PROGRAMMING THE KDD PROCESS USING XQUERY

Andrea Romei and Franco Turini
Computer Science Department, University of Pisa, Largo B. Pontecorvo 3, 56127 Pisa (PI), Italy

Keywords: Data mining, Query language, Inductive databases, KDD process, Mining constraints, XML, XQuery.

Abstract: XQuake is a language and system for programming data mining processes over native XML databases in the spirit of inductive databases. It extends XQuery to support KDD tasks. This paper focuses on the features required in the definition of the steps of the mining process. The main objective is to show the expressiveness of the language in handling mining operations as an extension of basic XQuery expressions. To this purpose, the paper offers an extended application in the field of analyzing web logs.

1 INTRODUCTION

Since the introduction of XML as a standard for representing semistructured data, the amount of information coded according to such standard is steadily growing. Systems for retrieving information out of such collections of XML data have been developed, up to the point that a number of implementations for handling native XML databases has been proposed[1]. XQuery is probably the most widely accepted language in this area (W3C, 2010). Many authors maintain that the process of data mining can be seen as a sophisticated way of querying the database, and, as a consequence, it is a good idea to extend query languages with features supporting data mining.

According to this point of view the XQuake system has been developed as an extension of XQuery designed to support data mining tasks (Romei and Turini, 2010). Besides being designed for mining native XML databases, XQuake takes advantage of the XML philosophy also for representing the results of the mining process, according to the PMML standard (The Data Mining Group, 2011). The uniformity of the representation of all the levels of information allows the full compliance with the closure principle of inductive databases.

XQuake provides a good basis for mining XML data, but it still offers opportunities for extensions. Two of them are presented in this paper:

- specification of constraints on the mining process;
- the possibility of *programming* different data mining processes in an expressive way.

The first issue is addressed by specifying constructs for binding data to mining models and for knowledge filtering. The second issue is addressed by extending the language with *mining functions*, that exploit typical functional language constructs, including recursion. The paper aims at highlighting the capability of the language of specifying mining tasks in an elegant and expressive way. The basic ideas for the system architecture are coherent with the design of XQuake and can be found in (Romei and Turini, 2010).

Section 2 contains a presentation of XQuake and the proposed extensions by providing its syntax and its semantics, and by exemplifying its use for coding typical mining subtasks. Section 3 offers the description of a concrete application by discussing the implementation of a mining process in detail. The last two sections deal with related work, future work, and some final consideration.

1.1 The `xmark` Database

Through the paper, we adopt an easily accessible source of XML documents, namely `xmark` (Schmidt et al., 2002). It models an Internet auction site, defining entities such as `people`, `open_auctions`, `closed_auctions`, `items` and `categories`. We report below a brief description (and the XML fragment in fig. 1) for the first three entities.

Specifically, the `<people>` tag is made up of a sequence of `<person>` elements encoding profiling information and the history of the visited auctions. The former has a (eventually empty) list of `<interest>`

[1]See http://www.w3.org/XML/Query/ for an exhaustive list of XML XQuery implementations.

Romei A. and Turini F..
PROGRAMMING THE KDD PROCESS USING XQUERY.
DOI: 10.5220/0003626501230131
In *Proceedings of the International Conference on Knowledge Discovery and Information Retrieval* (KDIR-2011), pages 123-131
ISBN: 978-989-8425-79-9

elements indicating the item categories interesting for the user. It also includes other personal information. The `<open_auctions>` element contains auctions in progress. Their properties are the initial price, the bid history along with references to the bidders, a reference to the item being sold and a reference to the seller. Each closed auction (tag `<closed_auction>`) contains the reference to the seller, buyer and item, the price and date of the closed transaction and the type of transaction.

