

UNIVERSITÀ DEGLI STUDI DI PISA
FACOLTÀ DI SCIENZE MATEMATICHE FISICHE E NATURALI
CORSO DI LAUREA IN INFORMATICA

RELAZIONE SUL TIROCINIO

Estensione del linguaggio e del sistema KDDML con operatori per pattern sequenziali

Daniele Cerra

Tutore: Dott. Salvatore Ruggieri

Anno Accademico 2004-2005

Indice

1	Introduzione	1
2	KDDML	4
2.1	Il linguaggio KDDML	4
2.2	Architettura	9
2.2.1	Repository Layer	10
2.2.2	Algorithms e Operators Layer	10
2.2.3	Interpreter Layer	12
2.2.4	User Interface Layer	12
3	Pattern Sequenziali	13
3.1	Definizioni	13
3.2	Operatori	15
4	Estensione del linguaggio: Specifica degli operatori	17
4.1	Model access	18
4.1.1	PMML_SEQUENCE_LOADER	18
4.1.2	SEQUENCE_LOADER	18
4.2	Model extraction	19
4.2.1	SEQUENCE_MINER	19
4.3	Model Application	20
4.3.1	SEQUENCE_SATISFY	20
4.3.2	SEQUENCE_EXCEPTION	20
4.4	Model (meta-)reasoning	21
4.4.1	SEQUENCE_RULE	21
4.4.2	SEQUENCE_FILTER	21
4.4.3	SEQUENCE_MAXIMAL_FILTER	24
4.4.4	SEQUENCE_AGGREGATE_FILTER	24
4.4.5	SEQUENCE_TIMESTAMP_FILTER	27
5	Estensione del sistema: Livello Core	30
5.1	Input: formato e supporto	30
5.1.1	Formato dei dati	30
5.1.2	Estensione del supporto	32
5.2	Modello: rappresentazione ed integrazione	35

5.2.1	Integrazione del nuovo modello	35
5.2.2	Guida all'estensione del modello	42
5.3	Interfacciamento con il sistema	43
6	Estensione del sistema: Livello Operatori	45
6.1	Operatori a livello di sistema	45
6.2	SEQUENCE_MINER	48
6.2.1	PREFIX_SPAN	48
6.3	SEQUENCE_SATISFY	52
6.4	SEQUENCE_EXCEPTION	53
6.5	SEQUENCE_MAXIMAL_FILTER	53
6.6	SEQUENCE_RULE	54
6.7	SEQUENCE_FILTER	55
6.8	SEQUENCE_AGGREGATE_FILTER	55
6.9	SEQUENCE_TIMESTAMP_FILTER	56
7	Conclusioni	58

Capitolo 1

Introduzione

L'utilizzo di sistemi informatici per la raccolta ed organizzazione di dati è un aspetto fondamentale di ogni azienda, come la capacità di analizzarli al fine di trarne nuove informazioni. Le grandi dimensioni dei dati a disposizione ed il loro costante aumento hanno reso necessaria l'automatizzazione del processo di scoperta/valutazione di nuove informazioni, dando vita ad un nuovo settore di ricerca nel campo informatico, il *Knowledge Discovery in Database (KDD)*.

Il KDD è il processo non banale di identificazione di modelli validi, nuovi, potenzialmente utili e comprensibili sui dati. Il processo KDD è *interattivo* in quanto necessita di un intervento umano su alcune decisioni, *iterativo* perchè richiede l'applicazione di una o più fasi precedenti se i risultati ottenuti in una determinata fase non sono soddisfacenti.

I passi principali del processo KDD sono:

1. *Consolidamento dei dati.*

Collezionamento dei dati da sorgenti eterogenee per la creazione del *Data Warehouse*, un database separato utilizzato come sorgente dati nel processo KDD.

2. *Selezione e preprocessing.*

Preparazione dei dati per la fase di data mining.

3. *Data Mining.*

Fase principale del processo, che si occupa della scoperta automatica di pattern e dello sviluppo di modelli predittivi.

4. *Interpretazione e valutazione.*

I pattern ed i modelli estratti, vengono interpretati per ottenere della conoscenza utile.

In questi ultimi anni, presso il dipartimento di informatica dell'Università di Pisa, è stato sviluppato un sistema ed un linguaggio, il *KDDML* (KDD Markup Language), per supportare l'intero processo KDD.

Scopo di questo tirocinio è estendere il sistema per supportare il processo di estrazione e valutazione di pattern sequenziali, un modello di data mining.

Allo stato iniziale, il sistema non prevedeva alcun tipo di supporto per i pattern sequenziali. Il punto di partenza è stato il *PrefixSpan* [10], un algoritmo di estrazione di pattern sequenziali precedentemente implementato [5].

L'obiettivo iniziale consisteva nell'integrazione dell'algoritmo nel sistema, e nello sviluppo di alcuni operatori di valutazione per i pattern estratti.

La prima parte del tirocinio è stata dedicata allo studio del problema di *estrazione di pattern sequenziali*, introdotto da Agrawal e Srikant [2], e di alcuni problemi correlati, come la *scoperta di episodi frequenti in sequenze di eventi*, introdotto da Mannila e al. [8], e la *formulazione universale* del problema di estrazione di pattern sequenziali, che generalizza i problemi introdotti, rispettivamente, da Agrawal e Srikant e da Mannila e al. [6]. Lo studio (non approfondito) di problemi correlati a quello dell'estrazione di pattern sequenziali è stato necessario al fine di avere una panoramica più completa del problema da trattare.

Il passo successivo è stato lo studio dell'algoritmo *PrefixSpan* e della sua implementazione, per comprenderne le funzionalità e capire come, successivamente, si sarebbe potuto integrare nel sistema.

Una volta contestualizzato il problema, si è proceduto con lo studio del sistema e dei linguaggi/tecnologie utilizzati/e, in modo da poter avere le basi per procedere all'estensione. Per quanto riguarda il sistema sono stati svolti studi introduttivi sull'architettura di massima e sui vari moduli, e sui linguaggi utilizzati (XML, PMML, XSLT).

