KDDML: a middleware language and system for knowledge discovery in databases

Andrea Romei Salvatore Ruggieri * Franco Turini

Dipartimento di Informatica, Università di Pisa
Largo Bruno Pontecorvo 3, 56127 Pisa, Italy

Abstract

KDDML (KDD Markup Language) is a middleware language and system designed to support the development of final applications or higher level systems which deploy a mixture of data access, data preprocessing, extraction and deployment of data mining models.

We present our three-years' experience in the development of KDDML. The design principles are motivated by requirements derived from recurring patterns in the KDD process.

The KDDML language is XML-based, both for query syntax and data/model representation. A KDDML query is an XML-document where XML tags correspond to operations on data/models, XML attributes correspond to parameters of those operations and XML sub-elements define arguments passed to the operators. We present the operators for data access and preprocessing, model extraction and deployment, and control flow ones.

The core of the KDDML system is a KDDML language interpreter with modularity and extensibility requirements as the main goals. Additional data sources, and preprocessing and mining algorithms can be easily plugged in the system.

Key words: Knowledge Discovery in Databases, Data Mining, Query Languages

1 Introduction

Knowledge discovery in databases (KDD) [10] covers a wide range of applicative domains (retail, marketing, finance, e-commerce, biology, privacy, only

* Corresponding author. Tel.: +39-050-2212782; fax: +39-050-2212726.
Email address: ruggieri@di.unipi.it (Salvatore Ruggieri).

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.

The KDD process, i.e. the process of finding "nuggets" of knowledge, is a complex task, heavily dependent on the problem and on the data at hand. As described in the CRISP-DM process model [32], it may consists of several repeated phases including business problem understanding, data understanding, 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 in a complex process, requiring to combine different sources of data and knowledge, and with many tasks iterated in order to reach a local optimum.

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 and RDBMS offering KDD algorithms. On the contrary, the design of final applications is still an "art", aimed at smoothly composing algorithm libraries, proprietary API's, SQL queries and stored procedure calls to RDBMS, and *much much* code.

There is a fervent activity of standardization in the area of mining model representation and access, and of mining algorithms API's [16,17,22,34]. We briefly overview those activities in Sect. 2. Nevertheless, 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 data access, data preprocessing, mining extraction and deployment. In this context, XML appears as a bridge between database technology and data mining tools. However, its use in existing tools appears to be limited to the exchange of mining models between applications. For instance, this is the prevalent use of the Predictive Markup Modelling Language (PMML) standard [34]. We would like to go further and conceive a language and system where XML is used for model and data representation and exchange, as well as for defining data and model processing tasks. XML seems a natural candidate to the purpose. First, modularity and extensibility of XML can be exploited in the design of the language and system, which must necessarily take into account an ever growing collection of models and algorithms. Second, XML is largely adopted as a machine-processable language, e.g. in the web technology or in transaction processing. This favor its use in a *middleware* framework, on top of which higher abstraction levels or final applications can be built.

In this paper, we present our three-years' experience in the design and development of KDDML, an XML-based middleware language and system in support of the KDD process.

In Sect. 3, we derive some requirements on middleware KDD systems from typical patterns of operations arising in KDD processes. Requirements cover data and model staging areas, external data and model access, meta-data management, compositionality of operations, independence of the language from specific implementations of extraction algorithms, extensibility of the system to cover different implementations.

The KDDML language is presented in Sect. 4. The syntax is XML-based, so to favor machine-processability, with each tag modelling an operator of the language. However, the semantics is purely "functional", which ensures compositionality of operators. 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 PMML standard.

In Sect. 5, the general architecture of the KDDML system is presented, consisting of layers for data/model repository, operators implementation and query interpretation. Modularity and extensibility are a must in the design of the system. The KDDML system includes a GUI for user friendly input of queries and for browsing extracted knowledge.

Summarizing, starting from a discussion of the issues involved in modelling and standardizing recurring patterns in the KDD process, we present in this paper a language and a system that support the development of final applications or higher abstraction layers using those patterns. As a middleware language and system, KDDML is machine-processable, opened to external data/model sources, easily extensible with new algorithms and data sources.

2 Background: querying and processing data and models

As a background to our presentation, let us recall the basic tools, technologies and standards on querying and processing data and mining models. We are mainly interested in query languages and in mechanisms for processing (accessing, querying, navigating results, etc.) relational data, XML documents and data mining models. We will be taking the view of a developer of a final KDD application that uses those query languages and process mechanisms as building blocks.

2.1 Relational databases

Relational data is the win-win model for transaction processing. As far as retrieving is concerned, a *query language* is usually adopted to denote the needed subset of data. Query languages are required to be both expressive in denoting the desired result and efficient to compute it – two contrasting requirements.

2.1.1 Querying

Structured Query Language (SQL) is the widely recognized query language for relational databases, with a solid standardization [13,14] and all major RDBMS conforming to it. SQL is a substantial tool of the data preprocessing phase in the KDD process.

The SQL-99 standard does not only cover the classic `SELECT-FROM-WHERE` pattern, but also triggers, transactions, recursion, structured and user defined data types, procedural language extensions, function and procedure definition and storage, OLAP extensions, embedding in general purpose programming languages, and call level interface (SQL/CLI), that is an API for invoking functionalities of SQL-engines.

2.1.2 Processing

Data access and processing from client applications can be achieved at a basic level using the SQL/CLI interface: for instance, IBM DB2 provides it as the native interface. An industry standard close to SQL/CLI is the Open Database Connectivity (ODBC) specification. ODBC adopts SQL as its database access language. Alternative industry standards include:

- the Java Database Connectivity (JDBC) specification, which provides a high level object hierarchy for connecting to data sources, querying them via SQL, retrieving results, mapping SQL types to Java types, accessing table meta data;
- the Microsoft OLE DB specification, which allows for accessing both relational and non-relational data sources, such as XML and OLAP for instance, and on top of it, the ActiveX Data Object (ADO) framework, which provides a high level object hierarchy.

It is worth noting that all of them have a bridge towards ODBC, which is then a minimum requirement for relational database inter-operability.

Data processing in Java from inside SQL has been recently standardized in [15], where Java static methods (resp., classes) can be invoked directly from SQL (resp., viewed as user defined data types). Analogously, a tight integration of SQL and Microsoft .NET programming languages has been announced for the forthcoming Microsoft SQL Server 2005.

2.2 XML

The *eXtensible Markup Language* (XML) [35,36] is the rapidly emerging standard for representing semi-structured data. XML facilitates exchange of data between systems since documents are self-describing, i.e. they don't need to conform to the internal layout of any system but only to an agreed exchange format. Such a format is specified as a *document type definition* (DTD) [36] or as an *XML Schema*[1] definition [37] associated to the exchanged XML document.

On the one side, XML is then another (inter-operable) representation of data (in addition to flat files, archives, databases, etc.) On the other side, XML is a natural candidate for representation and exchange of extracted mining models.

2.2.1 Querying

A relevant on-going effort of the W3C organization is the design of a standard query language for XML, called XQuery [8,38], which is built on intense research in the field and for which a large number of implementations already exists.

2.2.2 Processing

API's for navigating XML documents include the Document Object Model (DOM) and the Simple API for XML (SAX). DOM provides a main-memory hierarchical (tree-like) object oriented model of XML documents. SAX defines an event-based approach (more suitable for external-memory processing) whereby parsers scan through XML data, calling handler functions whenever certain patterns are found. As far as XQuery is concerned, there is no agreed API, yet several proposals and implementations exist, such as Java XQuery API.

[1] A DTD specifies the grammar of an XML document, but uses only few simple built-in data types. XML Schemas have richer data types (including user defined ones) and grammar rules.

2.2.3 XML, relational data and SQL

On the one side, transforming relational data into XML and back is easy to achieve with existing commercial RDBMS and middleware products. Unfortunately, each product provides its own DTD or schema. A standardization effort towards an official `sqlxml` schema is currently addressed by the SQL-200n draft standard [14].

On the other side, commercial RDBMS also provide "XML" data types for table columns, extending SQL with operators for XML querying via XQuery or a subset of it. As before, however, there is no standard extension of SQL dealing with this issue.

2.3 Data mining

Data mining models represent knowledge (in the form of association rules, sequences, classifiers, clustering, and several other patterns [10]) extracted from massive amount of data, typically stored in databases but also in flat files, in OLAP databases, or in data streams.

An industry standard for actual models representation as XML documents is the Predictive Model Markup Language (PMML) [34]. PMML consists of DTDs for a wide spectrum of models, including association rules, decision trees, clustering, naive-bayes, regression, neural networks. 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.

2.3.1 Querying

Since data and mining models are so tightly related, it is natural to integrate data mining algorithms with SQL. This line of research has been pursued by MineSQL [20], MSQL [12], NonStop SQL/MX [7] and Mine Rule [19]. These approaches extend SQL with operators for extracting models from relational data. On the one side, this amounts to add data mining *processing* features to SQL. On the other side, in order to maintain the closure principle of SQL (the results of a query can be queried themselves), the data mining models considered in the mentioned approaches were represented as relations. For instance, Microsoft OLE DB for Data Mining [21] considers models that are graphs, but allows the navigation by means of a binary relation that represents arcs between nodes.

Recently, the standardization of the integration of data mining algorithms

with SQL is being addressed in the SQL/MM Data Mining draft [16]. The proposed draft distinguishes the phases of training, testing and applications for association rules, clustering, classification, and other models.

2.3.2 Processing

The state-of-the-art of API's for data mining consists of a plethora of vendor-specific proposals typically tied to proprietary connection mechanisms, query languages, programming languages and libraries. On the open-source side, the Java-based tool Weka [39] can be claimed as the mostly adopted tool. Its API of algorithms is then a general reference worldwide.

Recently, however, the Java Data Mining API specification [17] has been released, which represent a solid competitor to Weka and vendor specific API's. JDM distinguishes API's for accessing models provided by a Data Mining Engine (which builds models) and a Metadata Repository (which persistently stores models). The overall structure of the Metadata Repository complies with a UML specification described by the Common Warehouse Metamodel [22] (Chapter 15 – Data Mining).

A higher level of developing KDD applications is to use a visual metaphora. Research and commercial tools [18,23,30,33,39] typically offer a GUI where the KDD process can be specified by means of a graph. Each node of the graph represents a task (data source access, preprocessing operation, model extraction, model application, model evaluation, visualization) and each arc represents the flow of data/models between tasks. The visual metaphora is really helpful in the trial-and-error development of application. However, it does not solve the issue of API's for data mining, since visual GUI are typically built on top of an (standard or proprietary) API.

3 Modelling the KDD process: challenges and requirements

Before starting presenting the KDDML language and system, it is worth discussing whether database and data mining (industrial or research) systems at the state-of-the-art are sufficiently expressive to cope with usage patterns that commonly arise in KDD processes. In order to be as concrete as possible, throughout this section we will target examples on a specific framework, namely OLE DB for Data Mining [21,31]. Notice, however, that commercial competitors (such as IBM DB2, Oracle 10g, SPSS Clementine, etc.) and alternative research proposals (see references cited in Section 2.3) offer comparable features.

```
SELECT A.id, A.age, A.education, T.day_of_week, T.brand, T.amount
FROM transactions T,
      OPENROWSET('IBMDADB2.1',
            'User ID=account;Password=***;
             Data Source=database;
             Location=host.univ.edu:50000',
        'SELECT *  FROM census') AS A
WHERE T.client_id = A.id
```

Fig. 1. Joining local and external tables through the `OPENROWSET` function.

3.1 Requirements on accessing physical data

Using relational database connectivity standards (such as ODBC, JDBC and OLE DB), RDBMS can transparently access tables on (local or remote) external sources, such as relational tables, flat files and spreadsheets.

