`equal_frequency_discretization`

Description

The *Equal Frequency Discretization* method (EFD) divides the range of a numeric attribute A into k intervals containing the same number of samples.
Suppose there are n training instances for which the values of A are known (missing values will be ignored). More in details, the algorithm sorts the observed values and then divides the sorted values into k intervals so that each interval contains (approximately[2]) the same number of training instances. Thus each interval contains n/k (possibly duplicated) adjacent values. The number of output intervals k and the number of required samples for each interval are mutually exclusive parameters.
When the intervals have been computed, the algorithm replaces each training instance value of A with an *interval label*. As previously reported, the system allows a numeric or nominal labeling (see section 4.2.8).
At present, the algorithm uses a proprietary Java implementation.

Input data format

The algorithm takes as input a preprocessing table containing at least a numeric field, representing the discretization attribute.

Parameters description

- `number_of_intervals`: it is the number of output intervals k. This parameter and the parameter `cardinality_of_intervals` are mutually exclusive.
- `cardinality_of_intervals`: it is the number of cases assigned to each interval. This parameter and the parameter `number_of_intervals` are mutually exclusive.
- `labeling`: the labeling strategy to be used. Possible values are:

[2]The number of instances can vary for the last computed interval.

  - *mean*: it is used to compute the mean of the values belonging to the interval;
  - *median*: it is used to compute the median of the values belonging to the interval;
  - *inf (sup)*: it is used to compute the inferior (superior) bound of the interval;
  - *enumeration*: it is used for a nominal interval labeling. In this case, the list of nominal labels can be (optionally) provided using the parameter `enumerated_label_list`.

- `enumerated_label_list`: the list of nominal labels to use when the `labeling` parameter is *enumeration* and the parameter `number_of_intervals` is specified. In this case, the system guarantees that the number of required intervals is equal to the number of nominal labels provided by the user. If the parameter is omitted, each attribute value is labeled with a string representing the interval (e.g., *(45, 50]*). The list of labels is given in a comma separated format (e.g., *young*, *adult*, *elder*).

Parameters specification

Table 4.13 contains the parameters specification for the equal frequency normalization method to be used in the `PARAM` XML element of figure 2.35.

Parameter Name	**Parameter Value**	**Usage**	**Default Value**
`number_of_intervals`	positive integer	required if `cardinality_of _intervals` is omitted	-
`cardinality_of_intervals`	positive integer	required if `number_of _intervals` is omitted	-
`labeling`	$< mean >$ $< median >$ $< inf >$ $< sup >$ $< enumeration >$	required	- -
`enumerated_label_list`	list of strings comma separated	optional	-

Table 4.13: `equal_frequency_discretization` parameters.

4.2.6 KMeans

Algorithm Name

`KMeans`

Description

The KMeans procedure [8] follows a simple and easy way to classify a given data set through a certain number of clusters (assume k clusters) fixed a priori. The main idea is to define k centroids, one for each cluster. These centroids should be placed in a cunning way because a different location causes different result. So, the better choice is to place them as much as possible far away from each other. The next step is to take each point belonging to a given data set and associate it to the nearest centroid. When no point is pending, the first step is completed and an early groupage is done. At this point we need to re-calculate k new centroids as centers of the clusters resulting from the previous step. After we have these k new centroids, a new binding has to be done between the same data set points and the nearest new centroid. A loop has been generated. As a result of this loop we may notice that the k centroids change their location step by step until no more changes are done. In other words centroids do not move any more.
KMeans is a centroid-based technique and it uses the *euclidean distance* to compare two cluster objects, the *absolute difference* to compare two continuous fields and the *delta function*[3] to compare two nominal fields. In the current version, the KMeans algorithm is implemented by using the WEKA [16] library[4].
The KMeans algorithm supports nominal or continuous attributes only: no string attributes are allowed in the input table.

Input data format

The KMeans algorithm supports nominal or continuous attributes only: no string attributes are allowed in the input table.

Parameters description

- `number_of_clusters`: specify the number of clusters, k, to generate.

Parameters specification

Table 4.14 contains the parameters specification for the KMeans algorithm to be used in the `PARAM` XML element of figure 2.35.

Parameter Name	**Parameter Value**	**Usage**	**Default Value**
`number_of_clusters`	positive integer	optional	2

Table 4.14: KMeans parameters

[3]$c(x, y) = 0$ if $x = y$, 1 otherwise, where x and y are two fields values.
[4]Notice that the algorithm is based on a main-memory Java implementation.