L'*eXtensible Markup Language* (XML) [15] è uno standard per la rappresentazione di dati semi-strutturati, ampiamente utilizzato per lo scambio di dati tra applicazioni, in quanto il formato di un documento XML è indipendente dall'applicazione che ne fa uso. Tale formato è specificato tramite un *document type definition* (DTD), associato al documento, che ne definisce la grammatica.

Il *Predictive Model Markup Language* (PMML) [13] è uno standard, basato su XML, per lo scambio di modelli di data mining, e consiste di un insieme di DTD per la rappresentazione dei modelli estratti.

L'*eXtensible Stylesheet Language Transformations* (XSLT) [14] è un linguaggio per trasformare documenti XML in altri documenti XML. Nel contesto del tirocinio, l'attenzione è stata rivolta all'utilizzo di XSLT come linguaggio per trasformare i modelli estratti, rappresentati come estensione dei modelli PMML, in documenti HTML.

Una volta acquisite tutte le conoscenze basilari, si è passati all'estensione vera e propria del sistema.

Il primo passo è stato estendere le funzionalità base del sistema per poter rappresentare i pattern sequenziali e fornire i supporti necessari al processo di estrazione: estensione dei formati di input e dei modelli estratti.

Successivamente è stato integrato l'algoritmo nel sistema, operazione che ha richiesto l'adattamento dei formati di input/output dell'algoritmo a quelli richiesti dal sistema.

L'ultimo passo è consistito nella definizione della specifica e nell'estensione degli operatori di sistema. Fino a questo punto, gli operatori da implementare non erano stati ben delineati. Tenendo conto del lavoro già svolto e della vasta gamma di operatori presenti in letteratura, sono stati concordati quelli da implementare.

Nel capitolo 2 viene presentato il KDDML dal punto di vista architetturale e del linguaggio. Nel capitolo 3 viene introdotto formalmente il problema dell'estrazione di pattern sequenziali. Il capitolo 4 riguarda l'estensione del linguaggio KDDML, e quindi la specifica di tutti gli operatori introdotti. Nel capitolo 5 e 6 viene presentata l'estensione del sistema con , rispettivamente, funzionalità base ed operatori. Il capitolo 7 conclude la relazione.

Capitolo 2

KDDML

KDDML (KDD Markup Language) è un sistema ed un linguaggio middleware progettato per supportare l'intero processo KDD.

In questo capitolo verranno presentati gli aspetti principali del KDDML: il linguaggio e l'architettura del sistema.

2.1 Il linguaggio KDDML

Il linguaggio KDDML si basa completamente su XML per la rappresentazione di dati, modelli e query, ottenendo, così, un alto livello di astrazione.

Le query sono documenti XML, dove i tag XML corrispondono alle operazioni da effettuare su dati e/o modelli, gli attributi XML corrispondono ai parametri di queste operazioni ed i sotto-elementi XML corrispondono agli argomenti da passare agli operatori. La figura 2.1 mostra un esempio di query KDDML, che estrarre un insieme di pattern sequenziali e calcola le corrispondenti regole.

```
<KDD_QUERY name="kddml_query">
  <SEQUENCE_RULE min_confidence="0.3" xml_dest="results.xml">
    <SEQUENCE_MINER xml_dest="patterns.xml">
      <TABLE_LOADER xml_source="coop_timestamp.xml" />
      <ALGORITHM algorithm_name="prefix_span">
        <PARAM name="min_support" value="0.6">
      </ALGORITHM>
    </SEQUENCE_MINER>
  </SEQUENCE_RULE>
</KDD_QUERY>
```

Figura 2.1: esempio di query KDDML

Pe ogni operatore è specificata una *signature*, che definisce il tipo restituito e la sequenza degli argomenti dell'operatore.

La signature di un operatore $f : t_1 \times \dots \times t_n \rightarrow t$ che ritorna un tipo t , è denotata

definendo un DTD per l'operatore (query) KDDML, che vincola gli argomenti (figli) ad essere del tipo t_1, \dots, t_n .

Il seguente frammento XML:

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

corrisponde ad un generico operatore.

La valutazione del frammento XML, consiste:

1. nella valutazione ricorsiva dei frammenti
`<ARG1_NAME> ... </ARG1_NAME>`, ... `<ARGn_NAME> ... </ARGn_NAME>` ;
2. valutazione degli attributi `att1 ... attM`;
3. invocazione dell'operatore $f_{\text{OPERATOR_NAME}}$, con argomenti i risultati prodotti da (1) e (2).

Se è specificato l'attributo `xml_dest`, il risultato finale viene memorizzato nel repository.

Il linguaggio KDDML soddisfa un *principio di chiusura*, vale a dire che qualunque operatore che ritorna il tipo t può essere usato ogni volta che un argomento di tipo t sia richiesto. Per soddisfare il principio di chiusura, viene definita una *entità* per ogni tipo restituito dagli operatori; ogni entità raggruppa tutti gli operatori che restituiscono lo stesso tipo. La convalida delle query, come documenti XML, con il DTD, corrisponde ad un controllo statico del tipo degli operatori nelle query.

Esempio 2.1 (Operatore di data mining).

```
<SEQUENCE_MINER>
  <ARFF_LOADER arff_file_path="repository/data/ARFF/"
               arff_file_name="coop_timestamp.arff"/>
  <ALGORITHM algorithm_name="prefix_span">
    <PARAM name="min_support" value="0.9"/>
  </ALGORITHM>
</SEQUENCE_MINER>
```

Di seguito viene riportato l'insieme dei tipi degli operatori del linguaggio KDDML.

Relational table (table)

Questa entità rappresenta una tabella relazionale. Una tabella KDDML è composta da:

- uno *schema*, che contiene i tipi degli attributi ed alcune statistiche sui valori;
- i *dati* sotto forma di file di testo in formato *csv* (Comma Separated Value);

I dati fisici possono essere rappresentati in diversi formati: *relazionale*, *transazionale*, e *timestamp*.