As an example, Microsoft SQL Server 2000 offers the possibility for dynamic connections to external data sources via the `OPENROWSET(`*driver*`,` *datasource*`,` *query*`)` function, which returns the result set of the SQL query *query* forwarded to a data source *datasource* via the OLE DB driver *driver*. Fig. 1 shows that joining data between a linked table and a local table is a totally transparent task. The query selects a table suitable for input to a classification algorithm intended to build a model of the expenses behavior of customers on the basis of their age, work-class and education (as provided by the remote table `census`), and on the basis of the brand and day of week of purchase (as provided by the local table `transactions`).

Summarizing, the database connectivity standards are essential tools for gathering data from multiple sources, and they are commonly available features of data mining suites.

Question: are then RDBMS connectivity standard a thorough answer to the physical data access requirements of the KDD process?

There are three main issues not covered by the connectivity standards.

First, data is not only an input to the KDD process, but also an intermediate output and a final output. As an example, a classification/regression model could be used to fill missing field values of corrupted buying transactions. An intermediate table is then necessary to store the recovered transactions, before using them further, e.g. before submitting them to a time series analysis algorithm for predicting future levels of purchases of a particular brand.

Requirement R1: a data repository should be available as a "data staging" area for storing input, output and intermediate data of the KDD process.

This requirement is explicitly fulfilled by most of the data mining systems, which allow for storing intermediate results on relational tables, flat files or in proprietary format.

Second, high performance data mining algorithms very often require in input a specialized organization of data on external memory. These organizations allows for optimizing scans of data, for indexing, and for efficient computing of aggregates.

Requirement R2: the data repository should allow for accepting/providing data in diverse representations, automatically performing format conversions.

Third, meta-data provided by external data sources may vary considerably. For instance, in the ARFF format adopted by the Weka system [39], the meta-data of the table resulting from the query in Fig. 1 would be described as follows.

```
@attribute id ignore
@attribute age integer
@attribute education string
@attribute day_of_week string
@attribute brand string
@attribute amount real
```

This is a much more compact representation than the one provided by SQL meta-data, where more specific information is available on: data types (`integer` vs `SMALLINT`, `INT`, `BIGINT`), indexes and primary keys (is `id` simply to be ignored or a primary key useful for fast search and scan?), constraints (is "-1" an admissible value for `age`?), default values, NULL values, derived columns, etc.

Requirement R3: the data repository should allow for accepting/providing meta-data in diverse representations, automatically performing meta-data mapping.

3.2 Requirements on specifying logical data

Data connectivity standards offer API's for connecting to a data source, for issuing SQL queries, for navigating returned recordsets, and for accessing database and recordset meta-data. However, this level of API's can be considered as the *physical* level [17]. A higher abstraction level concerns *logical data*, i.e. domains of data to be used as input to data mining operations in order

```
<DataDictionary>
   <DataField name="id" optype="continuous" dataType="integer"/>
   <DataField name="age" optype="continuous" dataType="integer"/>
   <DataField name="education" optype="categorical" dataType="string"/>
   <DataField name="day_of_week" optype="categorical" taxonomy="Day"/>
   <DataField name="brand" optype="categorical"/>
   <DataField name="amount" optype="continuous"  dataType="float"/>
 </DataDictionary>
 <Taxonomy name="Day">
   <ChildParent childColumn="day" parentColumn="working" >
       <InlineTable>
           <row><day>Monday</day><working>yes</working></row>
           <row><day>Tuesday</day><working>yes</working></row>
           <row><day>Wednesday</day><working>yes</working></row>
           <row><day>Thursday</day><working>yes</working></row>
           <row><day>Friday</day><working>yes</working></row>
           <row><day>Saturday</day><working>no</working></row>
           <row><day>Sunday</day><working>no</working></row>
       </InlineTable>
   </ChildParent>
 </Taxonomy>
```

Fig. 2. Sample PMML fragment describing table meta-data.

to specify the type of usage of attributes in building and applying a mining model.

A classical distinction is made between discrete and continuous attributes: discrete ones include binary, nominal, categorical and ordinal values; continuous ones include interval-scaled and ratio-scaled values.

Taxonomies (or hierarchies) are another logical data element. While from the physical point of view they are tables, from the logical point of view they model domain-knowledge by defining hierarchies exploited by data mining algorithms. Also, weights (or probabilities) are logical data elements that make sense in affecting the role of an attribute or of an input row in the construction of a model.

In PMML, the `Taxonomy` and `DataDictionary` tags define the logical level of extracted models. Fig. 2 shows a fragment from a PMML model describing metadata for the data of Fig. 1.

Notice that the `dataType` attribute is optional, and refers to the physical level of the related column. Also, the `taxonomy` attribute refers to the `<Taxonomy>` tag, which contains the corresponding hierarchy of values as a child-father

```
CREATE MINING MODEL [censusTree]
  ([id] LONG KEY,
   [age] LONG CONTINUOUS,
   [education] TEXT DISCRETE,
   [day_of_week] TEXT DISCRETE,
   [brand] TEXT DISCRETE,
   [amount] DOUBLE PREDICT)
USING Microsoft_Decision_Trees
```

Fig. 3. Meta-data (physical & logical) for a decision tree model in OLE DB for DM.

table.

Question: how do we specify elements at the logical level?

At the worst, the logical level specification may be completely induced by the physical one, as it happens in the Weka system where numeric columns of ARFF files are considered of continuous type, string columns are considered of discrete type, and the last attribute is the one to be predicted.

More typically, logical metadata is specified by the user. Also, it is quite common that during the KDD process several different specifications are tried in order to test performance of extracted knowledge by varying weights, types, taxonomies.

Consider again OLE DB for Data Mining. Meta-data are specified in the definition of the model, as shown in Fig. 3. Here, a richer set of types than PMML is available (other types not shown in the example include `ORDERED`, `CYCLICAL`, `SEQUENCE`).

Requirement R4: the language and system should allow for specifying logical metadata (such as attribute type, taxonomies, weights) in addition to physical metadata.

3.3 Requirements on mining model representation

As for data, extracted mining models (either the final or the intermediate ones of a complex KDD process) should be stored in an appropriate repository, for subsequent analysis or application. For instance, a classification model may be visually presented to a domain expert, or it may be applied on unseen data to predict unknown classes, or it can be input to an incremental classification algorithm that revise it based on additional training data.

Requirement R5: a model repository should be available as a "model staging area" for storing input, output and intermediate models of the KDD process.

A question naturally arises: how should models be represented in the repository?

Data mining suites typically adopt a proprietary binary representation, providing API's for access from external programs and allowing for import/export in some interchange format.

The XML-based Predictive Modelling Markup Language (PMML) [34] is a widely adopted interchange format. As an example, consider the decision tree built on data resulting from the SQL query of Fig. 3. Its PMML representation would start with application that generated the model, logical metadata, model name and metadata (attribute split type for decision trees), attribute usage (active, predicted or other use), and the actual model (a decision tree) – as shown in Fig. 4.

Import/export between proprietary formats and PMML can be achieved in the OLE DB for DM framework as shown in Fig. 5. Once exported to PMML format, mining models can be queried via XML query languages or navigated by visual tools, such as IBM Intelligent Miner for Visualization [11].

Question: can PMML be considered for data mining as the equivalent of data connectivity standards for relational database?

Unfortunately, no – not for the moment, at least. At the present PMML covers only the core of data mining models, possibly missing some issues. As an example, confusion (or misclassification) matrix [10] of a classification model is not defined in PMML. Therefore, two PMML-compliant systems exporting confusion matrix in different formats may not yield interchangeable models.

However, notice that PMML offers an `Extension` tag, allowing for including additional contents in the model representation.

Requirement R6: the model repository should allow for accepting/providing model meta-data in diverse representations, automatically performing meta-data mapping.

3.4 Requirements on accessing mining models

A data mining model represents knowledge extracted from data and, possibly, other models. A data mining API should allow for defining a new model, for populating it by extracting knowledge from data, for accessing the knowledge

```
<PMML version="2.0">
 <Header>
  <Application name="YaDT" version="1.0.0"/>
 </Header>
 <DataDictionary>
    <DataField name="id" optype="continuous" />
    <DataField name="age" optype="continuous" />
    <DataField name="education" optype="categorical" />
    <DataField name="day_of_week" optype="categorical" />
    <DataField name="brand" optype="categorical" />
    <DataField name="amount" optype="continuous" />
  </DataDictionary>
  <TreeModel modelName="censusTree" splitCharacteristic="multiSplit">
    <MiningSchema>
      <MiningField name="it" usageType="supplementary" />
      <MiningField name="age" usageType="active" />
      <MiningField name="education" usageType="active" />
      <MiningField name="day_of_week-num" usageType="active" />
      <MiningField name="brand-status" usageType="active" />
      <MiningField name="class" usageType="predicted" />
  </MiningSchema>
  <Node score="" recordCount="48842">
    <True/>
    <ScoreDistribution value="<=50K" recordCount ="37155" />
    <ScoreDistribution value=">50K" recordCount ="11687" />
    <Node score="" recordCount="46787">
        <SimplePredicate field="capital-gain"
                         operator="lessOrEqual" value="6849" />
        <ScoreDistribution value="<=50K" recordCount ="37127" />
        <ScoreDistribution value=">50K" recordCount ="9660 />
    ...
    </Node>
  ...
  </Node>
</PMML>
```

Fig. 4. PMML fragment describing a decision tree model.

in the model and, for predictive models, predicting values based on model knowledge.

At the state-of-the-art, data mining suites (such as Weka [39], Oracle [23], IBM Intelligent Miner [11], Xelopes [26] and the forthcoming JDM [17]) offer API's for navigating models from inside programs or for issuing queries in proprietary query languages.

```
CREATE MINING MODEL <model_name>
FROM PMML <pmml_string>

SELECT MODEL_PMML
FROM <model_name>.PMML
```

Fig. 5. PMML import/export of a mining model in OLE DB for DM.

```
INSERT INTO [censusTree] (id, age, education, day_of_week, brand, amount)
   OPENROWSET('IBMDADB2.1',
          'User ID=account;Password=***;
           Data Source=database;
           Location=host.univ.edu:50000',
          'SELECT id, age, education, day_of_week, brand, amount
           FROM census')
```

Fig. 6. Training a classification model in OLE DB for DM.

For instance, in the OLE DB for DM framework, the `CREATE MINING MODEL` of Fig. 3 allows for defining a new decision tree model on given attributes using the `Microsoft_Decision_Trees` algorithm. Populating a model means extracting knowledge from data and inserting it in the model. If the model already exists, this amounts to incrementally updating the model. An `INSERT INTO` statement is shown in Fig. 6 for training a classification model.

Accessing the contents of a model can take place either as a PMML document, as already observed, or as a SQL table. In the latter case, SQL can be used for querying model contents. As an example, a decision tree is represented with a row for each node and with the following columns: node unique identifier, father unique identifier, children cardinality, attribute selected at the node, statistical information at the node, etc.

Finally, it is possible to query a model for predictions and, possibly, to test the answers against known values, in order to estimate model accuracy. In the decision tree example, the model can classify the amount of expenses of new customers based on their age, education, day of week of purchase and brands. The `PREDICTION JOIN` syntax of OLE DB for DM shown in Fig. 7 expresses that. Here, the `OPENROWSET` function is intended for accessing a table with the predictive columns of new customers, while the `PREDICTION JOIN` is intended to join such a table with the (intensionally defined) table of all predictions of the `censusTree` model.

```
SELECT D.id, [censusTree].amount
FROM [censusTree] AS T
       PREDICTION JOIN
     OPENROWSET( ... ) As D
       ON T.age = D.age AND T.education = D.education AND
          T.day_of_week = D.day_of_week AND T.brand = D.brand
```

Fig. 7. Prediction of a classification model in OLE DB for DM.