From now on, all the examples will refer the `xmark` data source, that is stored in BaseX (Holupirek et al., 2009), i.e. the native XML database of XQuake. Some highlights on XQuery can be found in (Romei and Turini, 2011b).

```
<people>
 <person id="person27">
  <phone>+39....</phone>
  <profile income="96497.12">
    <interest category="category11"/>...
    <education>High School</education>
    <gender>male</gender>
    <business>Yes</business>
    <age>29</age>
  </profile>
  <watches>
    <watch open_auction="open_auction29"/>...
  </watches>
 </person>...
</people>
...
<open_auctions>
 <open_auction id="open_auction29">
  <initial>26.60</initial>
  <bidder>
    <date>03/28/1998</date>
    <personref person="person17"/>
    <increase>3.00</increase>
  </bidder>...
  <current>220.10</current>
  <itemref item="item255"/>
  <seller person="person1361"/>
 </open_auction>...
</open_auctions>
...
<closed_auctions>
 <closed_auction id="closed_auction9">
  <seller person="person964"/>
  <buyer person="person1650"/>
  <itemref item="item232"/>
  <price>162.44</price>
  <date>01/23/1999</date>
  <quantity>1</quantity>
  <type>Regular</type>...
 </closed_auction>...
</closed_auction>...
```

Figure 1: Three XML fragments of `xmark`.

2 XQuake EXTENDED

The section is organized as follows. We first present the syntax and the meaning of six clauses, that are used as a basis to construct mining operators. Then, such operators are introduced through simple examples. Finally, we show how to specify special mining functions.

2.1 Mining Constructs

Each mining operator is made up of a combination of base constructs. As shown in Figure 2, six operators have been considered as guidelines for the design of XQuake. Specifically, they serve to locate XML data and PMML models, to bind new data to an extracted set of patterns and to specify mining constraints or the format of the output result. After presenting a simple running example, we describe each construct in turn.

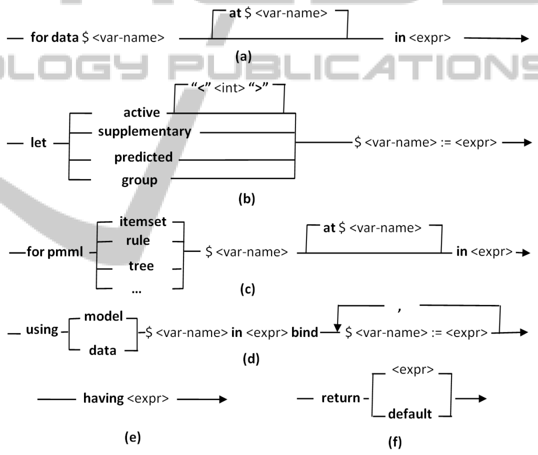

Figure 2: Syntax of the six basic clauses. The `for data` clause (a). The `let` clause (b). The `for PMML` clause (c). The `using` clause (d). The `having` clause (e). The `return` clause (f).

2.1.1 Running Example

To take confidence with the language philosophy, we introduce a "classical" example taken from the inductive database theory. Specifically, we aim at *"mining association rules from a dataset; on such a result, we find all the given instances that satisfy*[2] *the rules; finally, we induce a classification tree from those instances"*. XQuake offers three operators to solve this task.

Below, a set of association rules is extracted to find frequent correlations among the bidders in all

[2]By definition, a transaction satisfies an association rule $I_1 \dots I_n \rightarrow I_{n+1} \dots I_m$ if every item I_i for $i \in [1,m]$ occurs in the transaction.

the open auctions. The output model contains patterns like {Mark} → {John}, which states that when Mark appeared as bidder, also John was a bidder with a certain support and confidence. A condition requires that the size of the extracted rules is equal to 2 (i.e. exactly one item in the body and one item in the head of the rule).

```
for data $auc in doc("xmark")//open_auction
let group $pers := $auc/bidder//@person
having xquake:rule-size() = 2
return default
```

In the above query, the set of involved transactions (i.e. the <open_auction> elements) is specified through the for data clause. Items of each transaction (i.e. the person identifiers for each <bidder> element) are defined in the let group clause. Both the having and return clauses operate on the output result. The former is evaluated for each rule. It uses the rule-size() built-in function, defined in the reserved xquake namespace, to get the size of that rule and to implement the constraint. Notice that the parameter of rule-size() is implicitly an association rule. As soon as constraints are evaluated, the return clause is evaluated once to return a PMML document. Assume that the output rules are stored in my-rules.xml.