4.2.7 Min-max normalization

Algorithm Name

`min_max_normalization`

Description

It performs a linear transformation on the original data. Suppose that min_A and max_A are the minimum and maximum values of an attribute A. Min-max normalization maps a value v of A to v' in the range $[new\_min_A, new\_max_A]$ by computing

$$v' = \frac{v - min_A}{max_A - min_A}(new\_max_A - new\_min_A) + new\_min_A \tag{4.1}$$

where new_min_A and new_max_A are parameters of the algorithm. The system guarantees that $new\_min_A \leqslant new\_max_A$.

Input data format

The algorithm takes as input a preprocessing table containing at least a numeric field, representing the discretization attribute.

Parameters description

- `inf`: it is the new minimum value, new_min_A, for the attribute A.
- `sup`: it is the new maximum value, new_max_A, for the attribute A.

Parameters specification

Table 4.15 contains the parameters specification for the min-max normalization method to be used in the `PARAM` XML element of figure 2.35.

Parameter Name	Parameter Value	Usage	Default Value
`inf`	real	optional	-1
`sup`	real	optional	1

Table 4.15: `min_max_normalization` parameters.

4.2.8 Natural binning discretization

Algorithm Name

`natural_binning_discretization`

Description

The *natural binning discretization* method divides the range of a numeric attribute A into k intervals of equal width. The method is also known as *Equal Width Discretization (EWD)*.
Suppose that there are n training instances for which the values of A are known (missing values will be ignored) and suppose that the minimum and maximum value are v_{min} and v_{max} respectively. The algorithm sorts the observed values and then divides the number of values between v_{min} and v_{max} into k intervals of (approximately[5]) equal width. Thus the intervals have width

$$w = \frac{v_{max} - v_{min}}{k} \tag{4.2}$$

and the cut points are at $v_{min} + w, v_{min} + 2w, \ldots, v_{min} + (k-1)w$. The number of output intervals k and the width of the interval w are mutually exclusive parameters.
When the intervals have been computed, the algorithm replaces each training instance value of A with an *interval label*. *Numeric* or *nominal* labeling are allowed.
A *Numeric interval label* includes the mean, the median, the minimum or maximum calculated on the values belonging to the interval.
A *Nominal interval label* includes a list of strings, each containing the labels used to replace each training instance value belonging to the interval. The system guarantees that the number of nominal labels is equal to the number of output intervals k. The mapping between intervals computed by the algorithm and nominal labels starts from the interval containing the lowest values[6]. As an instance, suppose that the algorithm computes the intervals $I_1 = [6, 35), I_2 = [35, 65)$ and $I_3 = [65, 95)$. Moreover suppose that the nominal labels provided are *"young"*, *"adult"* and *"elder"* in that order. For each training input instance, a value v of the discretization attribute is replaced with *"young"*, *"adult"* and *"elder"* if $v \in I_1$, $v \in I_2$ and $v \in I_3$ respectively. By using the nominal interval labeling, the type of the discretization attribute become enumerated.
At present, the algorithm is implemented using (in part) the WEKA system library.

Input data format

The algorithm takes as input a preprocessing table containing at least a numeric field, representing the discretization attribute.

Parameters description

- `number_of_intervals`: it is the number of output intervals k. This parameter and the parameter `width_of_intervals` are mutually exclusive.

- `width_of_intervals`: it is the size of each interval w. This parameter and the parameter `number_of_intervals` are mutually exclusive.

- `labeling`: the labeling strategy to be used. Possible values are:

[5]The width of intervals can variate for the last interval computed.
[6]Remember that values are ordered by the algorithm.

  - *mean*: it is used to compute the mean of the values belonging to the interval;
  - *median*: it is used to compute the median of the values belonging to the interval;
  - *inf (sup)*: it is used to compute the inferior (superior) bound of the interval;
  - *enumeration*: it is used for a nominal interval labeling. In this case, the list of nominal labels can be (optionally) provided by using the parameter `enumerated_label_list`.

- `enumerated_label_list`: the list of nominal labels to be used when the `labeling` parameter is *enumeration* and the parameter `number_of_intervals` is specified. In this case, The system guarantees that the number of required intervals is equal to the number of nominal labels provided by the user. If the parameter is omitted, each attribute value is labeled with a string representing the interval (e.g., *(45, 50]*). The list of labels is given in a comma separated format (e.g., *young, adult, elder*).

Parameters specification

Table 4.16 contains the parameters specification for the natural binning discretization method to be used in the `PARAM` XML element of figure 2.35.