- *Formato relazionale* Ogni colonna di dati corrisponde ad un attributo logico, ed ogni riga ad un record (transazione).
- *Formato transazionale* I dati hanno un numero variabile di attributi logici (items) tra tutti quelli possibili; per ottenere una rappresentazione più compatta, vengono rappresentati solo gli items presenti. Una tabella transazionale ha un attributo **transaction** che identifica la transazione e un attributo **event** che contiene il singolo item. Le transazioni sono ordinate rispetto all'attributo **transaction**. La tabella può contenere altri attributi, ma, a seconda del contesto, possono essere ignorati dall'operatore.
- *Formato timestamp* Simile alla tabella transazionale, ma con un attributo **timestamp**. Quest'attributo definisce un ordinamento parziale fra le transazioni e gli item.

Preprocessing table (PPtable)

Rappresenta le tabelle utilizzate nella fase di preprocessing. Una tabella di preprocessing è composta da:

- uno *schema*, che contiene i tipi degli attributi ed alcune statistiche sui valori;
- i *dati* sotto forma di file di testo in formato *csv* (Comma Separated Value);
- i *dati di preprocessing* che includono le informazioni di preprocessing.

PMML models (rda, cluster, tree, sequence, hierarchy)

Questa entità è definita per rappresentare i modelli estratti: *regole d'associazione*, *clusters*, *alberi di classificazione*, *pattern sequenziali* e *gerarchie di items*.

I modelli KDDML sono rappresentati come estensione dei modelli PMML.

Ogni modello è composto da:

- un *data dictionary*, che contiene le meta-informazioni sui dati utilizzati per estrarre il modello;
- un *mining schema*, indica quali dati sono stati utilizzati per estrarre il modello, ed in che modo;
- una *descrizione del modello*, che varia da modello a modello.

Scalar (scalar)

Uno scalare contenente una costante numerica o una stringa.

```

<?xml version="1.0" encoding="UTF-8"?>
<KDDML_OBJECT>
  <KDDML_TABLE data_file="market_timestamp.csv">
    <SCHEMA logical_name="market" number_of_attributes="5"
      number_of_instances="38">
      <ATTRIBUTE name="transaction" number_of_missed_values="0"
        type="string">
        <STRING_DESCRIPTION/>
      </ATTRIBUTE>
      <ATTRIBUTE name="timestamp" number_of_missed_values="0"
        type="numeric">
        <NUMERIC_DESCRIPTION mean="3.079" variance="3.53"
          min="1.0" max="7.0"/>
      </ATTRIBUTE>
      <ATTRIBUTE name="event" number_of_missed_values="0"
        type="string">
        <STRING_DESCRIPTION/>
      </ATTRIBUTE>
      <ATTRIBUTE name="price" number_of_missed_values="0"
        type="numeric">
        <NUMERIC_DESCRIPTION mean="7.34" variance="10.99"
          min="2.0" max="15.0"/>
      </ATTRIBUTE>
      <ATTRIBUTE name="quantity" number_of_missed_values="0"
        type="numeric">
        <NUMERIC_DESCRIPTION mean="2.32" variance="2.65"
          min="1.0" max="6.0"/>
      </ATTRIBUTE>
    </SCHEMA>
  </KDDML_TABLE>
</KDDML_OBJECT>

```

Figura 2.2: Esempio di rappresentazione dei dati

Algorithm (alg)

Questa entità è utilizzata per specificare un algoritmo di data mining o preprocessing; l'attributo `algorithm_name` identifica l'algoritmo, mentre l'elemento `PARAM` permette di specificare una lista di parametri (`name`, `value`) da passare all'algoritmo.

Condition (cond)

Questa entità è definita per rappresentare la specifica di una condizione. Una condizione può essere utilizzata per valutare operatori booleani (come \geq) su attributi di una tabella (relazionale o di preprocessing) o su proprietà del modello (per es. il supporto di un pattern sequenziale). L'elemento `CONDITION` (Fig. 2.5) è una combinazione booleana (`AND`, `OR`, `NOT`) di casi base (elemento `BASE_COND`). Ogni condizione base è espressa utilizzando l'attributo `op_type`, che rappresenta l'operatore booleano. I parametri dell'operatore vengono passati utilizzando gli attributi XML dell'elemento `BASE_COND` (`term1`, `term2` e `term3`).

```

<?xml version="1.0" encoding="UTF-8"?>
<PMML version="2.0">
  <Header copyright="Copyright (c) 2004-2005 - University of Pisa,
    Department of Computer Science - All Rights Reserved.">
    <Application name="KDDML (Knowledge Discovery in Databases Markup Language)"
      version="2.0"/>
  </Header>
  <DataDictionary numberOfFields="3">
    <DataField name="transaction" optype="ordinal"/>
    <DataField name="timestamp" optype="ordinal"/>
    <DataField name="event" optype="ordinal"/>
  </DataDictionary>
  <SequenceModel modelName="visits_sequence_rules" functionName="sequences"
    algorithmName="PrefixSpan - Computer Science Departement, Pisa, Italy"
    numberOfTransactions="10000" maxNumberOfItemsPerTransaction="1179"
    avgNumberOfItemsPerTransaction="2.4244" minimumSupport="0.02"
    minimumConfidence="0.1" numberOfItems="15" numberOfSets="15"
    numberOfSequences="50" numberOfRules="33">
    <MiningSchema>
      <MiningField name="transaction" usageType="active"/>
      <MiningField name="timestamp" usageType="active"/>
      <MiningField name="event" usageType="predicted"/>
    </MiningSchema>
    <Item id="1" value="business"/>
    <Item id="2" value="frontpage"/>
    ...
    <Item id="15" value="weather"/>
    <Itemset id="1" numberOfItems="1">
      <ItemRef itemRef="1"/>
    </Itemset>
    ...
    <Sequence id="19" numberOfSets="3" occurrence="300" support="0.03">
      <SetReference setId="2"/>
      <Delimiter delimiter="acrossTimeWindows" gap="unknown"/>
      <SetReference setId="9"/>
      <Delimiter delimiter="acrossTimeWindows" gap="unknown"/>
      <SetReference setId="2"/>
    </Sequence>
    ...
    <SequenceRule id="1" numberOfSets="3" occurrence="200" support="0.02"
      confidence="0.667">
      <AntecedentSequence>
        <SequenceReference seqId="20"/>
      </AntecedentSequence>
      <Delimiter delimiter="acrossTimeWindows" gap="unknown"/>
      <ConsequentSequence>
        <SequenceReference seqId="1"/>
      </ConsequentSequence>
    </SequenceRule>
    ...
  </SequenceModel>
</PMML>

```

Figura 2.3: visits_sequence_rules.xml: un modello per pattern sequenziali

```

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

```

Figura 2.4: L'elemento ALGORITHM.

```

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

```

Figura 2.5: L'elemento CONDITION.