In the next XQuake fragment, we filter out a set of instances (i.e. the sequence of <closed_auction> elements) that do not satisfy at least ten association rules in my-rules.xml. To test whether an item (i.e. a person identifier) occurs in a transaction, we check whether that person has been either buyer or seller in the closed auction. Now, the return statement is evaluated for each input data. The result, stored in my-tuples.xml, is a sequence of <inst> tags, each encoding a <closed_auction> if it satisfies the constraint.

```
for data $d in doc("xmark")//closed_auction
using model $r in doc("my-rules")/PMML
bind $pers := $d/[seller|buyer]//@person
return <inst>{if (xquake:rule-satisfy($r) > 9)
              then $d else () }</inst>
```

Finally, a dummy PMML classification tree is induced from my-tuples.xml. It is built on the price and quantity properties of each closed auction to predict the type of the auction (i.e. regular or featured).

```
for data $auc in doc("my-tuples")
let active $price := $auc//price
let active $qty := $auc//quantity
let predicted $type := $auc//type
return default
```

As a general comment, we have supposed in this simple example to store the result of a step to be used as input in the next step. Moreover, we have not yet specified neither the kind of knowledge to extract, nor the mining algorithm to use in the query fragments. In sect. 2.3 we cover these aspects, and a more elegant way to combine mining results is presented.

2.1.2 Constructs for Locating XML Data

The first step in specifying a data mining task is the selection of the relevant data as input of the analysis. Relevant XML elements and attributes are selected by means of the clauses depicted in fig. 2 (a),(b).

The syntax of the for data expression (fig. 2 (a)) is similar to the for clause of XQuery. It sets-up an iteration over the sequence returned by the expression after the in keyword. Each item of the sequence is bound to a variable that can be used in the rest of the expression. The optional at clause allows for a positional variable, which is bound to an integer representing the iteration number.

The let clause (fig. 2 (b)) is used to bind a variable to a mining field. The keyword after the let refers to the role of such an attribute in the mining activity of interest. More specifically, the active keyword specifies that the field is used as input to the mining task: predicted specifies that it is a predicted attribute (e.g. in a classification task), supplementary states that it holds additional descriptive information, and finally, group groups atomic values (e.g. in an association or sequence analysis). Mining fields in input to the mining tasks are required to be atomic (e.g. string, numeric or date), except for a supplementary field that, in principle, can assume any complex XML type. It can be used to hold background knowledge information useful to evaluate constraints. Active fields also admit a special (and optional) syntax to express an atomic sequence of an explicit size in a let active specification. This facility is particularly useful when a large number of XML fields are used in the analysis.

2.1.3 Constructs for Locating PMML Models and Binding New Data

A similar syntax may be used to locate (parts of) a (new or extracted) pattern, represented via PMML. As shown in fig. 2 (c), a variable is bound to each item of the sequence resulting from the evaluation of the expression that follows the in clause. Unlike a for data clause, each item of the sequence is now a single mining model or a set of homogeneous patterns (i.e. either a set of classification tree or frequent itemsets or association rules) sharing the same *mining schema*[3].

[3]The mining schema lists the fields used by the model specifying their usage type, outlier treatment, missing values replacement policy and so on.

The kind of knowledge is specified by means of a special keyword following the `for pmml` expression. Importantly, since the structure of a PMML model is fixed, the user has to specify only the root of the model(s) (i.e. the `<PMML>` element(s)).

Often, new data has to be used in a model context. Consider, for example, the case in which a confusion matrix is constructed from a predictive model in classifying a test set, or, vice-versa, association rules are used to determine which instances violate them. The `using` clause of fig. 2 (d) accomplishes both tasks. A keyword after the `using` distinguishes between "evaluating a model over a dataset" (we say in this case that *the data is bound to the model*) and "evaluating a dataset over a model" (i.e. *the model is bound to the data*). The `using data` and `using model` clauses are used in the first and second case, respectively. In the former, the idea is to set-up an iteration over a sequence to bind each item to a variable. Such a variable can be used in the following `bind` expression. Here, each (non supplementary) field belonging to the mining schema of given mining models is bound to new data, by evaluating the expressions after the assignment symbol. Such binding is by name and type, i.e. each variable of the `bind` clause *must* coincide, in name and type, with a field of the mining schema. The `using model` clause is similar, but it specifies mining models after the `in` keyword and it binds such models to new given data in the `bind` statement.