Parameter Name	**Parameter Value**	**Usage**	**Default Value**
`number_of_intervals`	positive integer	required if `width_of_intervals` is omitted	-
`width_of_intervals`	positive integer	required if `number_of_intervals` is omitted	-
`labeling`	$< mean >$ $< median >$ $< inf >$ $< sup >$ $< enumeration >$	required	- -
`enumerated_label_list`	list of strings comma separated	optional	-

Table 4.16: `natural_binning_discretization` parameters.

4.2.9 Prefix Span

Algorithm Name

`prefix_span`

Description

PrefixSpan (*Prefix-projected Sequential pattern mining*) [17] mines the complete set of patterns but greatly reduces the efforts of candidate subsequence generation. Moreover, prefix-projection substantially reduces the size of projected databases and leads to efficient processing.
KDDML uses a proprietary main-memory Java implementation of the Prefix Span algorithm. In order to use the algorithm, the user must specify the minimum support of the sequences; otherwise, the number of output sequences is an optional parameter.

Input data format

Prefix Span takes as input a time-stamp table, as described in section 2.2.3. In this format, the Prefix Span algorithm uses only the attributes `item`, `transaction` and `timestamp` during the extraction process. Other attributes are allowed in the table, but they are ignored by the mining algorithm.

Parameters description

- `min_support`: it is the minimum support of a sequence;
- `max_number_of_sequences`: it is the maximum number of sequences to extract.

Parameters specification

Table 4.17 contains the parameters specification for the Prefix Span algorithm to be used in the `PARAM` XML element of figure 2.35.

Parameter Name	Parameter Value	Usage	Default Value
`min_support`	real value in (0,1]	required	-
`max_number_of_sequences`	positive integer	optional	-

Table 4.17: prefix_span parameters

4.2.10 Rule M

Algorithm Name

`rule_m_rewriting`

Description

The rule-M method works only on the preprocessing section of the input table and it does not execute any transformation to the data section.
In particular, the algorithm marks (i.e., a string is added to preprocessing information) the instance

values of the rewriting attribute if the pattern matches the entire attribute value. Notice that the matching procedure is performed on the entire input string value, i.e., the algorithm attempts to match the entire input sequence against the pattern. For example, the regular expression *"dog"* matches a sub-sequence of the pattern *"dogcatdog"*, but not the entire string. In this example, the matching procedure will return failure.

Input data format

The rule-m algorithm takes a preprocessing table as input.

Parameters description

- `regular_expression`: the regular expression. If the attribute is equal to *"?"*, then the algorithm applies the substitution to all missing instances.
- `mark`: the string value to be added to the preprocessing section of the rewriting attribute.

Parameters specification

Table 4.18 contains the parameters specification for the rule-M rewriting method to be used in the `PARAM` XML element of figure 2.35.

Parameter Name	**Parameter Value**	**Usage**	**Default Value**
`regular_expression`	string	required	-
`mark`	string	required	-

Table 4.18: `rule_m_rewriting` parameters.

4.2.11 Rule S

Algorithm Name

`rule_s_rewriting`

Description

The rule-S method replaces the entire input string value that matches the pattern with a given replacement string that may contain references to captured sub-sequences (see later). Notice that, the matching procedure is performed on the entire input string value, i.e., the algorithm tries to match the entire input sequence against the pattern. For example, the regular expression *"dog"* match a sub-sequence of the pattern *"dogcatdog"*, but not the entire string. In this example, the matching procedure returns failure.

The user must specify two replacement strings by using two different algorithm parameters. The first one, called `substitution_then`, is applied to the entire string value if the pattern matches that string; the second one, called `substitution_else`, is applied to the string value only if the pattern does not match. The `substitution_else` parameter is optional. For example, given the regular expression *"a*b"*, the "then" replacement string *"ok"* and the "else" replacement string *"ko"*, an invocation of this method on the instance value *"aaaaab"* would yield the string *"ok"*. Otherwise, an invocation on the instance value *"aaaaabb"* would yield the string *"ko"*.
If no substitution is performed, then the instance value is marked (i.e. a string is added to preprocessing information) with an exception for the rewriting attribute. Anyhow, the marking policy is optional.

Input data format

The rule-s algorithm takes a preprocessing table as input.

Parameters description

- `regular_expression`: the regular expression. The user can refer to missing instances using the symbol "?" in the XML attribute `value` of the regular expression parameter. This is useful when the user wants to rewrite all missing instances by a known value. Regular expressions can contain references to captured sub-sequences. Capturing groups are numbered by counting their opening parentheses from left to right. In the expression ((A)(B(C))), for example, there are four such groups:

  1. ((A)(B(C))),
  2. (A),
  3. (B(C)),
  4. (C).