Expression (expr)

Questa entità è utilizzata per rappresentare delle espressioni nel linguaggio.

```

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

```

Figura 2.6: L'elemento EXPRESSION.

2.2 Architettura

L'architettura del KDDML è strutturata a *strati*, come riportato in figura 2.7. Ogni strato implementa delle funzionalità del sistema, e fornisce un'interfaccia a gli strati superiori.

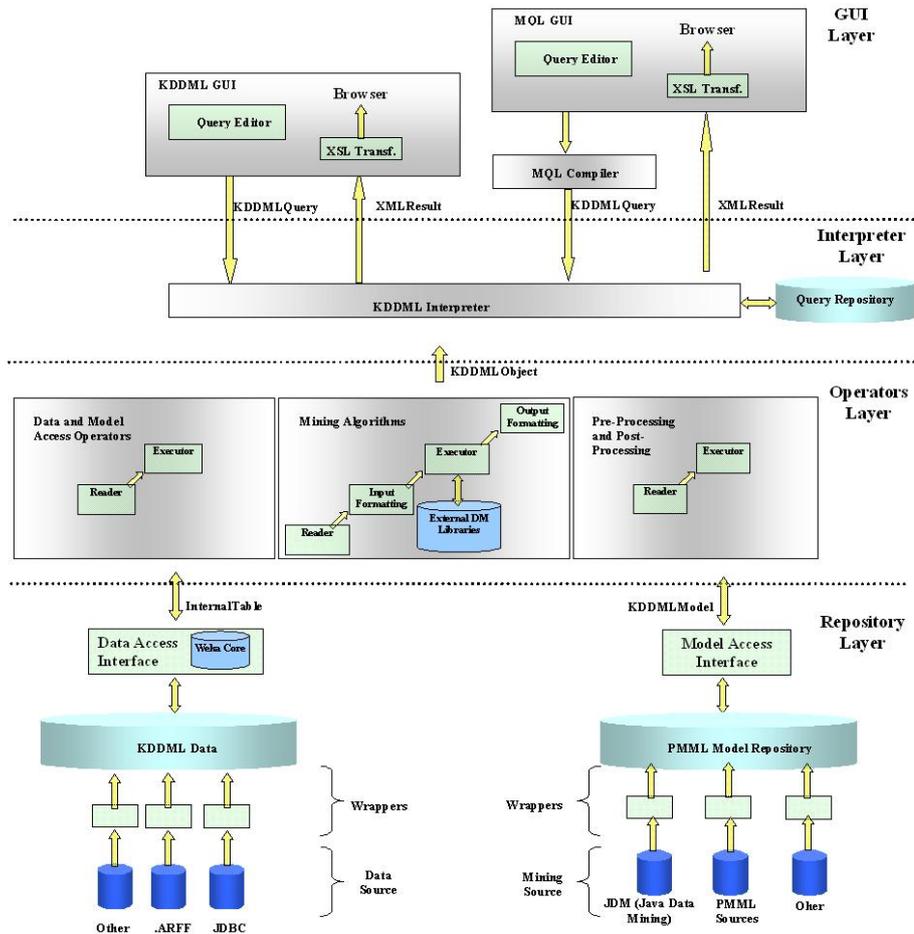


Figura 2.7: Architettura del sistema KDDML

2.2.1 Repository Layer

Il livello più in basso è il *Repository*, che costituisce il nucleo del sistema. Infatti, oltre a fornire un'interfaccia di gestione (*Manager*) per dati e modelli, fornisce dei moduli (*factory*) per accedere a dati e modelli. Esistono inoltre, dei moduli factory per l'accesso a dati e modelli in formato non proprietario, che si occupano di trasformare dati e modelli importati, nel formato richiesto dal sistema.

In pratica questo livello racchiude tutte le funzionalità necessarie ai livelli superiori per la gestione di dati e modelli:

- accesso, creazione e gestione;
- importazione di formati non proprietari.

2.2.2 Algorithms e Operators Layer

Questo livello è composto dalle implementazioni degli operatori del linguaggio.

Un operatore `<operator_name>` è implementato come una classe Java che soddisfa l'interfaccia `KDDMLOperator`, che richiede i seguenti metodi:

- `boolean runtimeCheckNeeded()`
restituisce `true` se il tipo del risultato del metodo `execute()` non è determinato a tempo di compilazione, ma deve essere controllato a run-time;
- `boolean abortIfIsEmpty()`
dice all'interprete di interrompere l'esecuzione della query se il risultato prodotto è vuoto;
- `KDDMLObjectType getParamType(int i)`
restituisce il tipo dell'*i*-esimo argomento dell'operatore;
- `boolean checkAttributes(Hashtable atts)`
esegue un controllo di correttezza sui parametri (attributi) dell'operatore.
- `KDDMLObject execute(Vector arguments)`
restituisce il risultato dell'esecuzione dell'operatore sugli argomenti dati.

La figura 2.8 mostra la gerarchia degli oggetti nel sistema KDDML. Come si può notare, esiste un oggetto per ogni tipo di operatore, ossia per ogni possibile risultato restituito da un operatore. Inoltre, sia dati che modelli, estendono la classe astratta `KDDMLObject`, di conseguenza, non esistono distinzioni tra operatori su dati, ed operatori su modelli.