2.1.4 Constructs for Constraints and Output Specification

We offer a simple and elegant way to express constraints useful to filter an inferred mining model. As shown in fig. 2 (e), a simple XQuery predicate following the keyword `having` is used. We define a library of built-in functions to refer (the main parts of) such output inside the XQuery predicate. This solution has two main advantages. First, it avoids specialized constructs and constraints are expressed more declaratively. Second, the user has to know only the signature and meaning of the external built-in functions to apply constraints. As an example, to filter out uninteresting itemsets, a built-in library offers special functions to get their size, support and other interesting measures, the complete list of the items belonging to the itemset as well as the background knowledge related to these items.

A similar strategy is used to offer to the user the capability of defining its own output, both for data and mining models. The basic idea is to use built-in functions inside an XQuery expression (fig. 2 (f)), that encapsulate the main parts of the result. However, since the output may have a very complex structure (e.g. in the case of mining models), a default output can be specified by means of the `return default` clause, which is a PMML document for mining models.

2.2 Mining Operators

In this section we integrate the running example of sect. 2.1.1 with additional examples of the mining operators, according to the aformentioned specification. Preprocessing, model extraction, filtering and deploying tasks are shown.

2.2.1 Preprocessing

Several preprocessing and data preparation tasks for sorting, selecting and filtering XML data can be directly obtained throughout the use of XQuery constructs. However, since the data preprocessing is a time consuming phase, ad-hoc constructs have to be designed for cleaning, discretization, aggregation, sampling and many others.

The syntax of a preprocessing operator admits a `for data` clause followed by a combination of `let` clauses (whose number and order depend on the kind of task), and by a `return` clause. In the following example, the value of the `<price>` element in each `<closed_auction>` is discretized. The result is encoded in a sequence of `<p>` XML tags.

```
for data $auc in doc("xmark")//closed_auction
let predicted $price := $auc/price
return <p>{xquake:discr($price)}</p>
```

Notice the usage of the built-in function `discr(.)` in the `return` clause, that returns the discretized value of its numeric argument.

2.2.2 Model Extraction

Mining models are directly inferred from XML data. The specification of a model extraction operator includes a `for data` statement, to specify input XML nodes, followed by a combination of `let` clauses, to specify the field (active, predicted or group) as input to the algorithm or the background information. The latter can be used, for example, to specify, by means of an optional `having` clause, domain-based constraints on the output model. A `return` statement closes the statement.

As an example, we can extend the first query of the running example by introducing a more complex constraint to reduce the number of generated rules. Below, the query also specifies that, in each rule, every person in the antecedent bought at least two items in the closed auction history.

```
for data $auc in doc("xmark")//open_auction
let group $pers := $auc/bidder//@person
let supplementary $count-buy := count(
  for $i in doc("xmark")//closed_auction
  where $i/buyer/@person eq $xquake:item
  return $i)
having every $j in xquake:body-context()
       satisfies $j > 1
return default
```

Here, after selecting the transactions (i.e. the auctions) and the items (i.e. the person's identifiers), the `count-buy` variable holds, for each distinct person, the number of items bought by that person. To this purpose, a join of the person identifier (referred by the special variable `$xquake:item`) and the set of `<closed_auction>` elements has been used. For each mined association rule, the built-in function `body-context()` in the `having` clause returns the context information (i.e. a sequence of int values) related to the body items of that rule.

2.2.3 Model Filtering, Application and Evaluation

The extracted knowledge can be filtered according to a condition, that, in principle, can be applied to every model. The general syntax begins with the clause `for pmml`, in which one has to specify the kind of model, followed by a `having` and `return` clause. Similar operators are used to apply an extracted model on new data, to predict features, to select data accordingly to the knowledge stored in the model, or to evaluate the model itself. In these cases, a `using` clause is used to bind data to the knowledge.