  Group zero always stands for the entire expression. Capturing groups are so named because, during a match, each subsequence of the input sequence that matches such a group is saved. The captured sub-sequence may be used later in the expression, via a back reference, and it may also be retrieved from the matcher once the match operation is complete. The notation `$n` returns the nth sub-sequence captured by the given group during the previous match operation. For example, in the example above, the expression `$1` refers to the sub-sequence A.

- `substitution_then`: the substitution string to be used if the pattern matching. The replacement string may contain references to sub-sequences captured during the pattern matching. In particular, the dollar sign `$` in front of the number of sub-sequence n, can be used to specify the nth captured sub-sequence.

- `substitution_else`: the substitution string to be used if the pattern does not match the entire input string value. Also for `substitution_then`, the notation `$n` can be used to return the nth captured sub-sequence.

- `mark_metadata_with_exception`: can be *true* or *false*. In the first case, the algorithm marks the preprocessing information with an exception if no substitution is performed. Otherwise, no mark is applied.

Parameters specification

Table 4.19 contains the parameters specification for the rule-S rewriting method to be used in the `PARAM` XML element of figure 2.35.

Parameter Name	**Parameter Value**	**Usage**	**Default Value**
`regular_expression`	string	required	-
`substitution_then`	string	required	-
`substitution_else`	string	optional	-
`mark_metadata_with_exception`	$< true >$ $< false >$	optional	$< true >$
`mark_exception`	string	fixed	$< no\_rule\_s\_matching >$

Table 4.19: `rule_s_rewriting` parameters.

4.2.12 Rule T

Algorithm Name

`rule_t_rewriting`

Description

The rule-T method replaces every sub-sequence of the input string value that matches the pattern with a given replacement string. It scans the input sequence looking for a match of the pattern. Then, each match is replaced in the result by the replacement string that may contain references to captured subsequences (see section 4.2.11). For example, given the regular expression *"dog"*, the input instance value *"dogcatdog"*, and the replacement string *"cat"*, an invocation of this method on a matcher for that expression would yield the string *"catcatcat"*.
If no substitution is performed, then the instance value is marked (i.e. a string is added to preprocessing information) with an exception for the rewriting attribute. However, the marking policy is optional.

Input data format

The rule-t algorithm takes a preprocessing table as input.

Parameters description

- `regular_expression`: the regular expression.
- `substitution_then`: the substitution string to be used if the pattern matches. The replacement string may contain references to subsequences captured during pattern matching. In particular, the dollar sign `$` before the number of sub-sequence n, can be used to specify the nth captured sub-sequence.
- `mark_metadata_with_exception`: can be *true* or *false*. In the first case, the algorithm marks the preprocessing information with an exception if no substitution is performed. Otherwise, no mark is applied.

Parameters specification

Table 4.20 contains the parameters specification for the rule-T rewriting method to be used in the `PARAM` XML element of figure 2.35.

Parameter Name	Parameter Value	Usage	Default Value
`regular_expression`	string	required	-
`substitution_then`	string	required	-
`mark_metadata_with_exception`	$< true >$ $< false >$	optional	$< true >$
`mark_exception`	string	fixed	$< no\_rule\_t\_matching >$

Table 4.20: `rule_t_rewriting` parameters.

4.2.13 Simple sampling

Algorithm Name

`simple_sampling`

Description

It returns a random subset of fixed cardinality from the input preprocessing table. According to the parameter `with_replacement`, simple sampling can use two distinct techniques: the *with replacement* method and the *without replacement* method.
More precisely, suppose that a data set contains N tuples. The *Simple Random Sampling WithOut Replacement* method (*SRSWOR*) selects n tuples from the input database, where the probability of drawing any tuple is $1/N$, that is, all tuples are equally likely.
The *Simple Random Sample With Replacement* method (*SRSWR*) is similar to SRSWOR, except that each time a tuple is drawn from the dataset, it is recorded and then replaced. That is, after a

tuple is drawn, it is placed back id the dataset, so that it may be drawn again.
The number of output instances, n, can be given either in absolute form (using the parameter `number_of_instances`) or as percentage (using the parameter `percentage`) with respect to the total number of input instances.
By now, the algorithm uses a proprietary Java implementation.

Input data format

The simple sampling algorithm takes a preprocessing table as input.