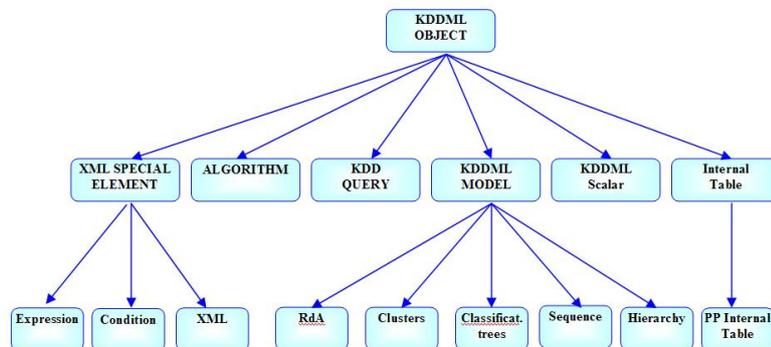


Figura 2.8: Gerarchia degli oggetti nel sistema KDDML

Esistono tre diversi pattern di implementazione per il metodo `execute()`, rispettivamente per operatori di accesso a dati/modelli, data mining e pre/post processing.

operatori di accesso a dati/modelli utilizzano direttamente le funzionalità del *Repository* per caricare/salvare dati e modelli.

operatori di data mining utilizzano un algoritmo per estrarre/applicare un modello da/a i dati in input. Solitamente, l'algoritmo è un programma esterno, che richiede/fornisce in input/output il proprio formato. Quindi, normalmente, il metodo `execute()` trasforma l'input nel formato richiesto dall'algoritmo, richiama l'algoritmo e trasforma l'output nell'appropriato `KDDMLObject`.

operatori di pre/post processing hanno un pattern simile a quello degli operatori di data mining quando l'operatore è implementato da un algoritmo esterno. Differentemente dagli operatori di data mining, gli operatori di pre/post processing sono implementati principalmente nel sistema KDDML, per evitare le trasformazioni del formato dell'input/output.

2.2.3 Interpreter Layer

Il livello *Interpreter* accetta query KDDML già convalidate, le valuta, salva il risultato nel repository e restituisce il corrispondente `KDDMLObject`.

L'aggiunta di nuovi modelli o algoritmi, non comporta la modifica dell'interprete, ma solamente dei livelli sottostanti (*Repository* e *Algorithms and Operators*). Tutte le operazioni svolte dall'interprete, fanno uso delle interfacce definite da `KDDMLObject` e `KDDMLOperator`.

2.2.4 User Interface Layer

L'ultimo livello, fornisce un'interfaccia utente per:

- aprire e modificare una query;
- creare una nuova query tramite un editor che ne controlla la sintassi;
- eseguire una query convalidata;
- trasformare il risultato di una query in HTML tramite XSL per poterla visualizzare in un browser.

È importante notare che questo livello, non è parte integrante del sistema. Infatti, essendo concepito come sistema middleware, altre applicazioni, possono invocare direttamente le funzionalità dell'interprete.

Capitolo 3

Pattern Sequenziali

3.1 Definizioni

In [2] viene introdotto il problema di estrarre pattern sequenziali da insiemi di dati di grosse dimensioni, dove un record è composto principalmente da un *identificatore* della transazione, dalla *data* in cui è avvenuta la transazione e dagli *items* ad essa appartenenti.

L'insieme di dati è visto come un insieme di *sequenze di dati*, dove una sequenza di dati è l'insieme completo di transazioni con lo stesso identificatore.

L'estrazione di pattern sequenziali, punta a scoprire le relazioni esistenti tra occorrenze di eventi sequenziali ovvero, la sequenzialità di *azioni* comuni tra le diverse sequenze di dati.

Esempio 3.1. *Analizzando le transazioni di acquisto dei clienti di una libreria, si possono scoprire quali sono i pattern sequenziali di acquisto frequenti; per esempio, l'80% dei clienti acquista "Il silenzio degli innocenti" ed "Hannibal", e successivamente il "Drago rosso".*

Definizioni Sia $I = \{i_1, i_2, \dots, i_m\}$ un insieme di letterali, chiamati *items*. Un *itemset* è un insieme non vuoto di items.

Una *sequenza* è una lista ordinata di itemset. Una sequenza s viene rappresentata come $\langle s_1, s_2, \dots, s_n \rangle$, dove s_j è un itemset; s_j è anche chiamato un *elemento* della sequenza, e viene rappresentato come (x_1, x_2, \dots, x_m) , dove x_j è un item.

Un elemento di una sequenza è un insieme distinto di items, $\forall (x_1, x_2, \dots, x_m)$, $x_j \neq x_i, \forall i = 1, \dots, m \wedge i \neq j$. Un itemset è considerato come una sequenza con un singolo elemento. Senza perdita di generalità si può assumere che gli items in un elemento di una sequenza siano ordinati lessicograficamente.

La *lunghezza* di una sequenza è data dal numero di elementi che la compongono. Una sequenza di lunghezza k è detta *k-sequenza*.

Una sequenza $a = \langle a_1, a_2, \dots, a_n \rangle$ è una *sottosequenza* di una sequenza $b = \langle b_1, b_2, \dots, b_m \rangle$, $a \sqsubseteq b$, se esistono $i_1 < i_2 < \dots < i_n$ tali che

$a_1 \subseteq b_{i_1}, a_2 \subseteq b_{i_2}, \dots, a_n \subseteq b_{i_n}$. Ad esempio, la sequenza $\langle(3)(5, 6)(7)\rangle$ è contenuta nella sequenza $\langle(2, 3, 4)(5, 6, 8)(6, 9, 11)(7, 9)(10)\rangle$, poichè $(3) \subseteq (2, 3, 4)$, $(5, 6) \subseteq (5, 6, 8)$ e $(7) \subseteq (7, 9)$, mentre $\langle(2, 5)\rangle$ non è contenuta in $\langle(2)(5)\rangle$ (e viceversa).