As an example, consider the third query of the running example and let suppose that we have induced a set of trees, stored in `my-trees.xml`. Below, their mining schema is shown:

```
<MiningSchema>
 <MiningField name="price" usage="active"/>
 <MiningField name="qty" usage="active"/>
 <MiningField name="type" usage="predicted"/>
</MiningSchema>...
```

Here, the PMML `<MiningSchema>` lists the fields (i.e. name and usage) which a user has to provide in order to apply the model. The first query below filters out those trees having a training confidence lower than 50% for `type = "regular"` in the root node, where the path expression in the `let` clause returns the PMML `<ScoreDistribution>` element of the root node (see (The Data Mining Group, 2011)).

```
for pmml tree $t in doc("my-trees")/PMML
having let $d := $t//Node/ScoreDistribution
                         [@value eq "regular"]
       return $d/@confidence > 0.5
return $t
```

Given new XML data compliant with the mining schema above, the next two queries return the set of PMML confusion matrixes (one for each tree) constructed on such data and the predicted values of the target field, respectively.

```
for pmml tree $t in doc("my-trees")/PMML
using data $d in doc("xmark")//closed_auction
bind $price := $d/price,
     $qty := $d/quantity
     $type := $d/type
return <tree>{xquake:conf-matrix()}</tree>
```

```
for data $d in doc("xmark")//closed_auction
using model $t in doc("my-trees")/PMML
bind $price := $d/price,
     $qty := $d/quantity
     $type := $d/type
return <classes>{xquake:class($t)}</classes>
```

Observe that the two queries above have a similar syntax, but different semantics. The first one evaluates the expression in the `return` clause for each input tree. At each iteration, it sets-up a cycle over the data sequence to construct the confusion matrix and to compute the evaluation metrics. The second one returns a `<classes>` element for each item of the input data sequence. Each `<classes>` tag encapsulates the predicted values, so that its size coincides with the number of input trees. Given a set of association patterns (resp. rules), similar operators can be used to get the contingency tables of each itemset (resp. rule), or to predict the instances that violate/satisfy those itemsets (resp. rules).

2.3 Putting it all Togheter

At this point, one should note that in the simple queries above, we haven't yet defined neither the kind of knowledge mined, nor the mining algorithm used, nor, and more importantly, how to deploy a mining operator inside a KDD process. From this latter perspective, two important aspects have to be modelled: *iteration* and *interaction*. The KDD is an interactive, iterative and multi-step process in the sense that, at any stage, the user should have the possibility to choose different algorithms/parameters, to evaluate a condition that selects a "then" branch or an "else" branch, or to iteratively repeat some step to achieve better results. Also, a language supporting a KDD process should include constructs encouraging the reuse of (parts of) the process previously defined to easily integrate this sub-query (i.e., sub-process) inside a more complex one.

To make the KDD process modular and reusable, XQuake adds to XQuery the capability of defining special *mining user-defined functions* whose body is made up of a mining operator. Below, an example of mining function declaration is shown:

```
declare mining function
    my-nmspace:my-fun($my-param as xs:int) {
 < mining operator >
};
```

As for standard functions and variables, user-defined mining functions can be called either from almost any place in a query or in an external mining module. For example, they can be invoked inside a `FLWOR`, conditional, switch or quantified expression, as well as in mining functions themselves. The syntax of a mining function call is the same of any other function, except for the first argument that is an algorithm specification with relative parameters. For example, to call the function above by using the `apriori` algorithm with a minimum support and confidence of 10%, one mights use:

```
my-nmspace:my-fun(rules:apriori(0.10,0.10), 1)
```

The `rules` namespace indicates the kind of knowledge to be mined, in this case association rules. Currently, XQuake supports discretization, `discr`, sampling, `sampl`, the generation of frequent itemsets, `itemsets`, rules `rules`, classification trees `trees` and their filtering, evaluation and usage.

3 APPLICATION SCENARIO

This section reports a concrete usages of XQuake. The goal is to present a simple (but also taken from our real-experience in data mining) KDD process to show how XQuake is particularly suitable for supporting an inductive database framework. An additional example of KDD process based on frequent itemsets can be found in (Romei and Turini, 2011b).

In `xmark`, about 50% of the users provide a profile with personal information (see the `<profile>` tag in the first XML fragment of fig. 1). Also, they specify their categories of interest (e.g. music or sport auctions) during the registration process. Among the registered users, only a subset provides personal information on the age (about 45%). The idea is to use the other personal information to predict that missing information. To this purpose, we aim at extracting a classification tree able to discriminate age based both on the other personal information and on the specified interests. The knowledge of the missing information of the registered user will allow to offer, at time of accessing, personalized banners, promotions or news. The overall process is schematized in fig. 3.