Parameters description

- `percentage`: it is the percentage of output instances with respect to the total number of instances of the input table. The parameter `percentage` and the parameter `number_of_instances` are mutually exclusive; i.e. if this last parameter is specified by the user, the other must be omitted;
- `number_of_instances`: it is the absolute number, n, of output instances. Since the total number of input instances is known only at run-time, it can happen that $n > N$. In this case, if the *SRSWOR* technique is selected, then all input instances will be returned by the algorithm: no tuple replication is performed. Otherwise, if the *SRSWR* method is selected, then the output table will contain more instances that the input table and some instances will be replicated. If this parameter is specified by the user, the parameter `percentage` must be omitted.
- `with_replacement`: it selects the replacement strategy. It can be *true* or *false*. In the first case, the *SRDWR* method is selected. Otherwise, the algorithm uses the *SRSWOR* method for sampling.

Parameters specification

Table 4.21 contains the parameters specification for the simple sampling technique to be used in the `PARAM` XML element of figure 2.35.

Parameter Name	**Parameter Value**	**Usage**	**Default Value**
`percentage`	real in $(0, 1]$	optional	-
`number_of_instances`	positive integer	optional	-
`with_replacement`	$< true >$ $< false >$	optional	$< false >$

Table 4.21: `simple_sampling` parameters.

4.2.14 Stratified sampling

Algorithm Name

`stratified_sampling`

Description

Given a dataset with n instances and a nominal attribute A, containing M distinct values, that divides the dataset into M mutually disjoint parts called *strata*, the *stratified sampling* applies a simple random sampling at each stratum using a *with replacement* or a *without replacement* policy (see sect. 4.2.13), according to the parameter `with_replacement`. This helps to ensure a representative sample, especially when the data are skewed.
The number of output instances for each nominal category can be given either in absolute form (using the parameter `number_of_instances_per_category`) or as a percentage (using the parameter `percentage`) with respect to the number of instances for that category.
Currently the algorithm uses a proprietary Java implementation.

Input data format

The algorithm takes as input a preprocessing table containing at least a nominal field, representing the clustering attribute.

Parameters description

- `attribute_name`: it is the name of the stratified sampling attribute. Stratified sampling can be applied only to nominal attributes.
- `percentage`: it is the percentage of output instances for each category with respect to the total number of instances of that category. The parameter `percentage` and the parameter `number_of_instances_per_category` are mutually exclusive; i.e. if this parameter is specified by the user, the other one must be omitted;
- `number_of_instances_per_category`: it is the absolute number of output instances for each category. Since the total number of input instances is known only at run-time, it may happen that, for a category C_A, $k_{C_A} > K_{C_A}$, where k_{C_A} and K_{C_A} are, respectively, the total number of requested instances and the total number of input instances for that category. In this case, if the *SRSWOR* technique is selected, then all instances belonging to C_A will be returned by the algorithm: no tuple replication is performed. Otherwise, if the *SRSWR* method is selected, then the output table will contain more instances that the input table for C_A and some instances will be replicated. If this parameter is specified by the user, the parameter `percentage` must be omitted.
- `with_replacement`: it selects the replacement strategy to be applied at each stratum. The parameter can be *true* or *false*. In the first case, the *SRSWR* method is selected and instances can be replicated for a stratum. Otherwise, the algorithm uses the *SRSWOR* method for simple sampling.

Parameters specification

Table 4.22 contains the parameters specification for the stratified sampling technique to be used in the `PARAM` XML element of figure 2.35.

Parameter Name	**Parameter Value**	**Usage**	**Default Value**
`attribute_name`	string	required	-
`percentage`	real in $(0, 1]$	optional	-
`number_of_instances_per_category`	positive integer	optional	-
`with_replacement`	$< true >$ $< false >$	optional	$< false >$

Table 4.22: `stratified_sampling` parameters.

4.2.15 YaDT (Yet Another Decision Tree builder)

Algorithm Name

`YaDT`

Description

YadT (*Yet Another Decision Tree builder*) [5] is an efficient implementation of the entropy-based decision tree construction algorithm which vastly improves the computational performance over the well-known C4.5.
It has been designed and implemented in C++ with strong emphasis on efficiency (time and space) and portability (Windows/Linux).

Input data format

YaDT algorithm supports nominal or continuous attributes only: no string attributes are allowed in the input table.

Parameters description

- `num_instances_for_leaf`: the minimum cases to split a node.
- `confidence_for_pruning`: a real in the range (0,1] representing the pruning confidence level.
- `percentage_split`: it randomly splits training data in an actual training data and an additional test data. Default: no test data set.