La sequenza $\langle(3)(5)\rangle$ indica che gli items 3 e 5 appartengono a due transazioni distinte t_1, t_2 , t.c. $3 \in t_1$ e $5 \in t_2$, e $time(t_1) < time(t_2)$ (*inter-transaction items*), mentre la sequenza $\langle(3, 5)\rangle$ indica che gli items 3 e 5 appartengono alla stessa transazione (*intra-transaction items*).

Dato un insieme di sequenze S , una sequenza $s \in S$ è *massimale*, se s non è contenuta in nessun'altra sequenza di S .

Input Sia dato un database di sequenze SDB . Con il termine *data-sequence* si denota una sequenza $d = \langle d_1, \dots, d_m \rangle \in SDB$, dove d_i rappresenta gli items contenuti nella i -esima transazione di d ; cioè una lista di transazioni, ordinate in modo crescente rispetto alla data.

Una transazione è composta dai seguenti campi: *sequence-id*, identificatore della sequenza, *transaction-id*, identificatore della transazione, *transaction-time*, data della transazione, e gli items presenti nella transazione.

Per semplicità, si assume che nessuna coppia di transazioni sia stata effettuata nella stessa data, ed una transazione può essere identificata dal campo *transaction-time*, omettendo il campo *transaction-id*.

Gli items rappresentano delle variabili binarie, i.e. indicano quali items sono contenuti in una transazione e non quanti.

Sequence-Id	Transaction-Time	Transaction Items
Seq001	10	a, b, c
Seq001	15	b, d, e, f
Seq001	20	d, f
Seq002	15	e, f
Seq002	30	a, b, c
Seq002	50	a, c, f
Seq003	20	e, f
Seq003	25	a, c, d, f
Seq004	25	a, b, f

Tabella 3.1: Esempio di database di sequenze.

Supporto Una data-sequence d *supporta* una sequenza s se e solo se s è una sottosequenza di d , $s \subseteq d$.

Dato una database di sequenze SDB , il *numero di occorrenze* di una sequenza s è definito come il totale delle data-sequence $d \in SDB$ che *supportano* s :

$$occurrence(s) = |d \in SDB, s \subseteq d| ,$$

il *supporto* di una sequenza s è definito come la percentuale di data-sequence che supportano s :

$$support(s) = \frac{occurrence(s)}{|SDB|} .$$

Con riferimento alla tabella 3.1, la sequenza $s = \langle (a, c)(f) \rangle$ occorre 2 volte, rispettivamente in *Seq001* e *Seq002*, quindi $occurrence(s) = 2$. Il numero totale di data-sequence nel database è 4, quindi $support(s) = 0.5$ (50%).

Problema Dato un database di sequenze SDB , il problema di estrarre pattern sequenziali consiste nel trovare tutte le sequenze $s \in SDB$ il cui supporto è maggiore del *supporto-minimo* (specificato dall'utente). Ognuna di queste sequenze rappresenta un *pattern sequenziale*, chiamato anche *sequenza frequente*.

Regole Sequenziali Data una sequenza frequente $s = \langle s_1, s_2, \dots, s_n \rangle$, è spesso utile conoscere il supporto delle relazioni tra gli elementi della sequenza. Ossia, quale percentuale delle data-sequence che supporta $\langle s_1 \dots s_i \rangle$ supporta l'intera sequenza s . Essendo $\langle s_1 \dots s_i \rangle$ a sua volta una sequenza frequente, il supporto di questa relazione può essere facilmente calcolato.

Sempre con riferimento alla tabella 3.1, abbiamo detto che la sequenza $s = \langle (a, c)(f) \rangle$ occorre 2 volte ed ha supporto 0.5. La sequenza $s' = \langle (a, c) \rangle$, invece, occorre 3 volte, quindi ha supporto 0.75. Il supporto tra s' ed s è dato da $support(s)/support(s')$, pari a 0.67, che sta ad indicare che il 67% delle volte che in una transazione occorrono gli items (a, c) , in una transazione successiva occorre anche l'item (f) .

In generale, dato un insieme iniziale di "azioni" intraprese da un utente, questo tipo di informazione può essere utile per predire quelle future.

3.2 Operatori

Il processo di estrazione, può restituire un numero enorme di pattern sequenziali, di cui molti possono risultare non interessanti.

In [11] viene presentato un nuovo problema, l'*estrazione vincolata* di pattern sequenziali. L'obiettivo è quello di inserire più vincoli possibili nel processo di estrazione, ottenendo due importanti risultati:

1. ridurre il numero di pattern estratti durante il processo di estrazione, aumentando l'efficienza;
2. estrarre solo i pattern che soddisfano un particolare insieme di condizioni, concentrando l'estrazione solo sui pattern di effettivo interesse.

Anche se la trattazione è focalizzata sull'inserimento dei vincoli all'interno del processo di estrazione, fornisce un punto di partenza per capire quali operazioni possano essere applicate ai pattern una volta estratti.

Si possono distinguere tre tipologie di operazioni, in base alle proprietà su cui sono espressi i vincoli:

1. *proprietà del pattern* i vincoli sono espressi esclusivamente su caratteristiche del pattern, quali:

- supporto,
 - lunghezza,
 - numero di items,
 - items.
2. *proprietà degli items* i vincoli sono espressi sull'aggregazione (delle proprietà) degli items che compongono un pattern (ad es. la somma dei prezzi degli items nel pattern).
 3. *proprietà temporali* i vincoli sono espressi su proprietà temporali del pattern, quali:
 - durata complessiva,
 - tempo occorso tra i suoi elementi.

La prima tipologia di operazioni, quella operante sulle proprietà del pattern, corrisponde a delle semplici operazioni di filtraggio (se il pattern non soddisfa la condizione specificata viene rimosso).