Question: is the result of the query in Fig. 7 reusable as an input to another data mining or preprocessing algorithm?

In other words, does OLE DB for DM ensure compositionality of mining operations? The answer is yes. The result of the query in Fig. 7 is accessible from any OLE DB consumer, and then from another data mining operator via the `OPENROWSET` function.

Requirement R7: compositionaly of mining operations should be pursued in the design of a middleware KDD language.

Unfortunately, however, the highlighted limits of present standards for model representation and API do not allow for having compositionality with other KDD frameworks. This is somewhat different from what happens in the relational databases world, where transparent integration can be achieved in accessing/querying external data sources.

Requirement R8: compositionaly of data mining systems should be pursued in the design of a middleware KDD system.

4 KDDML: KDD Markup Language

The challenges and requirements of the previous section have motivated us in the design of a (middleware) language and system in support of the KDD process. In this section, we present the KDDML language, where the acronym stands for *KDD Markup Language*. In the next section, we will overview the KDDML system, which implements an interpreter of the language.

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

```
<KDD_QUERY name="sample">
   <TREE_CLASSIFY xml_dest="results.xml">
      <TREE_MINER xml_dest="tree.xml" target_attribute="class">
         <TABLE_LOADER xml_source="trainingSet.xml"/>
         <ALGORITHM algorithm_name="YADT">
            <PARAM name="confidence_for_pruning" value="0.4"/>
            <PARAM name="num_instances_for_leaf" value="3"/>
          </ALGORITHM>
       </TREE_MINER>
      <TABLE_LOADER xml_source="testSet.xml"/>
   <TREE_CLASSIFY>
</KDD_QUERY>
```

Fig. 8. A sample KDDML query.

implementations of data mining algorithms, with the aim of confining the technicalities at the level of the implementation of the KDDML system.

The KDDML language assumes a *data repository*, containing relational tables, a *model repository*, containing mining models, and a *query repository*, containing queries. Tables, models and queries can be referenced by an identifier [2]. 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. As an example, the query of Fig. 8 specifies the construction and the application of a decision tree.

The root tag is `<KDD_QUERY>`, with the query identifier 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 decision tree to be applied is provided by the first sub-element (with tag `<TREE_MINER>`) which specifies the construction of a decision tree. The test set is provided by the second element (with tag `<TABLE_LOADER>`), which specifies a table in the data repository. In turn, the construction of a decision tree (tag `<TREE_MINER>`) takes place on a training set `trainingSet.xml` from the data repository by applying a decision tree induction algorithm (here, YADT from [29]) with parameters concerning the pruning strategy of the algorithm. The name of the class attribute is provided as attribute of the `<TREE_MINER>` element. As it will be shown later on, the

[2] In actual implementation, the identifier coincides with the name of the file in the repository.

KDDML systems embeds a library of algorithms and basic mechanisms for adding new ones.

As one could expect, arguments to 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 $f : t_1 \times \ldots \times t_n \rightarrow t$ returning type t by defining a DTD for KDDML queries that constraints sub-elements to be of type t_1, ..., t_n. Thus, KDDML queries corresponds to terms in the algebra of operators, though syntactically represented as XML documents.

The set of types of KDDML operators consists of: `table`, `tree`, `rda`, `sequences`, `clusters`, `hierarchy`, `scalar`, `algs` and `xml`. Intuitively, there is one type for data sources, one type for each mining model (`tree`, `rda`, `sequences`, `clusters`), one type for hierachies, one type for algorithms (`algs`) and one for operators that return a scalar (i.e., a number or a string). Finally, the `xml` type denotes arguments that are generic XML elements to be evaluated directly by the operator.

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) recursive evaluation of fragments from `<ARG1_NAME> ... </ARG1_NAME>` to `<ARGn_NAME> .... </ARGn_NAME>`; in case the ith argument of `<OPERATOR_NAME>` is expected of type `xml`, the element `<ARGi_NAME> ... </ARGi_NAME>` is itself the result of its evaluation;
(2) evaluation of attributes `att1 ... attM` returning a set of scalar values;
(3) a call to an operator $f_{\texttt{OPERATOR\_NAME}}$, accepting results from (1) and (2) and yielding the final result of the fragment.

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, the following fragment of a 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 `kdd_query_table` (i.e. all operators returning a table). The DTD is then another (simple and general) way of specifying an algebra of types and operators.

The set of operators can be classified on the basis of the type they return. However, we prefer to follow a different presentation. In the rest of this section, we discuss first the representation of data and the operators to access data. Next, we present the representation of mining models and operators to extract and deploy them. Finally, we introduce operators for control flow.

4.1 Data access and preprocessing

Data format

The KDDML language assumes a *data repository*, containing tables that can be referenced by an identifier. A table is represented as an XML file, containing a *schema* and a reference to the actual data, which is stored in CSV (Comma Separated Values) format. In principle, however, the coding of actual data can assume any format: CSV has been chosen here as a trade-off between readability (vs binary files) and space occupation (vs XML).

Fig. 9 shows the XML document describing the data set from the query of Fig. 1, which from now on we will name `census.xml`. The `data_file` attribute of the `<KDDML_TABLE>` tag refers the location of physical data. The XML document specify metadata information. As it is readily checked, metadata on attributes include attribute type (nominal, numeric or string) and some simple statistics on attribute values: cardinality of each value for nominal attributes; min, max, mean and variance for numeric attributes.

Data access

The data repository is populated by KDDML queries that yield tables as output. As one could expect, data access operators are basically available to access RDBMS (using SQL SELECT queries) and text files in the ARFF format.

```
<KDDML_TABLE data_file="census.csv">
   <SCHEMA logical_name="census" number_of_attributes="6"
           number_of_instances="16">
      <ATTRIBUTE name="id" number_of_missed_values="0"
                 type="numeric">
         <NUMERIC_DESCRIPTION mean="8.5" variance="22.67"
                              min="1.0" max="16.0"/>
      </ATTRIBUTE>
      <ATTRIBUTE name="age" number_of_missed_values="0"
                 type="numeric">
         <NUMERIC_DESCRIPTION mean="40.75" variance="237.8"
                              min="18.0" max="70.0"/>
      </ATTRIBUTE>
      <ATTRIBUTE name="education" number_of_missed_values="3"
                 type="nominal">
         <NOMINAL_DESCRIPTION number_of_values="4">
            <VALUE value="doctorate" cardinality="1"/>
            <VALUE value="bachelors" cardinality="7"/>
            <VALUE value="HS-grad" cardinality="3"/>
            <VALUE value="masters" cardinality="2"/>
         </NOMINAL_DESCRIPTION>
      </ATTRIBUTE>
      <ATTRIBUTE name="day_of_week" number_of_missed_values="0"
                 type="nominal">
         <NOMINAL_DESCRIPTION number_of_values="7">
            <VALUE value="fri" cardinality="4"/>
            <VALUE value="thu" cardinality="1"/>
            <VALUE value="wen" cardinality="0"/>
            <VALUE value="sat" cardinality="4"/>
            <VALUE value="sun" cardinality="0"/>
            <VALUE value="mon" cardinality="5"/>
            <VALUE value="tue" cardinality="2"/>
         </NOMINAL_DESCRIPTION>
       </ATTRIBUTE>
       <ATTRIBUTE name="brand" number_of_missed_values="0"
                  type="string">
          <STRING_DESCRIPTION/>
       </ATTRIBUTE>
       <ATTRIBUTE name="amount" number_of_missed_values="0"
                  type="numeric">
          <NUMERIC_DESCRIPTION mean="906.25" variance="557958.33"
                               min="100.0" max="2500.0"/>
       </ATTRIBUTE>
   </SCHEMA>
</KDDML_TABLE>
```

Fig. 9. `census.xml`: sample data representation in KDDML.

```
<ARFF_LOADER xml_dest="BasketData.xml"
             arff_file_path="D:/Repository/BasketData.arff"/>

<DATABASE_LOADER xml_dest="BasketData.xml"
                 database_name="jdbc::odbc::basketDB"
                 sql_query="SELECT * FROM BasketData"/>

<TABLE_LOADER  xml_source="BasketData.xml"/>
```

In the first (resp. second) query fragment, an ARFF table (resp. database table) is accessed, transformed into the internal representation, saved into the repository (recall that, however, the `xml_dest` attribute is optional), and finally passed up to the father node of the query fragment. Mapping from ARFF (resp., SQL) data types to the logical data types of the XML representation is automatic. However, preprocessing operators (see later on) allows for specifying different logical types of attributes of loaded tables.

In the third example, a table already in the data repository is passed up to the father node.

Export of tables to database and ARFF formats is achieved by the `<DATABASE_WRITER>` and `<ARFF_WRITER>` operators, as in the following fragment:

```
<DATABASE_WRITER database_name="jdbc:odbc:basketDB",
                 table_name="BasketData">
   <TABLE_LOADER xml_source="BasketData.xml"/>
</DATABASE_WRITER>
```

Data preprocessing

Data preprocessing [27] is a time-consuming phase of the KDD process, including tasks such as data selection, filtering, merging, cleaning, discretization, sorting, aggregating and many others. KDDML offers some operators for data preprocessing, yet they are the latest addition to the language and system. Their typing is quite intuitive, typically requiring a table as an argument.

As a simple example, the following fragment removes the `age` attribute from the input table. Its typing is $f_{\text{PP\_FILTER\_ATTRIBUTES}} : \texttt{table} \rightarrow \texttt{table}$.

```
<PP_FILTER_ATTRIBUTES xml_dest="census_removed.xml"
                      attributes_list="age"
                      take_or_remove="remove">
  <TABLE_LOADER xml_source="census.xml"/>
</PP_FILTER_ATTRIBUTES>
```

On the contrary, new attributes can be added by the operator PP_NEW_ATTRIBUTE In this case, the attribute is derived from existing ones by means of a simple expression language. Also, the type of the derived attribute can be set. Its typing is $f_{\texttt{PP\_NEW\_ATTRIBUTE}} : \texttt{table} \times \texttt{xml} \rightarrow \texttt{table}$. The second argument is an EXPRESSION tag, which is directly interpreted by the operator in order to compute calculated values.

```
<PP_NEW_ATTRIBUTE attribute_name="born_year"
                  attribute_type="numeric"
                  position="1">
   <TABLE_LOADER xml_source="census.xml"/>
   <EXPRESSION>
      <SEQ_TERM op_type="subtract">
         <BASE_TERM value="2004"/>
         <BASE_TERM value="@age"/>
      </SEQ_TERM>
   </EXPRESSION>
</PP_NEW_ATTRIBUTE>
```

In this example, a new numeric attribute born_year is added in the first position of the census.xml dataset. The values of the new attribute are calculated as the difference between the current year and the year of birth. With the special symbol "@" in the second term of the expression, we denote the input table attribute age.