Parameters specification

Table 4.23 contains the parameters specification for the YaDT algorithm to be used in the `PARAM` XML element of figure 2.35.

Parameter Name	**Parameter Value**	**Usage**	**Default Value**
`num_instances_for_leaf`	integer greater than 1	optional	2
`confidence_for_pruning`	real in $(0, 1]$	optional	0.25
`percentage_split`	real in $(0, 1]$	optional	1

Table 4.23: YaDT parameters

4.2.16 Z-score normalization

Algorithm Name

`z_score_normalization`

Description

The values of an attribute A are normalized on the basis of the mean and standard deviation of A. A value v of A is normalized to v' by computing

$$v' = \frac{v - E(A)}{Var(A)} \tag{4.3}$$

where $E(A)$ and $Var(A)$ are the mean and the standard deviation, respectively, of the attribute A.

Input data format

The algorithm takes as input a preprocessing table containing at least a numeric field, representing the discretization attribute.

Parameters description

None.

Parameters specification

None.

BIBLIOGRAPHY

[1] J. Han and M. Kamber, *Data Mining: Concepts and Techniques*. San Mateo, CA: Morgan Kaufmann, 2000.

[2] SPSS, "CRISP-DM step by step data mining guide," Version 1.0, 2000, `http://www.crisp-dm.org`.

[3] The Data Mining Group, "Predictive Model Markup Language (PMML)," Version 2.1, 2003, `http://www.dmg.org`.

[4] JSR-73 Expert Group, "Java Data Mining API," 2004, java Specification Request No. 73, `http://www.jcp.org/en/jsr/detail?id=73`.

[5] S. Ruggieri, "Efficient C4.5," *IEEE Trans. on Knowledge and Data Eng.*, vol. 14(2), pp. 438–444, 2002.

[6] W3C World Wide Web Consortium, "Extensible Markup Language (XML) 1.0 (second edition)," W3C Recommendation, 2000, `http://www.w3.org/TR/REC-xml`.

[7] T. Imielinski and H. Mannila, "A database perspective on knowledge discovery." *Commun. ACM*, vol. 39, no. 11, pp. 58–64, 1996.

[8] MacQueen, "Some methods for classification and analysis of multivariate observation," in *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, vol. 1. University of California Press, Berkeley, 1967, pp. 281–297.

[9] S.L.Lauritzen, "The em algorithm for grafical association models with missing data," *Computational Statics and Data Analysis*, vol. 19, pp. 191–201, 1995.

[10] G. Folino and G. Spezzano, "SPARROW: A spatial clustering algorithm using swarm intelligence," in *AIA'2003, Innsbruck*, 2003.

[11] W3C World Wide Web Consortium, "XQuery: XML Query Language," On-line documentation, 2004, `http://www.w3.org/XML/Query`.

[12] Quiz/Open, "XQuery open-source Java implementation," `http://www.xfra.net/qizxopen/`.

[13] J. Regex, `http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex`.

[14] G. Folino, C. Pizzuti, and G. Spezzano, "CAGE: A tool for parallel genetic programming applications," in *Genetic Programming, Proceedings of EuroGP'2001*, ser. LNCS, J. F. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon, Eds., vol. 2038. Springer-Verlag, 2001, pp. 64–73.

[15] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri, "Adaptive and resources-aware mining of frequent sets," in *IEEE ICDM Int. Conf. on Data Mining*. IEEE Computer Society, 2002, `http://hpc.isti.cnr.it/~palmeri/datam/DCI`.

[16] I. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan & Kaufmann, 2000, version 3.4.3 from `http://www.cs.waikato.ac.nz/ml/weka`.

[17] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M. Hsu, "Mining sequential patterns by pattern-growth: The prefixspan approach," *IEEE Trans. on Knowledge and Data Eng.*, vol. 16, no. 11, pp. 1424–1440, 2004.

APPENDIX A
Extending the KDDML language

In this appendix, we show how to extend the KDDML language in order to add new algorithms, new operators, or new models.

A.1 Adding a new algorithm

A data mining (resp. preprocessing) algorithm is a technique or procedure that, when applied to data, yields a model (resp. a preprocessing table). The set of algorithms is large and growing. As it is now, KDDML does not include a large set of algorithms, but it specifies a framework for including new ones. This enables users to provide additional algorithms and functionalities in advance of their inclusion in the standard version.