Diversamente dalla prima, la terza tipologia è un pò più complessa, in quanto per stabilire se un pattern soddisfa o meno una condizione, è necessario controllare i dati da cui il pattern è stato estratto; inoltre, bisogna tenere a mente che il pattern è stato estratto in quanto il suo supporto è maggiore del supporto minimo. Da ciò ne consegue che il pattern soddisfa una data condizione, solo se la percentuale di data-sequence che supporta il pattern e soddisfa la condizione continua ad essere maggiore del supporto minimo. Questo tipo di operazione, ridefinisce il processo di *support-counting*, ossia viene ridefinito in che modo una data-sequence supporta un pattern.

La seconda tipologia di operazioni, è invece un pò più controversa, in quanto in [11] viene presentata come una semplice operazione di filtraggio; ciò deriva dall'assunzione che le proprietà degli items, rimangono invariate nel tempo. Questo tipo di approccio però, può risultare poco realistico, si pensi ad esempio ai prezzi delle componenti di un computer: in poco tempo subiscono grandi variazioni. D'altra parte, assumere che le proprietà degli items rimangono invariate comporta una maggiore efficienza di calcolo, in quanto basterebbe una scansione dei dati contenenti le proprietà di ogni item per risolvere il problema. Al contrario, è necessario ridefinire il processo di support-counting, in quanto il fatto che un pattern continui ad essere frequente dipende dagli attributi degli items. Riprendendo l'esempio delle componenti di un computer, il prezzo degli items nel pattern $\langle (harddisk, memory) \rangle$ è soggetto a variazioni, quindi ricalcolando il supporto del pattern tenendo conto dei prezzi, il pattern potrebbe non essere frequente.

Altre operazioni sono state selezionate dalle definizioni, ad es. un filtro per tutti i pattern non massimali o un algoritmo per calcolare le regole sequenziali; altre invece, già presenti per le regole d'associazione, sono state adattate ai pattern sequenziali, ad es. la valutazione dei pattern rispetto ai dati da cui sono stati estratti.

Ovviamente, le operazioni da effettuare non si esauriscono qui, ad esempio un'altra operazione interessante potrebbe essere la classificazione di nuovi pattern ([4]).

Capitolo 4

Estensione del linguaggio: Specifica degli operatori

A livello di linguaggio, aggiungere un nuovo modello, significa definire:

- il formato del modello¹
- un'entità che specifichi il tipo dei nuovi operatori.
- degli operatori che permettano di:
 1. estrarre il nuovo modello, e
 2. caricarlo dal repository

Ovviamente, quanto appena detto, costituisce l'insieme minimo di operatori necessari, ma affinché abbia senso introdurre un nuovo modello, è necessario definire, almeno, un insieme di operatori che lavorino sul modello estratto (*postprocessing*) per *ragionare* sulla nuova *conoscenza*.

La figura 4.1 mostra l'entità `kdd_query_sequence`, che definisce il tipo degli operatori per i pattern sequenziali (tutti gli operatori che restituiscono un modello per i pattern sequenziali).

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

Figura 4.1: tipo degli operatori per pattern sequenziali

¹eventuali estensioni del modello PMML

4.1 Model access

4.1.1 PMML_SEQUENCE_LOADER

DTD

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

Importa un modello PMML contenente un insieme di pattern sequenziali. Nell'implementazione corrente, il KDDML supporta la versione 2.0 del PMML. Il modello implementato è un sotto-insieme del modello PMML. Gli elementi non supportati sono:

- SetPredicate
- Delimiter (parzialmente)

Per quanto riguarda l'elemento `Delimiter`, sono supportati solo quelli con attributi `delimiter` e `gap` uguali, rispettivamente a `acrossTimeWindow` e `unknown`.

Signature

$f_{\langle \text{PMML_SEQUENCE_LOADER} \rangle} : \text{empty} \rightarrow \text{sequence}$.

Attributi Obbligatori

- `pmml_source`: il sorgente PMML. I sorgenti PMML possono essere raccolti nel file system locale (ad es. `D:/MyRepository/coop.xml`) o nel web, tramite protocollo ftp (ad es. `ftp://www.foo.edu/PMML/coop.xml`).

4.1.2 SEQUENCE_LOADER

DTD

```
<!ELEMENT SEQUENCE_LOADER EMPTY>
<!ATTLIST SEQUENCE_LOADER xml_dest %string; #IMPLIED>
```

Carica un modello contenente insieme di pattern sequenziali dal repository di sistema.

Signature

$f_{\langle \text{SEQUENCE_LOADER} \rangle} : \text{empty} \rightarrow \text{sequence}$.

Attributi Obbligatori

- `xml_source`: il file sorgente xml contenuto nel repository.

4.2 Model extraction

4.2.1 SEQUENCE_MINER

DTD

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

Data una tabella ed un algoritmo di data mining, quest'operatore restituisce l'insieme di pattern sequenziali estratti dall'algoritmo specificato.

Essendo un generico operatore di mining, non esistono restrizioni sul formato della tabella, ma possono essere richieste da un particolare algoritmo.

Signature

$$f_{\langle \text{SEQUENCE_MINER} \rangle} : \text{table} \times \text{alg} \rightarrow \text{sequence}.$$

Prefix Span

L'algoritmo utilizzato per l'estrazione di pattern sequenziali è il *PrefixSpan* [10].

L'implementazione utilizzata all'interno del KDDML, è una versione main-memory dell'algoritmo, che permette di risparmiare i costi di proiezione ed elaborazione dei dati. Ovviamente, quest'implementazione ha dei limiti dovuti alla dimensione dei dati e delle relative strutture per la pseudo-proiezione.

Il PrefixSpan estrae l'insieme completo di pattern sequenziali. La tabella in input deve essere nel formato timestamp [2.1]; sono ammessi attributi supplementari, ma sono ignorati dall'operatore.

Parametri PrefixSpan

- **min_support**: supporto minimo dei pattern
- **max_number_of_sequences**: massimo numero di pattern estratti. I pattern restituiti, sono ordinati in base al supporto.