Sampling is a largely used task: for an input table, a subset of rows is selected accordingly to a sampling method. Therefore, the typing of a sampling operator is $f_{\texttt{PP\_SAMPLING}} : \texttt{table} \times \texttt{algs} \rightarrow \texttt{table}$. Below is an example query selecting 66% of input rows without replacement sampling policy.

```
<PP_SAMPLING xml_dest= "sampling.xml">
   <TABLE_LOADER xml_source= "census.xml"/>
   <ALGORITHM algorithm_name="simple_sampling">
      <PARAM name="percentage" value="0.66"/>
      <PARAM name="with_replacement" value="false"/>
   </ALGORITHM>
</PP_SAMPLING>
```

Another widely used task is discretization of numeric attributes. In the following example, the values of the age attribute are discretized by the natural binning method into three intervals.

```
<PP_NUMERIC_DISCRETIZATION xml_dest= "census_discrete.xml",
                           attribute_name = "age">
   <TABLE_LOADER xml_source= "census.xml"/>
   <ALGORITHM name="natural_binning_discretization">
      <PARAM name="number_of_intervals" value="3"/>
```

```
      <PARAM name="labeling" value="enumeration"/>
      <PARAM name="enumerated_label_list" value="young, adult, elder"/>
   </ALGORITHM>
</PP_NUMERIC_DISCRETIZATION>
```

There are a few other preprocessing operators in KDDML, including: `PP_RENAME_ATTRIBUTES` to rename attributes, `PP_CHANGE_TYPE` to change the logical type of attributes, `PP_REMOVE_ROWS` to delete rows under specified conditions, `PP_REWRITING` to apply pattern matching rewriting of attribute values, `PP_SORTING_ATTRIBUTE` to sort rows according to the values of an attribute or according to their frequencies.

Nevertheless, observe that the list of KDDML preprocessing operators is not comparable to the huge number of preprocessing tasks described in the literature or available in commercial/research systems. However, we point out that adding a new operator in KDDML is not much of a problem, since it is enough to specify the operator signature and the DTD of the operator tags. The overall (functional & compositional) semantics of KDDML allows for a smooth integration and compositionality of the new operator in the language. We refer the reader to Section 5 for a discussion of the effort needed to add a new operator in the system architecture.

4.2 Mining models

Model format

As for data, the KDDML language assumes a *model repository*, containing extracted data mining models, which can be referenced by an identifier (in a different namespace from data).

KDDML represents models as an extension of PMML documents. We have already observed that PMML in its present version is not sufficient to capture all details of mining models. We deploy the PMML extension mechanism, i.e. the `<EXTENSION>` tag, in two cases.

In the first one, the notion of confusion matrix is added to decision tree models. 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. In the following example, a confusion matrix for a class with values `yes` and `no` is reported. Notice that the extension mechanism of PMML requires new tags to have the `X-` prefix in their name.

```
<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>
```

The second extension of PMML concerns again decision trees. We allow for classification models that exploit predictions of two or more decision trees. A classic example concerns meta-classifiers, 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. The example below represents a voting classifier among two decision trees.

```
<Node score="">
   <Extension>
      <X-VotingTree combination_type="committee"/>
   </Extension>
   <Node score="will play">
      ...
   </Node>
   <Node score="no play">
      ...
   </Node>
</Node>
```

It is worth noting that in its early versions, the KDDML language and systems used a proprietary DTD for models. At that time, PMML was not well-established, yet. Also PMML came out as a *interchange* format between data mining applications. On the contrary, we found that it rapidly evolving towards a *representation* language. Its use in the KDDML language and system demonstrates this fact.

Model access

Direct access to models in the *model repository* is achieved by the `TREE_LOADER`, `SEQUENCE_LOADER`, `RDA_LOADER`, `CLUSTER_LOADER`, `HIERARCHY_LOADER` operators. As the name suggests, the forms of knowledge currently addressed include decision trees, sequential patterns, association rules (`RDA`), clustering and item

hierarchies. Also, PMML compliant models provided from external tools can be accessed and imported in the repository.

```
<TREE_LOADER  xml_source="DecisionTree.xml"/>

<PMML_RDA_LOADER  xml_dest="ExternRdA.xml"
                  pmml_source="ftp://www.foo.edu/models/RdA.xml"/>

<HIERARCHY_LOADER xml_source="cities_hierarchy.xml"/>
```

In the first query fragment a decision tree is referenced from the model repository and passed up to the father node of the query fragment. In the second one, a set of association rules is gathered from the internet, copied in the model repository, and passed up.

In the last one, a hierarchy over cities-states-countries is loaded and passed up. Notice that at this stage, hierarchies are not yet related to a table column as meta-data. They can be built from any table, as in the following example.

```
<TABLE_2_HIERARCHY xml_dest="cities_hierarchy.xml">
    <ARFF_LOADER arff_file_path="cities_hierarchy.arff"/>
</TABLE_2_HIERARCHY>
```

Assigning the hierarchy to a table column as meta-data information can be achieved by the preprocessing operator <PP_ADD_HIERARCHY>.

```
<PP_ADD_HIERARCHY attribute_name="city"/>
  <TABLE_LOADER xml_source="customers.xml"/>
  <HIERARCHY_LOADER xml_source="cities_hierarchy.xml"/>
</PP_ADD_HIERARCHY>
```

Model extraction

Mining models are extracted from a data source using a data mining algorithm. In the next example, the top 20 association rules are extracted from market basket data with minimum support of 40% and confidence of 60%.

```
<RDA_MINER xml_dest="MineBasket.xml">
   <ARFF_LOADER arff_file_path="D:/Repository/BasketData.arff"/>
   <ALGORITHM algorithm_name="DCI">
      <PARAM name="min_support" value="0.4"/>
      <PARAM name="min_confidence" value="0.6"/>
      <PARAM name="max_number_of_rules" value="20"/>
   </ALGORITHM>
</RDA_MINER>
```

The <RDA_MINER> tag expects a sub-element with input data and a second sub-element with the algorithm name and parameters (name and value).

Input data can be in the *transactional format*, i.e. with an attribute `transaction` and an attribute `event`. This format allows for deriving intra-attribute association rules, such as `spaghetti AND tomato juice` → `parmesan`.

Also, the *relational format* is recognized, i.e. with an attribute for every item (or item category). This format allows for deriving inter-attribute association rules, such as `carType=racing AND homeInsurance=false` → `married=false`.

The algorithm used here is DCI (Direct Count & Intersect) from [24]. DCI is an efficient procedure that takes into account density or sparsity of input transactions. Again, we point out that other algorithms can be easily integrated in the KDDML language, as shown later on.

Analogously, tags <TREE_MINER>, <SEQUENCE_MINER> and <CLUSTER_MINER> exists for extracting decision trees, sequence patterns and clusters. The algorithms used are respectively the already mentioned YADT [29] for decision tree induction, a main-memory implementation of the PrefixSpan [25] for sequential patterns, and the EM and KMeans clustering algorithms from the Weka library [39]. In the next example, three clusters are extracted from `census.xml` using the EM algorithm.

```
<CLUSTER_MINER xml_dest="MineClusters.xml">
   <TABLE_LOADER xml_source="census.xml"/>
   <ALGORITHM algorithm_name="EM">
      <PARAM name="number_of_clusters" value="3"/>
   </ALGORITHM>
</CLUSTER_MINER>
```

Model application and evaluation

Extracted models can be applied on (new) data to predict features or to select data accordingly to the knowledge stored in the model.

We have seen in Fig. 8 how a decision tree extracted from a training set can be applied to predict the class of tuples in a test set. More in detail, <TREE_CLASSIFY> yields a `table` with an additional column (whose name is the one of the class column followed by *_predicted*) consisting of the class predicted by the decision tree. The procedure used to determine the class predicted is the one adopted in the C4.5 algorithm [28]. The mapping between attributes used in the decision tree and attributes in the test set is by name. By the data preprocessing operators, however, attribute name remapping can be achieved.

In addition to <TREE_CLASSIFY>, operators for model application include:

- <MISCLASSIFIED>: given a table returned by the <TREE_CLASSIFY> operator, selects the rows where the predicted class value differs from the actual class one;
- <RULE_SATISFY> (resp., <RULE_EXCEPTION): given a set of association rules and a table, extracts those transactions in the table that satisfy (resp., are exceptions to [3]) one or more of the association rules; <SEQUENCE_SATISFY> and <SEQUENCE_EXCEPTION> are the equivalent operators for sequence models;
- CLUSTER_NUMBER: given a cluster model and a dataset, this operator returns the tuples of the dataset belonging to a specified cluster number (max can also be specified to get the tuple in the cluster of maximal cardinality);
- CLUSTER_CENTROID: given a cluster model, it returns tuples describing the cluster centroids.

Model (meta-)reasoning

Models extracted by data mining algorithms very often need to be further processed, e.g., combined with other models. As observed in Section 4.2, voting classifiers are included in KDDML as an extension of PMML. The example below returns a voting classifier among three decision trees: tree1.xml and tree2.xml are already present in the model repository, and tree3.xml is mined from trainingSet.xml.

```
<TREE_COMMITTEE xml_dest="treeCommittee.xml">
   <TREE_LOADER xml_source="tree1.xml"/>
   <TREE_LOADER xml_source="tree2.xml"/>
   <TREE_MINER xml_dest="tree3.xml" target_attribute="class_name">
      <TABLE_LOADER xml_source="trainingSet.xml"/>
      <ALGORITHM algorithm_name="YADT">
         <PARAM name="confidence_for_pruning" value="0.4"/>
         <PARAM name="num_instances_for_leaf" value="3"/>
      </ALGORITHM>
   </TREE_MINER>
</TREE_COMMITTEE>
```

The operator performs a run-time checking that the three classifiers share the same meta-data. If this is not the case, the evaluation of the query terminates with a run-time error.

[3] A transaction satisfy an association rule $I_1, \ldots, I_n \rightarrow I_{n+1}, \ldots, I_m$ if every item I_i for $i \in [1, m]$ appears in the transaction. A transaction is an exception to the association rule above if every item I_i for $i \in [1, n]$ appears in the transaction, but some I_i for $i \in [n+1, m]$ does not.

Other operators on model (meta-)reasoning available in the system include:

- `RDA_FILTER (SEQUENCE_FILTER)`: given an association rules model (resp., sequential patterns model), extracts those rules (resp., sequences) satisfying specified conditions. As an example, the query fragment:

```
<RDA_FILTER>
  <RDA_LOADER xml_source="rules.xml"/>
  <CONDITION>
    <AND_COND>
      <BASE_COND op_type="is_in" term1="@body" term2="bread"/>
      <BASE_COND op_type="is_not_in" term1="@head" term2="milk"/>
      <BASE_COND op_type="equal" term1="@head_cardinality" term2="2"/>
      <BASE_COND op_type="greater" term1="@support" term2="0.3"/>
   </AND_COND>
 </CONDITION>
</RDA_FILTER>
```

  selects the tuple:
  · having item `bread` in their body
    (`<BASE_COND op_type="is_in" term1="@body" term2="bread"/>`),
  · and not having item `milk` in their head
    (`<BASE_COND op_type="is_not_in" term1="@head" term2="milk"/>`),
  · and having exactly two items in the head
    (`<BASE_COND op_type="equal" term1="@head_cardinality" term2="2"/>`),
  · and having support greater than 30%
    (`<BASE_COND op_type="greater" term1="@support" term2="0.3"/>`).

  It is worth noting here that the signature of the operator is $f_{<\texttt{RDA\_FILTER}>} : \texttt{rda} \times \texttt{xml} \rightarrow \texttt{rda}$. In other words, the selection condition is specified as a generic XML document, directly interpreted by the operator (and not by the KDDML interpreter).

- `RDA_PRESERVED`: 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 father level yields a rule that belongs to $\mathcal{R}_2$. Here it is a sample query fragment:

```
<RDA_PRESERVED xml_dest="preserved_rules.xml">
  <HIERARCHY_LOADER xml_source="hierarchy.xml"/>
  <RDA_LOADER xml_source="rulesForItems.xml"/>
  <RDA_LOADER xml_source="rulesForBrands.xml"/>
</RDA_PRESERVED>
```

In this section, we describe the operators that allow for better control of flows of data and models in queries, and for calling external programs.