To this purpose, adding a new preprocessing, tree induction, clustering, association rules or sequential pattern mining algorithm should be as simple as possible. The tag `<ALGORITHM>` (see sect. 2.5) has been added to the KDDML language in order to represent algorithm settings used both for DM operators (such as `RDA_MINER, TREE_MINER`) and pre-processing operators (such as `PP_NUMERIC_DISCRETIZATION`). Notice that the algorithm name and parameters are not part of the language syntax: non-strict semantics for algorithm settings help us to preserve an high language extendibility. Correctness checking of parameters is not part of the parsing of the XML document representing the query, but it is demanded directly to the KDDML system.

In summary, adding a new pre-processing or mining algorithm in the future will not require any DTD modification to the KDDML language.

A.2 Adding a new operator

The extension of a new I/O, preprocessing, postprocessing or mining operator to the language syntax can be achieved through four steps, as shown below.

Step 1: definition of the operator signature

The first step to do when adding a new operator is to define an operator signature. An operator signature is a function

$$f : t_1 \times \ldots t_n \to t$$

that takes n KDDML objects as input and returns a KDDML object as output; the set of types of the KDDML language has been listed in the first column of the table 2.5. As one can expect, the signature depends on the kind of operator.

As shown in sect. 3.1.2, preprocessing operators take a preprocessing table as their first argument and return a preprocessing table as output. Other input arguments are not required, and they can be optionally specified by the user in order to use KDDML objects such as conditions or expressions defined on the input preprocessing table. Their signature is reported below:

$$f_{\texttt{<NEW\_PP\_OPERATOR>}} : \texttt{PPtable} \times \cdots \to \texttt{PPtable}.$$

By looking at the sect. 3.1.3, we can notice that a mining operator has a fixed signature. It takes a relational table as its first argument, and an algorithm setting as its second argument, and it produces a model as output. Its final signature is reported below:

$$f_{\texttt{<NEW\_MODEL\_MINER>}} : \texttt{table} \times \texttt{alg} \to \texttt{model}.$$

On the contrary, I/O and postprocessing operators have a generic specification.

Step 2: definition of a set of attributes

In an XML document, attributes often are not part of the data but they provide additional information about the elements. In KDDML, while XML tags correspond to arguments of an operator, XML attributes correspond to the parameters of the operator. Attribute-list declarations specify the name, data type, and default value (if any) of each attribute associated with a given element type (i.e. operator):

- *Name*: used to define the set of attributes pertaining to a given operator.
- *Usage*: can be required, fixed, implied; moreover, a default value can be specified. Possible values are:
  - `#REQUIRED`: means that the attribute must have a value every time this operator is listed;
  - `#FIXED`: the attribute is not required, but if it occurs, it must have the specified default value;
  - `#IMPLIED`: the attribute value is not required, and no default value is provided.

  The notation `"DEFAULT VALUE"` can be used in order to specify a default value for the attribute. In fact, an attribute can be given any legal value as a default. The attribute value is not required on each element in the document, but if it is not present, it will appear to be the specified default. If the attribute is not included in the element, the processing program assumes that this is the attribute value.

- *Type*: KDDML supports two types for operator attributes:
  - `%string;`: it may take any literal string as a value that includes also numbers or complex types such as lists or records (i.e. string attributes are strings where any text is allowed)[1].
  - $\{V_1|\ldots|V_n\}$: this represents the enumeration type in which attributes are defined by a list of acceptable pipe delimited values, from which the document author must choose. In this case, each of the values is explicitly enumerated in the declaration and the checking is performed directly by the XML parser.

Step 3: putting all together by defining a DTD

Steps one and two permit to define a Document Type Definition for the element representing the operator (take a look at the general operator structure of section 2.5).

The element root name corresponds to the function name.

The children definition have a one-to-one correspondence to the input types of the signature defined at step one (take again a look at the table 2.5 that shows the correspondence between types and XML entities). Every input type of the signature can be then replaced with the related entity in the DTD.

Finally, attributes are defined as in step two.

Step 4: declation of the new operator to KDDML

The last thing you have to do is to declare your operator to KDDML. This is performed by adding the operator to the right entity according to the kind of object it returns. In other terms, as soon as the referred entity has been detected, you must declare the new operator by adding a new entry (referring the operator tag name) to this entity.

A.3 Adding a new model

Adding a new mining model to the language means adding a new type to the operators signature, which amounts to non-destructive changes in the DTD of the operators.

Step 1: definition of the physical model

The first step is to define the physical format of the model. This target can be easily achieved by using the PMML v. 2.0. Extension mechanism of PMML is optional. It can be used in order to define proprietary features of a model, that are not supported by the PMML standard.