[Discretization]. In order to use a classification algorithm, the `age` information is discretized into three

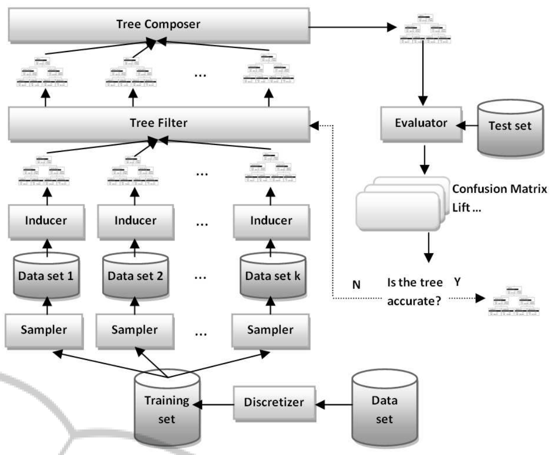

Figure 3: A sample KDD process based on classification.

distinct intervals, `young`, `mid` and `old`. The overall data is then partitioned into two samples for training and testing. At this stage, we do not use a sampling algorithm, but rather the users are selected among those having provided a phone number (about 50%) and the others. The use of the phone information offers a quite randomized partition.

[Bagging Classification]. Accuracy can be increased via a *bagging classification*. More specifically, a classifier is trained on a sample of instances taken *with a replacement* strategy from the training set. This task is repeated k times and, at each iteration, the sample size is equal to the size of the original training set. The output is a set of k inducers: $T = \{t_1, \ldots, t_k\}$. Then, T is filtered according to a condition. Specifically, for each t_i, $i \in [1,k]$, these conditions must hold: (i) the overall number of nodes of t_i is below a certain threshold, α, and (ii) the accuracy of each leaf that classifies as `young` in t_i is greater than a parameter, β. The result of this phase is a new set of inducers, $T' = \{t_1, \ldots, t_h\}$, with $h \leq k$. Notice that the first condition above tries to reduce the complexity of the trees avoiding those subject to *overfitting*. The second one permits to consider only those trees that are more precise in predicting young users. The survived classifiers are composed to generate a bagged classifier, $t_{T'}$, that returns the class that has been predicted most often by means of a voting method among $t_i \in T'$.

[Evaluation]. Once the composed tree, $t_{T'}$, has been constructed, it can be applied to a test set to evaluate its performance in terms of an accuracy error. If the resulting accuracy is greater than a given threshold, γ, then $t_{T'}$ is returned. Otherwise, the filtering task is repeated on T' by using a more stringent value of the α parameter, to the aim of filtering out additional trees with an high number of nodes. The survived trees $T'' = \{t_1, \ldots, t_j\}$, with $j \leq h$ are composed and the procedure is repeated until the condition on the accuracy is fulfilled or $T'' = \emptyset$.

The KDD process just described can be implemented in XQuake as reported in fig. 4. In the `registered-users` module a set of mining and XQuery functions is defined.

The `discretizer()` function discretizes the `<age>` XML element for each person having specified a value for the age. In the result, we append, to each `<profile>` element, a `<age-discr>` tag containing that discretized value. The `sampler(.)` function gets as input a sequence of `<person>` elements and it uses the built-in function `num-times-in-sample($i)` to get the number of times the current item (i.e. person) of the sequence belongs to the sample of index `$i` (we recall that we have a single sample with index 1 and a replacement strategy is used). Below, a fragment of the output is shown:

```
<person>
  <profile income="96497.12">
    <interest category="category11"/>...
    ...
    <age>29</age>
  </profile>
  <age-discr>young</age-discr>
</person>...
```

The `inducer(.)` function extracts a classification tree given a sequence of `<person>` elements (i.e. the training set). Active fields of the task are the sub-element `<business>`, `<education>` as well as the top-five interests specified by each user. For the sake of brevity, we suppose a user-defined XQuery function `select-interests($p, $n)` (not shown in fig. 4) is defined. Given an XML `<person>` element, `$p`, and the number of required categories, `$n`, that function returns a sequence of boolean values of size `$n`. Each boolean value indicates whether the person `$p` has an interest on the i^{th} category, with $i \in [1, n]$.