Calls to external programs / RDBMS

Specialized procedures can sometimes be useful to preprocess or analyze data. The <EXT_CALL> operator allows for calling external programs, including e.g., calls to RDBMS stored procedures.

```
<EXT_CALL path="/usr/bin/mysql">
   <PARAM value="localhost">
   <PARAM value="UPDATE mytable SET cost = cost * 1.10"/>
</EXT_CALL>
```

The <PARAM> operator returns a scalar (the one of the value attribute), which is then used as a command line argument of the called program. <EXT_CALL> also returns a scalar, e.g., the number of updated rows in the example above.

Calls of queries

In order to modularize (long) queries, an operator that retrieves and evaluates queries in the query repository is provided. Queries admit parameters, whose list is specified at the <KDD_QUERY> tag. Actual parameters are substituted to formal parameters at the time the query is loaded from the repository.

```
<KDD_QUERY name="generic_tree" par_list="perc,source,dest">
    <TREE_MINER xml_dest="#dest#" target_attribute="class_name">
      <PP_SAMPLING xml_dest= "sampling.xml">
          <TABLE_LOADER xml_source="#source#"/>
          <ALGORITHM algorithm_name="simple_sampling">
              <PARAM name="percentage" value="#perc#"/>
              <PARAM name="with_replacement" value="false"/>
          </ALGORITHM>
      </PP_SAMPLING>
      <ALGORITHM algorithm_name="YADT">
        <PARAM name="num_instances_for_leaf" value="3"/>
      </ALGORITHM>
    </TREE_MINER>
</KDD_QUERY>
```

The above query builds a decision tree on a subset of a data source and saves the tree in a specified destination. Notice that the syntax for using a formal

parameter requires to write it between # signs. The sample query can be called from within other queries as follows:

```
<CALL_QUERY name="generic_tree">
   <PARAM name="perc" value="0.6"/>
   <PARAM name="source" value="training.xml"/>
   <PARAM name="dest" value="tree.xml"/>
</CALL_QUERY>
```

The operator <CALL_QUERY> returns the same 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.

Sequences and parallelism of queries

It is sometimes useful to evaluate queries in strict sequence or to mark potential parallelism. As an example, consider building two distinct models on a same training set. The preprocessing of the training set is preliminary to the (independent, hence potentially parallel) evaluation of the tree building queries.

```
<KDD_QUERY>
  <SEQ_QUERY>
    <EXT_CALL path="mypreprocessing">
      <PARAM value="inputdata.arff"/>
      <PARAM value="training.arff"/>
    </EXT_CALL>
    <PAR_QUERY>
       <TREE_MINER xml_dest="tree1.xml>
          ...
       </TREE_MINER>
       <TREE_MINER xml_dest="tree2.xml>
          ...
       </TREE_MINER>
    <PAR_QUERY>
  </SEQ_QUERY>
</KDD_QUERY>
```

The <SEQ_QUERY> operator (resp., <PAR_QUERY>) models sequentialization (resp., potential parallelism). The returned value of both operators is assumed to be the one of the last operator in the sequence of their arguments [4].

[4] Note that with this assumption, <PAR_QUERY> is functionally equivalent to <SEQ_QUERY>, and then it can be implemented as <SEQ_QUERY> when physical parallelism is not available.

Decision and recursion

Recall the "functional" semantics of KDDML operators. It is then natural to introduce operators for decision and recursion. The latter is achieved by simply allowing a call to a query by inside the query itself. Concerning decision, the following schema:

```
<IF>
  <COND expr=" XQuery expression ">
    ... input to expr
  </COND>
  <THEN>
    ... then branch ...
  </THEN>
  <ELSE>
    ... else branch ...
  </ELSE>
</IF>
```

allows for evaluating an XQuery expression, and, on the basis of the boolean result, to evaluate (in *non-strict* semantics) a *then* branch or an *else* branch.

As an example, the query below builds a decision tree on a subset of a given table. If the number of incorrectly classified cases in the subset is larger than a threshold (100, in the example) then a larger subset is considered.

```
<KDD_QUERY name="sampleGrow" par_list="perc">
 <IF>
    <COND expr="#perc# < 1 and
         input()//x-ConfusionMatrixTest[@x-incorrectlyInst] > 100"/>
      <TREE_MINER xml_dest="tree.xml" target_attribute="class">
         <PP_SAMPLING xml_dest= "sampling.xml">
            <TABLE_LOADER xml_source= "trainingSet.xml"/>
            <ALGORITHM algorithm="simple_sampling">
               <PARAM name="percentage" value="#perc#"/>
               <PARAM name="with_replacement" value="false"/>
            </ALGORITHM>
         </PP_SAMPLING>
         <ALGORITHM algorithm_name="YADT">
            <PARAM name="num_instances_for_leaf" value="3"/>
         </ALGORITHM>
      </TREE_MINER>
    </COND>
    <THEN>
      <CALL_QUERY name="sampleGrow"/>
          <PARAM name="perc" value="#perc#*2">
      </CALL_QUERY>
```

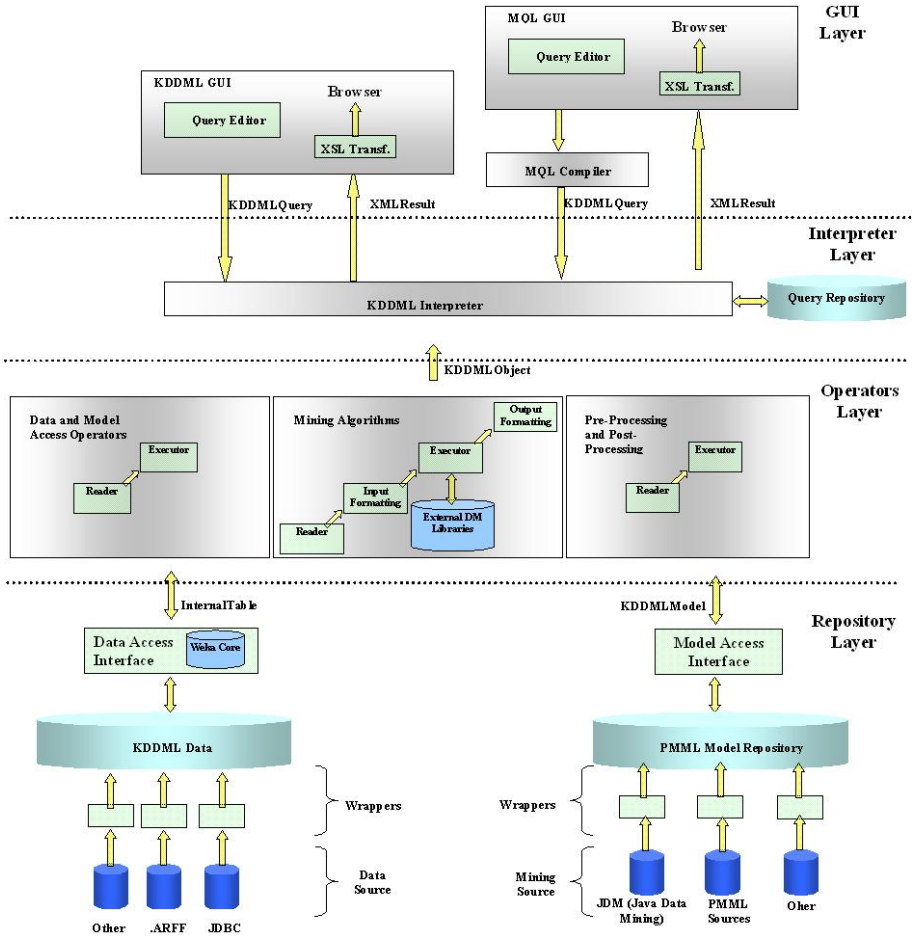


Fig. 10. KDDML system architecture.

```
    </THEN>
    <ELSE>
      <TREE_LOADER xml_source="tree.xml"/>
    </ELSE>
 </IF>
</KDD_QUERY>
```

Notice that `input()` in the XQuery expression refers to the decision tree returned by the `<TREE_MINER>` operator.

5 KDDML: system architecture

The design of the KDDML system had to take into special account the requirements of extensibility of the KDDML language, which can be distinguished into:

- **data sources extensibility:** adding a new data source type in the KDDML language consists of simply adding a new tag (such as `<NEW_DATA_SOURCE_LOADER>`) with appropriate attributes and sub-elements specifying how to locate the table. As a consequence, the system should allow for transparently adding a wrapper to/from the new data source type, which encapsulates the details of transforming data and meta-data into the ones the internal table representation;
- **algorithms extensibility:** adding a new preprocessing, tree induction, clustering, association rules or sequential pattern mining algorithm should be as simple as possible. As for data sources, the idea is that the new algorithm should be *pluggable in* the language and system. Notice that, as far as the language part is concerned, this is not much of a problem, since the algorithm name and parameters are not part of the language syntax;
- **models extensibility:** to some extent, also adding new forms of extracted knowledge should not break the overall design of the system. Adding a new mining model to the language means adding a new type in the operators signatures, which amounts to non-destructive changes in the DTD of the operators.

KDDML is implemented in Java, in order to be portable, with its overall architecture structured in layers as reported in Fig. 10. Each layer implements a specific functionality and supplies an interface to the layer above. In the following, we give an overview of the design of each layer, commenting on how they address the above extensibility requirements.

5.1 Repository layer

The bottom layer manages the read/write access to data and models repositories and the read access to data and models from external sources providing programmatic functionality to the higher layers.

Data and models manager

On the one side, the repository layer provides to higher levels a data/model access interface to manage tables and mining models.

Accessing a table yields a Java object satisfying an interface `InternalTable`, which abstracts sequential and random read/write access to the table rows, and provides metadata (such as column types and preprocessing history) and statistics on table columns. In the earlier versions of the system, `InternalTable` was exactly the class of the Weka systems abstracting (main-memory resident) tables. Next, it has evolved to a more structured set of classes, including access