[1]As for algorithm parameters, correctness checking about string attributes is not part of the parsing of the XML document representing the query, but it is passed directly to the KDDML system.

Step 2: definition and inclusion of an XML entity in the KDDML language

Take again a look at the table 2.5 and to the figure 2.25. The entity `kdd_operator` contains an enumeration of all KDDML operators classified according to the output type they return. Adding a new mining model M in KDDML means adding a new entity (i.e. a new type) whose scope is to group all operators returning M as output. This entity can be then used where an operator returning the type M is required, according to the closure principle.

In order to declare the new object to KDDML, you must define an empty[2] entity, named, as instance, `new_model_entity`:

```
<!ENTITY % new_model_entity "(%kdd_query_object;)">
```

After this, you can add the entity to the language by adding a new entry in the `kdd_operator` entity, as reported below:

```
<!ENTITY % kdd_operator "(...|%new_model_entity;)">
```

Step 3: to define a set of operators

In order to manage the new model, you must provide a set of basic operators that, in general, permit to:

- read the physical model from the system repository or from an external PMML source;
- extract the model from a data source by using a mining algorithm;
- filter the new model;
- apply the model on a new data source in order to predict features or to select data;
- combine the model with other models.

New operators can be added to the KDDML language as described in the previous section of this appendix. Remember that, for each operator provided, you must add it to the required entity, according to the output type the operator returns, as specified at step 4 of the previous section.

A.4 Example: adding a naive bayes model

The naive bayes classifier technique is based on the so-called bayesian theorem and it is particularly suited when the dimensionality of the inputs is high. It is based on an assumption of conditional independence, to predict the value of a target (output) independence, from evidence given by one or more predictor (input) fields.

A.4.1 Physical model

PMML 2.0 naive bayes model. No extension mechanism used.

[2]The entity contains only operators returning a generic KDDML object.

A.4.2 Model entity

```
<!ENTITY % kdd_query_naive_bayes "(%kdd_query_object;)">

<!ENTITY % kdd_operator
    "(%kdd_query_clusters;|%kdd_query_rules;|%kdd_query_sequence;|
      %kdd_query_table;|%kdd_query_trees;|%kdd_query_hierarchy;|
      %kdd_query_scalar;|%kdd_query_PPtable;|%kdd_query_naive_bayes;)">
```

A.4.3 Operators

A.4.4 Operator one: model loader

It loads a naive bayes model from the system repository.

Signature

$f_{<\texttt{NAIVE\_BAYES\_LOADER}>} : empty \rightarrow \texttt{naive\_bayes}$.

Attributes

- `xml_source`.
  - type: string;
  - usage: required.

DTD

```
<!ELEMENT NAIVE_BAYES_LOADER EMPTY>
<!ATTLIST NAIVE_BAYES_LOADER xml_source %string; #REQUIRED>
```

Target entity entry

`kdd_query_naive_bayes`.

A.4.5 Operator two: model miner

It extracts a naive Bayes model from a data source by using an algorithm setting and a target attribute.

Signature

$f_{<\texttt{NAIVE\_BAYES\_MINER}>} : \texttt{table} \times \texttt{alg} \rightarrow \texttt{naive\_bayes}$.

Attributes

- `xml_dest`.
  - type: string;
  - usage: implied.
- `target_attribute`.
  - type: string;
  - usage: required.

DTD

```
<!ELEMENT NAIVE_BAYES_MINER ((%kdd_query_table;), ALGORITHM)>
<!ATTLIST NAIVE_BAYES_MINER xml_dest %string; #IMPLIED>
<!ATTLIST NAIVE_BAYES_MINER target_attribute %string; #REQUIRED>
```

Target entity entry

`kdd_query_naive_bayes`.

A.4.6 Operator three: classifier

It classifies an input data source using a naive bayes model.

Signature

$f_{\texttt{<NAIVE\_BAYES\_CLASSIFY>}}$: $\texttt{naive\_bayes} \times \texttt{table} \longrightarrow \texttt{naive\_bayes}$.

Attributes

- `xml_dest`.
  - type: string;
  - usage: implied.

DTD

```
<!ELEMENT NAIVE_BAYES_CLASSIFY ((%kdd_query_naive_bayes;),
                                (%kdd_query_table;))>
<!ATTLIST NAIVE_BAYES_CLASSIFY xml_dest %string; #IMPLIED>
```

Target entity entry

`kdd_query_table`.