The filtering module is implemented in the `filter(.)` function. It yields a sequence of PMML trees and the α and β parameters. It also uses a built-in function, namely `xquake:tree-leaves($t)`, in the `having` clause, to get the list of leaves as PMML elements.

Finally, the composition, classification and evaluation are performed by means of the function `bagging(.)`. It takes a set of PMML trees and a test set as a sequence of (discretized) `<person>` elements. Then, it sets-up an iteration in which, for each `<person>`, the predicted values of the `<age>` element are collected for each classification tree. This is achieved via the `classifier(.)` mining function that returns a sequence of predicted classes (containing one value for each tree) for each item of the input sequence. Such single predictions are used to predict the target attribute, according to a majority strategy (XQuery function `majority-class(.)` not shown in fig. 4). A sequence of misclassified values is returned, as shown in the following XML fragment:

```
<mis>mid young</mis>
<mis>old young</mis>
<mis>old mid</mis>...
```

The overall process is assessed throughout an XQuery program. It yields as parameters the number of iterations, k, and the α, β and γ thresholds. Also, it uses the recursive XQuery function, `tester(.)`, to filter, compose and evaluate the induced trees until the condition on the accuracy is respected or no more trees survive to the filter. This is an elegant way to simulate iterations depending on a condition.

4 RELATED WORK

The explotation of XML as a flexible and extensible instrument for IDBs has been studied in (Euler et al., 2006; Romei et al., 2006; Meo and Psaila, 2006).

RapidMiner (Euler et al., 2006) is an environment for KDD and machine learning in which experiments are described via XML documents. While the graphical user interface supports interactive design, the underlying XML representation enables automated applications after the prototyping phase.

KDDML (Romei et al., 2006) and XDM (Meo and Psaila, 2006) are the most related works. In both, the KDD process is modeled as an XML document and the description of an operator application is encoded by means of an XML element. They integrate XQuery expressions into the mining process. For instance, XDM encodes XPath expressions into XML attributes to select sources for the mining, whilst KDDML uses an XQuery expression to evaluate a condition. XQuake does not use XML for the process representation, but rather it directly extends XQuery to achieve a better expressiveness in representing the KDD process.

Mining XML data are used in an instrumental way in (Baralis et al., 2007), to construct summarized representations of XML data. The authors propose to extract association rules from XML databases as the basis for a pattern based representation of XML datasets. The idea is to use the patterns to answer queries on the datasets.

Finally, we mention (Blockeel et al., 2008) as far as the definition of a relational-based inductive database.

```
mining module namespace reg = "registered-users";

declare mining function reg:discretizer() {
  for data $pers in doc("xmark")/site/people/person[not(empty(profile/age))]
  let predicted $age := $pers/profile/age
  return <pers> {
            ($pers, <age-discr>{xquake:discr($age)}</age-discr>)
         } </pers>
};

declare mining function reg:sampler($dataset as node()*) {
  for data $person in $dataset
  return (for $i in (1 to xquake:num-times-in-sample(1)) return $person)
};

declare mining function reg:inducer($training-set as node()*) {
  for data $person in $training-set
  let active $education := $person//profile/education
  let active $is-business := $person//profile/business
  let active<5> $interests := reg:select-interests($person, 5)
  let supplementary $age = $person//age-discr
  return default
};

declare mining function reg:filter($trees as node()*, $alpha, $beta) {
  for pmml tree $t in $trees
  having (count($t//Node) <= $alpha) and
         (every $i in xquake:tree-leaves()
          satisfies $i/@value eq "young" and $i/@confidence > $beta)
  return $t
};

declare function reg:bagging($trees, $test-set) {
  for $pers in $test-set
  let $pred := reg:majority-class(reg:classifier(trees:apply(), $trees, $person))
  return if ($pred != $pers/age-discr)
         then <mis>{($pers/age-discr,$pred)}</mis> else ()
};

declare mining function reg:classifier($trees, $test-set) {
  for data $person in $test-set using model $t in $trees/PMML
  bind $education := $person//profile/education,
       $is-business := $person//profile/business,
       $interests := reg:select-interests($person, 5),
       $age = $person//age-discr
  return default
};

declare function reg:tester($trees, $test-set, $alpha, $beta, $gamma) {
  let $trees := reg:filter(trees:filter(), $trees, $alpha, $beta)
  return if ((count(reg:bagging($trees, $test-set)) <= $gamma) or (empty($trees)))
         then $trees else reg:tester($trees, $test-set, $alpha - 5, $beta, $gamma)
};
```

Figure 4: The `registered-user` mining module implementing the KDD process of fig. 3.