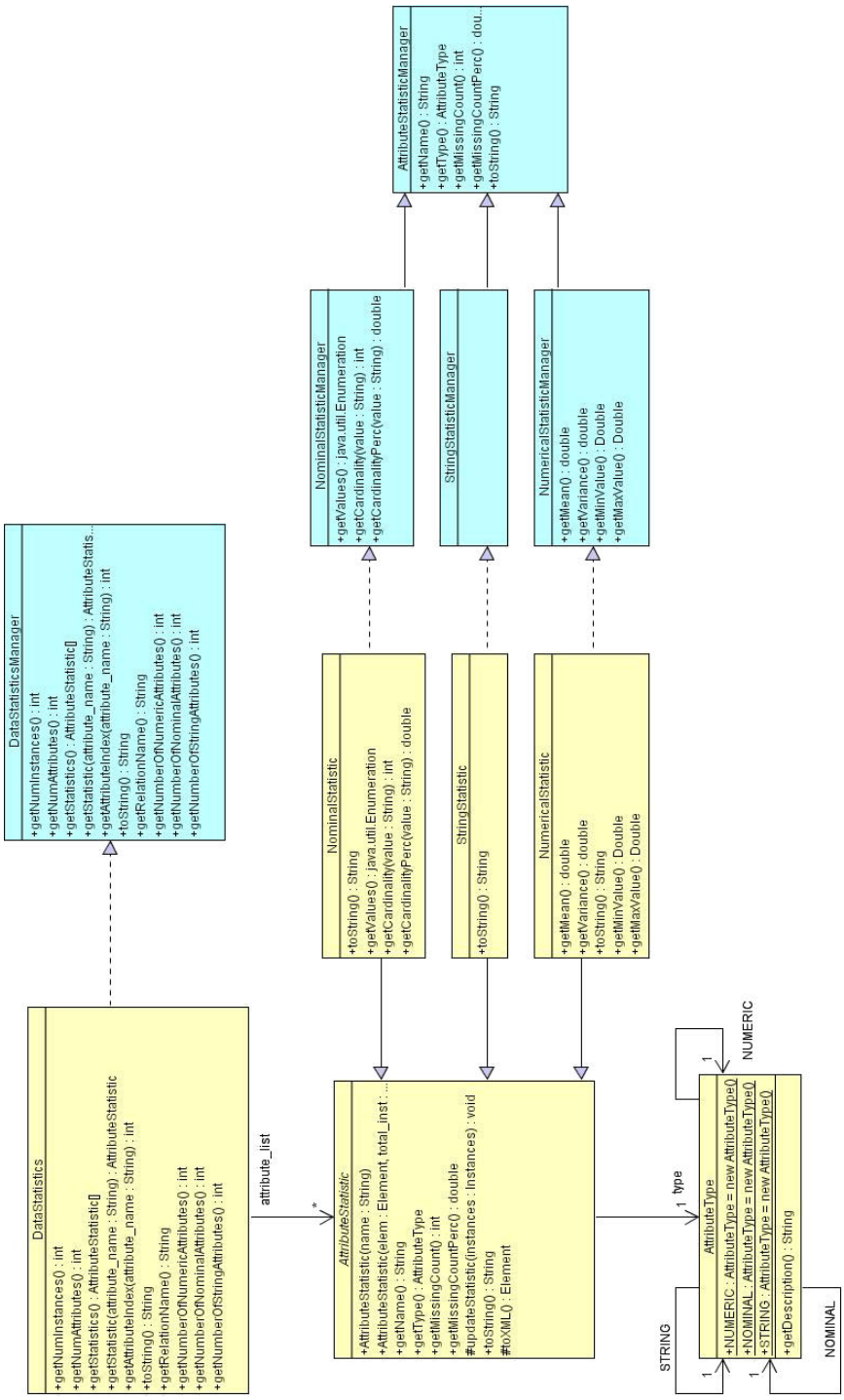


Fig. 11. UML diagram of the attribute statistics calculation module.

to a portion of a table, statistics on attributes, meta-data management. See Fig. 11 for the UML diagram of the statistics data source package.

Accessing a model yields a Java object satisfying an interface `AssociationModel`, `TreeModel`, `ClusteringModel`, `SequenceModel` or `HierarchyModel`. Such interfaces provide programmatic read/write access to the model contents, e.g. `AssociationModel` includes methods for adding and removing rules and itemsets. Fig. 12 shows the UML diagram of the association modelling package. In addition, all the interfaces above extend the abstract class `KDDMLModel`, which provides model metadata (PMML data-dictionary and meta-schema) read/write access.

Data and models factory

On the other side, the repository layer includes wrapper modules for accessing and importing tables and models from external sources and to store, retrieve and delete objects in the repositories. Wrappers take care of translating the format of the data into the internal representation (`InternalTable` or `KDDMLModel`). Automatic conversions are performed by the system on data and metadata. Additional conversions can be forced by the preprocessing operators of the KDDML language.

As far as data is concerned, there are wrappers around ARFF text files, serialized Weka tables and RDBMS tables (accessed via JDBC). Concerning models, there is a wrapper importing from PMML files [5] and it is currently being implemented a wrapper around serialized JDM (Java Data Mining) [17] objects.

Wrappers are organized in a factory class, which provides objects for managing data and models. The "factory" approach allows for separating the access to data/model sources from the system interface of data/model object management.

5.2 Operators and algorithms layer

The upper layer is composed of the implementations of language operators.

Let us consider again the XML syntax of a generic operator.

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

[5] Since the internal representation is an extension of PMML, this amounts to add default values for tags beyond the standard.

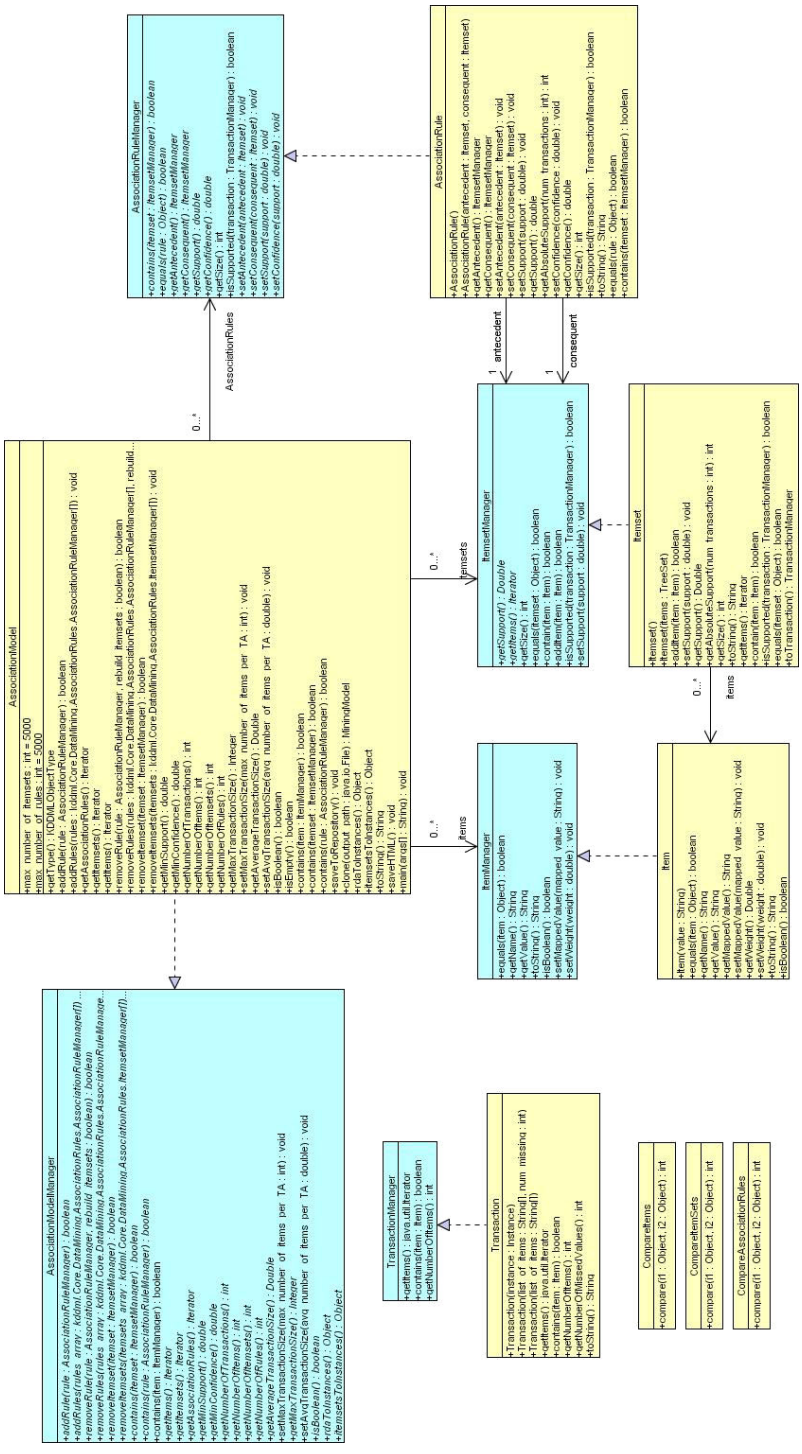


Fig. 12. Association rule manager UML diagram.

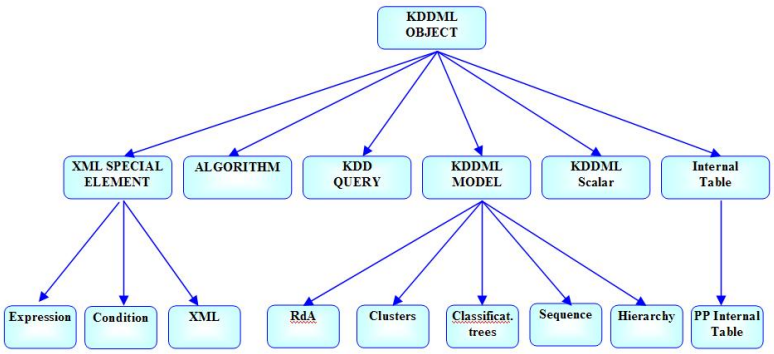


Fig. 13. Object hierarchy in the KDDML system.

```
    ...
    <ARGn_NAME> .... </ARGn_NAME>
</OPERATOR_NAME>
```

In order not to have to distinguish between *data* operators and *model* operators, we consider both `InternalTable` and `KDDMLModel` as subclasses of the abstract class `KDDMLObject`. The overall hierarchy of `KDDMLObject` is reported in Fig. 13. Note that there is a subclass of `KDDMLObject` for each possible result returned by an operator. Also, consider the `xml` data type. We have already observed that its semantics is to model XML tags that are not interpreted directly by KDDML, but simply passed as arguments to the operator being evaluated. However, there are two exceptions to this rule, intended to model recurring XML tags, namely `<EXPRESSION>` and `<CONDITION>`. For those two tags, the interpreter build a Java object able to process respectively expressions and conditions. This object is passed to the operator, not the XML. Hence, the hierarchy of Fig. 13 shows classes `Expression` and `Condition` as inheriting from `XML Special Element`, in addition to `XML` which models the default case.

`<OPERATOR_NAME>` is implemented as a Java class satisfying the `KDDMLOperator` interface, which requires the following methods:

- `boolean runTimeCheckNeeded()`, returns `true` if the type of the result of the `execute()` method is not fixed at compile time, but need to be checked at run-time against the one required by the operator calling `<OPERATOR_NAME>`;
- `KDDMLObjectType paramType(int i)`, returns the expected type of the i^{th} argument of the operator, where `KDDMLObjectType` is an object over an enumeration of all types;
- `boolean checkAttributes(Hashtable atts)`, performs a correctness control about the input attributes of `<OPERATOR_NAME>` given as hash-table (e.g., check that the minimum support of the `<RDA_MINER>` operator is a

real number between 0 and 1). The method returns `true` if no errors are been found.
- `KDDMLObject execute(Vector args)`, returns the result of evaluating the operator over the passed arguments.

The last two methods are called by the higher level (the KDDML interpreter), which first evaluates the sub-elements of the query (`<ARG1_NAME> ... </ARGn_NAME>`), so computing `args`, and the attributes of the XML tag `<OPERATOR_NAME>`, so computing `atts`.

We distinguish three implementation patterns for the `execute()` method, respectively for data/model access, data mining and pre/post processing operators.

Data/model access operators, such as `<ARFF_LOADER>` and `<TREE_LOADER>`, are implemented by direct calls to the repository layer for retrieving and storing data and models.

Mining operators, such as `<TREE_MINER>` and `<RDA_MINER>`, call an appropriate algorithm for extracting or applying a model from/to given training/test data[6]. Typically, the algorithm is an external program (such as YaDT, DCI or Weka algorithms) which requires its own input format and provides its own output format. As a consequence, the implementation of `execute()` normally scans passed data, transforms it into the required input, call the actual algorithm, and finally interprets the output to return an appropriate `KDDMLObject`.

Pre/post processing operators, such as `<PP_SAMPLING>` and `<RDA_FILTER>`, have a similar pattern as mining operators when the operators is implemented by an external algorithm. However, differently from data mining operators, input/output format transformation is time-consuming, since typically one scan over input is sufficient. For that reason, pre/post processing operators are mainly implemented in the KDDML system (i.e., not calling external programs) and they work directly on `InternalTable` and `KDDMLModel` objects.

5.3 Interpreter layer

The interpreter layer accepts a validated KDDML query (either in XML or as a DOM tree), evaluates it, save the final result in the repository and returns it as a `KDDMLObject`. The result can be further processed by standard XML management tools and libraries.

[6] While currently not implemented, models could be extracted incrementally from both data and already existing models. However, this would not change the overall structure of the KDDML language and system.

```
public KDDMLObject resolve_core(Element query, ResultType type)
{
  Vector children = XMLDocument.getChildren(query);

  KDDMLOperator op = KDDMLFactory.getOperator( query.getTagName() );
  Vector params = new Vector();
  for (int i=0; i<children.size(); i++) {
      Element elem = (Element) children.get(i);
      KDDMLObject obj = resolve(elem, op.paramType(i) );
      params.add(obj);
  }

  NamedNodeMap list = query.getAttributes();
  Hashtable attributes = new Hashtable();
  for (int i=0; i<list.getLength(); i++) {
      Node n = list.item(i);
      KDDMLScalar expr = exprEval( n.getNodeValue() );
      attributes.put( n.getNodeName(), expr );
  }
  KDDMLObject result = null;
  if (op.checkAttributes( attributes )) {
      result = op.execute( params );
      if ( op.runTimeCheckNeeded() )
          runTimeChecking( result, type );
  }

  return result;
}
```

Fig. 14. Core KDDML interpreter

The core algorithm of the interpreter is reported in Fig. 14. The interpreter recursively traverse the DOM tree representation of the query, yielding a `KDDMLObject` as a result. Also, the expected type of the result is passed together the query.

At each tag, the strict functional interpretation is applied. A `KDDMLOperator` object is constructed from the XML tag using a factory of objects from the operators layer. Each sub-element is evaluated, returning a vector of `KDDMLObject` [7]. The operator provides the expected type for the sub-element.

[7] Since KDDML queries are `KDDMLObjects`, meta-operators are in principles conceivable, i.e., operators that return KDDML queries. In such a case, an intuitive extension of the interpreter consists of evaluating the returned query, passing its result as the overall result of the operator.

```
public KDDMLObject resolve(Element query, ResultType type)
{
  KDDMLObject result = null;
  Element children = query.getChildren();

  switch( query.getTagName() ) {
    case "IF":
      Element elem = (Element) children.get("COND");
      KDDMLObject obj = resolve(elem);
      NamedNodeMap list = query.getAttributes();
      Node n = list.item("expr");
      if(  xQueryExprEval( n.getNodeValue(), obj ) )
         result = resolve( (Element) children.get("THEN"), type );
      else
         result = resolve( (Element) children.get("ELSE"), type );
      break;

    case "CALL_QUERY":
      Vector params = new Vector();
      for (int i=0; i<children.size(); i++) {
         Element elem = (Element) children.get(i);
         String paramName = elem.getAttribute("name");
         String paramValue = elem.getAttribute("value");
         params.add( new Pair( paramName, paramValue) );
       }
       Element calledQuery = loadQuery( query.getAttribute("name"), params );
       result = resolve( calledQuery, type );
       runTimeChecking( result, type );
       break;

    default:
      result = resolve_core(query, type);
      break;
  }

  String xml_dest = query.getAttribute("xml_dest");
  if ( xml_dest != null)
     saveToRepository( result, xml_dest );

  return result;
}
```

Fig. 15. KDDML interpreter

Also, each attribute is evaluated, returning a vector of `KDDMLScalar`, i.e., numbers or strings. Finally, a parameters control is performed and the operator is executed on the vector above. If the operator claims for dynamic type checking, the returned results is checked against the expected type. Exceptions are raised on critical situations (they are not reported in Fig. 14 for sake of space).

The general interpreter of KDDML is however a little bit more complex. Tags with meta-meaning, such as `<IF>` and `<CALL_QUERY>` must be taken into account. Fig. 15 shows the overall interpreter, which distinguishes three cases.

- The `<IF>` tag has a non-strict semantics. The `<COND>` sub-element is evaluated first, returning a `KDDMLObject`. The attribute `expr` specify an XQuery expression, which is evaluated on (an XML representation of) the returned object, and which is assumed to return a truth value. Based on that, the `<THEN>` or the `<ELSE>` branch are evaluated and their result returned as the overall result.
- The `<CALL_QUERY>` tag has a meta-interpretation. We recall that sub-elements specify actual parameters for the called queries. After collecting pairs of formal/actual parameters, the called query is loaded from the repository and parameters substitution takes place. The `loadQuery` method returns the query that must be evaluated by the interpreter.
- The third case is the KDDML core interpreter of Fig. 14.

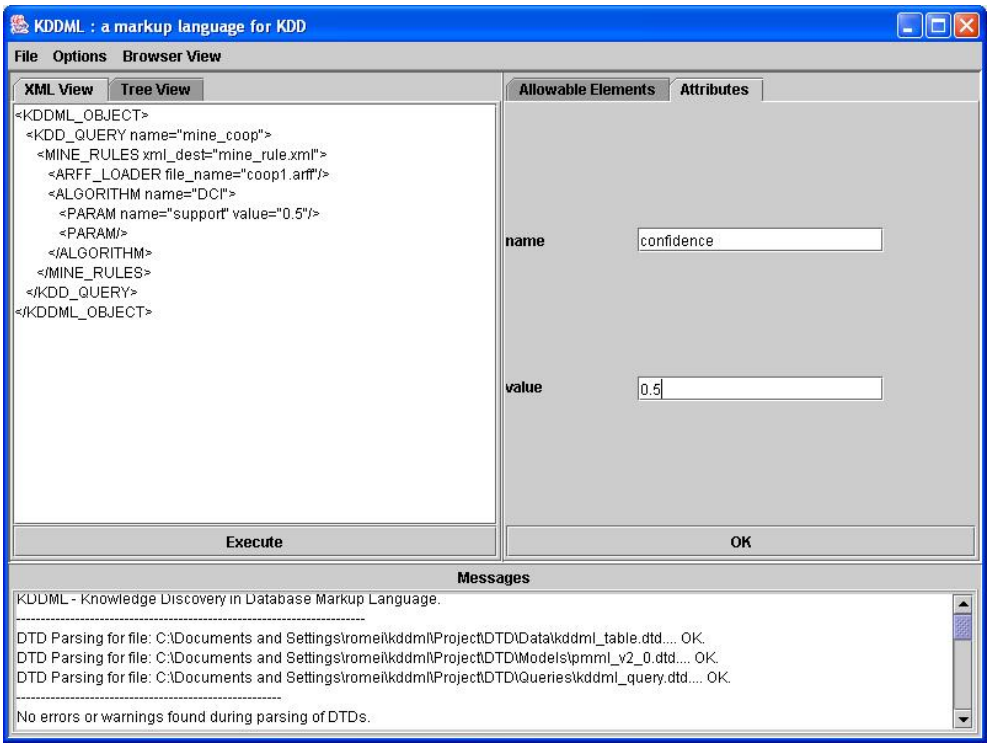
In all cases, if the `xml_dest` attribute is specified, the result is stored in the appropriate (data or model) repository.

As far as the *algorithm extensibility* is concerned, if a new algorithm is added to the system, the only change at the interpreter layer is in the `KDDMLFactory` class, that maps the tag identifier to the operator object. However, this mapping is currently implemented *by name*, i.e., by dynamic loading of a class with name derived from the tag identifier. So, the operator class is actually *plugged-in* at run-time.

Finally, concerning *models extensibility*, it is worth observing that the overall algorithm of the interpreter is not affected by the number and type of mining models. Adding a new mining model implies changes to the repository layer (new interface for programmatic access) and to the operators layer (new operators for extracting, applying and reasoning) as well as the changes to the DTD of the language (for static type checking).

5.4 User interface layer

The KDDML system includes a GUI for user friendly input of queries and for browsing of extracted knowledge. See Fig. 16 for a snapshot of the GUI. The



Frequent Itemsets

Frequent Itemsets (ordered by support)			
Id	Size	List of Items	Support % (absolute support)
1	1	windy=FALSE	0.666 (10 of 15)
2	1	play=yes	0.666 (10 of 15)
3	2	play=yes windy=FALSE	0.466 (7 of 15)

Association Rules

Association Rules (ordered by confidence)				
Antecedent	-->	Consequent	Support % (absolute support)	Confidence
play=yes	-->	windy=FALSE	0.466 (7 of 15)	0.700
windy=FALSE	-->	play=yes	0.466 (7 of 15)	0.700

Fig. 16. The GUI of the KDDML system and a sample of association rules visualization.

GUI allows for:

- opening and modifying an existing query;
- creating a new query through a syntax driven editor, which builds valid queries against the KDDML DTD;
- executing a validated query;
- transforming query results into HTML browsable format via XSL style sheets.

Strictly speaking, the GUI is not part of the core of the system.

In fact, KDDML queries can be generated by other programs, such as a vertical applications that need performing some KDD steps. In other words, programs can link the KDDML interpreter as an external library/driver, invoking the interpreter during their executions. This makes running KDDML queries as simple as running SQL queries over RDBMS. The result of the invoked interpreter is returned as a DOM object or an XML document, which can be further processed with standard tools.

Also, queries can be generated by higher layers of abstractions. As it happens for XML, (long) KDDML queries may result poorly readable and manageable to a human user, distracted by too many syntactic details. As an example of higher abstraction, we have designed and developed an algebraic language (in the style of SQL), called MQL (Mining Query Language) described in [3]. MQL queries are compiled into KDDML queries, which are then executed by the KDDML system. The result is transformed into browsable format (HTML) and presented to the user (see Fig. 16).

Also, visual languages, such as [18,23,30,33,39], are natural higher abstraction layers. The visual metaphora of the KDD process as a graph of tasks (nodes) and flows (arcs) allows for intuitive modelling of the KDD process and for rapid prototyping. We can think of visual languages as friendly interfaces that compile graphs into one or more KDDML queries. In our view, visual GUIs are then somewhat complementary to the KDDML language and system, not competitors.

6 Related work and conclusions

6.1 Related work

Let us briefly compare KDDML with two related system, FlexiMine and Yale, that share several of our design objectives.

FlexiMine

FlexiMine [4] is designed as a test-bed for data mining research.

Similar to the KDDML system, FlexiMine emphasizes integration of several KDD operations, such as preprocessing and visualization, extraction of mining models (decision trees, association rules, bayesian knowledge-bases) and

model filtering. As for KDDML operators, a precise (Java) interface is provided in order to add new algorithms in the system, thus facilitating extension and experimentation. Nevertheless, adding a new algorithm in the FlexiMine systems is not as modular as in KDDML, since both the GUI and the "intermediate layer" are affected. Concerning data representation, FlexiMine is based on an Oracle database as data repository.

Differently from KDDML, FlexiMine does not include a language, but only a GUI for the processing of data, the extraction of models and the filtering of extracted models. The KDD process is not specified as a statement or as a graph. Intermediate tables can be saved as SQL queries. Intermediate models can be saved in text format, which prevents reusing the model in subsequent tasks.

Summarizing, FlexiMine has the same design objectives as KDDML on the system architecture side, since it is mainly intended as a test-bed for testing new algorithms or new implementations. However, KDDML has further objectives on the side of the expressiveness of the language, since it is intended as a middleware for the (rapid) development of vertical applications.

Yale

In Yale [9] (Yet another learning environment), the KDD process is seen as a (possibly nested) chain of sequential tasks.

As an example, consider the chain `[A B[C, D] E]`. Tasks `A`, `B[C, D]` and `E` are performed sequentially. `A` and `E` are basic operators, such as Weka [39] algorithms for machine learning and preprocessing. Task `B[C, D]` is a nested chain consisting of applying the operator `B` (e.g., n-fold cross validation) on tasks `C` (e.g., classification training) and `D` (e.g., classification testing). The output of a task is passed to the next sequential task.