For a recent and complete review on inductive databases see (Romei and Turini, 2011a).

5 CONCLUSIONS

XQuake is a new implementation of an inductive database system over XML data. The scenario pre-

Table 1: Summarization of the XQuake language.

Inductive Database requirement	XQuake perspective
Data and model storage	Native XML Database (models represented via PMML)
KDD process representation	XQuery program + special mining functions
KDD process parametrization	Parametrization of XQuery functions
Closure principle	Achieved by means of the XQuery closure
Constraints & interesting measures	XQuery expression + built-in function library
Output specification	XQuery expression (optional) + built-in function library
Data binding	Based on the PMML mining schema

sented in this paper offers an idea of its potentialities and advantages. First, XML data is mined where it is, in a native XML database. Second, great attention has been paid to the closure principle: the examples highlight the ability of combining the results of the knowledge extraction in order to evaluate certain indicators, to compose preprocessing, data mining and post-processing, and to use background knowledge to filter models. Finally, the KDD process has now an integrated view and it can be easily made modular and parametric. Tab. 1 summarizes the main features of XQuake, according to the inductive database principles.

Since our project aims at a completely general solution for XML data mining, there are further extensions that need an in-depth investigation. An on going work is the integration of both further knowledge (specifically, sequential patterns) and a rich library of mining algorithms. Also, we are working on providing the formal semantics of XQuake. Future work can go in two (often orthogonal) directions: (i) the exploitation of ontologies to represent metadata (on the expressiveness side), and (ii) the study of query rewriting techniques for optimization purposes (on the architectural side). The study of more sophisticated high-level guis for the design of the queries is another aspect to be considered in the future.

REFERENCES

Baralis, E., Garza, P., Quintarelli, E., and Tanca, L. (2007). Answering XML queries by means of data summaries. *ACM Trans Info Syst*, 25(3):1–10.

Blockeel, H., Calders, T., Fromont, E., Goethals, B., Prado, A., and Robardet, C. (2008). An inductive database prototype based on virtual mining views. In *KDD*, pages 1061–1064, New York, NY, USA. ACM.

Euler, T., Klinkenberg, R., Mierswa, I., Scholz, M., and Wurst, M. (2006). YALE: rapid prototyping for complex data mining tasks. In *KDD '06*, pages 935–940, Philadelphia, PA, USA.

Holupirek, A., Grün, C., and Scholl, M. (2009). BaseX and DeepFS - Joint Storage for Filesystem and Database. In *EDBT*, pages 1108–1111, Saint Petersburg, Russia. ACM.

Meo, R. and Psaila, G. (2006). An XML-based database for knowledge discovery. In *EDBT '06*, pages 814–828, Munich, Germany.

Romei, A., Ruggieri, S., and Turini, F. (2006). KDDML: a middleware language and system for knowledge discovery in databases. *Data Knowl. Eng.*, 57(2):179–220.

Romei, A. and Turini, F. (2010). XML data mining. *Softw., Pract. Exper.*, 40(2):101–130.

Romei, A. and Turini, F. (2011a). Inductive database languages: requirements and examples. *Knowl. Inf. Syst.*, 26(3):351–384.

Romei, A. and Turini, F. (2011b). Programming the KDD process using XQuery. Technical Report (extended version) TR-11-10, University of Pisa, Department of Computer Science.

Schmidt, A., Waas, F., Kersten, M., Carey, M. J., Manolescu, I., and Busse, R. (2002). XMark: a benchmark for XML data management. In *VLDB*, pages 974–985.

The Data Mining Group (2011). The Predictive Model Markup Language (PMML). Version 4.0.1. www.dmg.org/pmml-v4-0-1.html.

W3C (2010). XQuery 3.0: An XML Query Language. W3C Working Draft 14 December 2010. www.w3.org/TR/xquery-30/.