Under this interpretation, the main difference between Yale and KDDML lies in the *procedural* semantics of Yale vs the *functional* semantics of KDDML. In addition, (intermediate) input/outputs in Yale have no specific "data staging" area neither some standard (XML) representation.

As for KDDML, (chains of) tasks are specified in Yale as XML documents. However, operators are untyped, i.e. there is no DTD as in KDDML. As a consequence, all type checks are demanded to the run-time execution of operators.

On the system side, Yale defines interfaces for extending basic operators and chain of operators. Also, access to data tables is clearly specified and includes meta-data information. Adding a new operator is slightly simpler in Yale than

in KDDML, since no DTD has to be modified. However, Yale operators must include additional run-time checks of operands and parameters.

6.2 Experiences with KDDML

The design and development of KDDML over the last three years has benefited from a few experiences in modelling and experimenting complex KDD processes using KDDML. This approach was intended as a form of validation of the language constructs and system architecture. As a result, several weaknesses of the early versions of the system were highlighted, and solutions suggested by the experience were implemented.

In [6], we rewrote in KDDML the steps needed to pre-process web logs, extract a predictive model of future references to web pages, and then deploy the model in an intelligent caching strategy. The steps were taken from a previously developed research project described in [5]. The process was iterative, requiring to determine the best caching strategy at the varying of cache size. This requirements led to adding to KDDML recursion and parametric queries. Also, XPath (a precursor of XQuery) expressions were added.

In the e-commerce project described in [1] the goal was to extract data mining models from visits to a city-news portal with the intent to characterize sex and topics-of-interest of new visitors. The KDD experimentation was only partly conducted using KDDML. But the final system was produced as a Java application calling at several times the KDDML interpreter. This experience highlighted the lack of adequate pre-processing operators, which were subsequently added.

Currently, we are using KDDML for extracting knowledge from annotated literary texts [2], represented as XML documents. This experience is requiring the need for extending the source data type of KDDML to include XML documents. Also, XQuery should be used not only in `<IF>` expressions but as a distinct operator as well.

Summarizing, the validation of the approach proposed in this paper is an ongoing task, interleaving practical experiences of success and challenges with continuous re-design. As shown throughout the paper, however, the overall language definition and the system architecture cover a large set of data, models and algorithms typically needed in KDD applications. More importantly, they are robust enough to allow for smoothly adding new ones.

Knowledge Discovery in Database has reached a maturity state as far as the design of 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. There is a fervent activity of standardization in this area, as reported in Section 2.

In this paper, we have presented the design of a middleware language and system that support the development of higher level applications and systems. Our presentation has started in Section 3 from an analysis of the requirements that such a middleware should satisfy. The KDDML language and system adopt the emerging XML standard as a glue for query definition and data/model representation. XML seems an appropriate choice both for its extensibility and modularity – needed at the language level – as well as for its extensive machine-processability – needed at the system level.

The overall language definition and system architecture allows for satisfying the requirements of Section 3:

- The KDDML data and model repositories fulfill requirements *R1* and *R5* respectively, by providing repositories for data/model staging area.
- The data access operators of the language and the repository layer of the system allows for accessing data and models in diverse representations, automatically performing format conversion and meta-data mapping. This satisfy requirements *R2* and *R3*.
- Also, the model repository is designed to store and access models in a superset of the PMML standard, with appropriate wrapper modules towards pure PMML sources and Java Data Mining (JDM) model sources. This satisfy requirement *R6*.
- The language admits preprocessing operators for specifying/changing logical meta-data of attributes, as required by *R4*.
- The KDDML language is typed and compositional, i.e. an operator of a given type can be used as an argument of another operator which expects that type. KDDML have a clear functional semantics. Type checking is mainly static (by means of XML DTDs), but in some cases dynamic type checking is necessary. Summarizing, the KDDML language satisfies the closure principle of *R7*.
- Concerning compositionality and extensibility of the KDDML system, i.e. requirement *R8*, the overall architecture allows for easy plugging-in of new

data source and of new algorithms for known model type. The system requires non-destructive changes in case new form of mining models had to be added.

KDDML is a prototype system, with some enhancement and optimizations currently being pursued. On-going activities include the development of a parallel version of the interpreter, the addition of other pre and post-processing operators, the design of a compiler of visual descriptions of KDD tasks and flows into KDDML queries, the design of higher level systems built on top of it.

Downloading KDDML.

KDDML is distributed under the GNU GPL licence at the web site:

`http://kdd.di.unipi.it/kddml`

Acknowledgments.

This work has been partially supported by Italian national FIRB project no. RBNE01KNFP *GRID.it* and by Italian national strategic project *legge 449/97* No. 02.00640.ST97.

References

[1] M. Baglioni, U. Ferrara, A. Romei, S. Ruggieri, and F. Turini. Preprocessing and mining web log data for web personalization. In A. Cappelli and F. Turini, editors, *Proceedings of the 8th Congress of the Italian Association for Artificial Intelligence (AI*IA)*, volume 2829 of *Lecture Notes in Computer Science*, pages 237–249. Springer-Verlag, 2003.

[2] M. Baglioni, M. Nanni, and E. Giovannetti. Mining literary texts by using domain ontologies. In *Proc. of the ECML/PKDD 2004 Workshop on Knowledge Discovery and Ontologies*, pages 79–84, 2004. `http://ecmlpkdd.isti.cnr.it`.

[3] M. Baglioni and F. Turini. MQL: An Algebraic Query Language for Knowledge Discovery. In A. Cappelli and F. Turini, editors, *Proceedings of the 8th Congress of the Italian Association for Artificial Intelligence (AI*IA)*, volume 2829 of *Lecture Notes in Computer Science*, pages 225–236. Springer-Verlag, 2003.

[4] R. Ben-Eliyahu-Zohary, C. Domshlak, E. Gudes, N. Liusternik, A. Meisels, T. Rosen, and S. Eyal Shimony. FlexiMine - a flexible platform for kdd research and application development. *Annals of Mathematics and Artificial Intelligence*, 39:175–204, 2003.

[5] F. Bonchi, F. Giannotti, C. Gozzi, G. Manco, M. Nanni, D. Pedreschi, C. Renso, and S. Ruggieri. Web log data warehousing and mining for intelligent web caching. *Data and Knowledge Engineering*, 32:165–189, 2001.

[6] D. Bruno. Extension and experimentation of an xml-based system for kdd, 2001. Master Thesis (in Italian), Dipartimento di Informatica, Univ. di Pisa, Italy.

[7] J. Clear, D. Dunn, B. Harvey, M.L. Heytens, P. Lohman, A. Mehta, M. Melton, L. Rohrberg, A. Savasere, R.M. Wehrmeister, and M. Xu. NonStop SQL/MX Primitives for Knowledge Discovery. In *Proc. of ACM SIGKDD Int. Conference on Knowledge Discovery and Data Mining*, pages 425–429. ACM Press, 1999.

[8] M. Fernandez, J. Siméon, and P. Wadler. An algebra for XML query. In *Proc. of Foundations of Software Tecnology and Theoretical Computer Science (FSTTCS)*, volume 1974 of *LNCS*, pages 11–45. Elsevier Science Inc., 2000.

[9] S. Fischer, R. Klinkenberg, I. Mierswa, and O. Ritthoff. YALE: Yet Another Learning Environment 2.4.1, 2004. University of Dortmund, `http://yale.cs.uni-dortmund.de`.

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

[11] IBM. IBM DB2 Intelligent Miner 8.1, 2004. `http://www-306.ibm.com/software/data/iminer/`.

[12] T. Imielinski and A. Virmani. MSQL: A Query Language for Database Mining. *Data Mining and Knowledge Discovery*, 3(4):373–408, 1999.

[13] International Organization for Standardization (ISO). Information Technology – Database Language SQL, 1999. Standard No. ISO/IEC 9075:1999.

[14] International Organization for Standardization (ISO). Information Technology – Database Language – SQL, 2003. Draft Standard No. ISO/IEC 9075:2003.

[15] International Organization for Standardization (ISO). Information Technology – Database Language – SQL – Part 13: SQL Routines and Types using the Java Programming Language (SQL/JRT), 2003. Draft Standard No. ISO/IEC 9075-13:2003.

[16] International Organization for Standardization (ISO). Information Technology – Database Language – SQL Multimedia and Application Packages – Part 6: Data Mining, 2003. Draft Standard No. ISO/IEC 13249-6:2003.

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

[18] KDNuggets. Data mining software, 2004. `http://www.kdnuggets.com/software`.

[19] R. Meo, G. Psaila, and S. Ceri. An extension to SQL for mining association rules. *Data Mining and Knowledge Discovery*, 2(2):195–224, 1998.

[20] T. Morzy and M. Zakrzewicz. SQL-Like Language for Database Mining. In *Proceedings of the First East-European Symposium on Advances in Databases and Information Systems (ADBIS)*, pages 311–317. Nevsky Dialect, 1997.

[21] A. Netz, S. Chaudhuri, U.M. Fayyad, and J. Bernhardt. Integrating data mining with SQL databases: OLE DB for data mining. In *Proc. of Int. Conference on Data Engineering (ICDE)*, pages 379–387. IEEE Computer Society, 2001.

[22] Object Management Group (OMG). Common Warehouse Meta-model (CWM), 2002. Version 1.1, `http://www.omg.org/cwm`.

[23] Oracle. Oracle 10g Data Mining, 2004. `http://www.oracle.com/technology/products/bi/odm`.

[24] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri. Adaptive and resources-aware mining of frequent sets. In *IEEE Int. Conf. on Data Mining (ICDM)*, pages 338–345. IEEE Computer Society, 2002. `http://hpc.isti.cnr.it/~palmeri/datam/DCI`.

[25] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M.C. Hsu. Mining sequential patterns by pattern-growth: The prefixspan approach. *IEEE Trans. on Knowledge and Data Eng.*, 16(11):1424–1440, 2004.

[26] Prudsys
AG. XELOPES: eXtEnded Library fOr Prudsys Embedded Solutions, Version 1.1, 2004. `http://www.prudsys.com/Produkte/Algorithmen/Xelopes`.

[27] D. Pyle. *Data Preparation for Data Mining*. Morgan Kaufmann Publishers, San Francisco, 1999.

[28] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993.

[29] S. Ruggieri. YaDT: Yet another Decision Tree builder. In *Proc. of the 16th Int. Conf. on Tools with Artificial Intelligence*, pages 260–265. IEEE Computer Society, 2004.

[30] SAS. SAS Enterprise Miner 5.1, 2004. `http://www.sas.com/technologies/analytics/datamining/miner`.

[31] C. Seidman. *Data Mining with Microsoft SQL Server 2000 Technical Reference*. Microsoft Press, 2001.

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

[33] SPSS. Clementine 9.0, 2004. `http://www.spss.com/clementine`.

[34] The Data Mining Group. Predictive Model Markup Language (PMML). Version 3.0, 2004. `http://www.dmg.org`.

[35] W3C World Wide Web Consortium. Namespaces in XML. W3C Recommendation, 1999. `http://www.w3.org/TR/REC-xml-names`.

[36] W3C World Wide Web Consortium. Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation, 2000. http://www.w3.org/TR/REC-xml.

[37] W3C World Wide Web Consortium. XML Schema, Parts 0, 1, and 2. W3C Recommendation, 2001. http://www.w3.org/TR/xmlschema-0, -1, and -2.

[38] W3C World Wide Web Consortium. XQuery: XML Query Language. On-line documentation, 2005. http://www.w3.org/XML/Query.

[39] I.H. 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.