Nome	Valore	Uso
<i>min_support</i>	reale nell'intervallo (0,1]	obbligatorio
<i>max_number_of_sequences</i>	intero positivo	opzionale

Tabella 4.1: parametri prefix_span

4.3 Model Application

4.3.1 SEQUENCE_SATISFY

DTD

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

Dato un insieme di pattern sequenziali ed una tabella, l'operatore estrae tutte le data-sequence nella tabella che supportano almeno un pattern. La tabella in input, deve essere nel formato timestamp.

Ad esempio, data la data-sequence

$$\langle (10, \{bread, milk, mais\})(12, \{milk, bread, wine\})(25, \{mais\}) \rangle,$$

il pattern $\langle (milk, bread)(mais) \rangle$ è supportato dalla data-sequence, ma il pattern $\langle (milk, bread)(wine)(milk) \rangle$ no.

Quest'operatore, rimuove tutti i dati dalla tabella in input, che, rispetto al modello a cui sono applicati, non generano "informazione".

Signature

$$f_{\langle SEQUENCE_SATISFY \rangle} : \text{sequence} \times \text{table} \rightarrow \text{table}.$$

4.3.2 SEQUENCE_EXCEPTION

DTD

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

Dato un insieme di pattern sequenziali ed una tabella, l'operatore estrae tutte le data-sequence nella tabella che non supportano alcun pattern. La tabella in input, deve essere nel formato timestamp. È il duale dell'operatore precedente.

Signature

$$f_{\langle SEQUENCE_EXCEPTION \rangle} : \text{sequence} \times \text{table} \rightarrow \text{table}.$$

4.4 Model (meta-)reasoning

4.4.1 SEQUENCE_RULE

DTD

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

Dato un insieme di pattern sequenziali, l'operatore calcola tutte le corrispondenti regole sequenziali che soddisfano la confidenza minima.

Una regola è un'implicazione $X \rightarrow Y$, dove X , Y e $Z = X(Y)$ sono pattern sequenziali. La confidenza ed il supporto di una regola sono definiti come:

$$\text{confidence}(X \rightarrow Y) = \text{support}(Z) / \text{support}(X) ,$$

$$\text{support}(X \rightarrow Y) = \text{support}(Z) .$$

Per costruire correttamente tutte le regole sequenziali, questo operatore necessita dell'insieme completo di pattern sequenziali estratti, altrimenti il supporto dei sotto-pattern non può essere calcolato (ad es. applicando l'operatore `SEQUENCE_RULE` sull'insieme di pattern restituito dall'operatore `SEQUENCE_MAXIMAL_FILTER` [4.4.3], non verrà costruita nessuna regola).

Signature

$f_{\langle \text{SEQUENCE_RULE} \rangle} : \text{sequence} \rightarrow \text{sequence}$.

Attributi Obbligatori

- `min_confidence`: confidenza minima delle regole, reale $\in (0, 1]$

Attributi Opzionali

- `max_number_of_rules`: massimo numero di regole calcolate, intero positivo. Le regole restituite, sono quelle con confidenza più alta.

4.4.2 SEQUENCE_FILTER

DTD

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

Dato un insieme di sequenze, l'operatore restituisce tutte quelle che soddisfano una data condizione.

Una condizione rappresenta un vincolo, o una composizione booleana di vincoli. Un vincolo è una funzione booleana, definita come:

$$op_type(seq_attr, term) ,$$

dove *op_type* è la funzione (vincolo), *seq_attr* è un attributo della sequenza (su cui applicare il vincolo) e *term* una costante. Quest'operatore raggruppa un insieme di vincoli base, riguardanti:

- il supporto, la cardinalità, il numero di item distinti ed il massimo numero di item per ogni elemento, di una sequenza;
- gli elementi di una sequenza;

Signature

$$f_{\langle \text{SEQUENCE_FILTER} \rangle} : \text{sequence} \times \text{condition} \rightarrow \text{sequence}.$$

Specifica della condizione

Per specificare una condizione, viene utilizzato l'elemento `CONDITION` (figura 2.5) messo a disposizione dal linguaggio.

I vincoli vengono espressi utilizzando l'elemento `BASE_COND`; in particolare il vincolo viene identificato tramite l'attributo `op_type`.

I vincoli esprimibili sulle sequenze sono:

- **equal**, **not_equal**, **greater**, **greater_or_equal**, **less**, **less_or_equal** permettono di specificare vincoli su
 - supporto (`term1=@sequence_support`)
 - cardinalità (`term1=@sequence_cardinality`)
 - numero di items distinti (`term1=@sequence_distinct_items_cardinality`)
 - massimo numero di items per ogni elemento (`term1=@max_number_of_items_per_set`)

di una sequenza (per es. solo le sequenze con supporto uguale ad 1, o le sequenze che contengono almeno 5 items distinti);

- **is_in** controlla se un item appartiene ad una sequenza (per es. solo sequenze che contengono l'item "milk");
- **is_in_all** controlla se un item appartiene a tutti gli elementi di una sequenza (per es. solo le sequenze in cui tutti gli elementi contengono l'item "milk");

- **is_not_in** controlla se esiste almeno un elemento della sequenza che non contiene l'item (per es. solo sequenze in cui almeno un elemento non contiene l'item "milk");
- **is_not_in_all** controlla se un item non appartiene a tutti gli elementi di una sequenza (per es. solo le sequenze in cui tutti gli elementi non contengono l'item "milk");
- **sub_sequence** controlla se una sequenza in input è contenuta nella sequenza. La sequenza in input viene passata come una stringa, nel seguente formato: ogni elemento di una sequenza è espresso come una lista di items nel formato csv; tutti gli elementi sono separati da ";" (per es. solo sequenze che sono super-sequenze di "milk,soap;wine,tomato").

La tabella 4.2 contiene il tipo dei termini da usare per ogni op_type.

OpType	Term 1	Term 2
equal	@sequence_support	reale in (0,1]
not_equal	@sequence_cardinality	intero positivo
greater	@sequence_distinct_items_cardinality	
greater_or_equal	@max_number_of_items_per_set	
less		
less_or_equal		
is_in	@sequence	stringa
is_in_all		
is_not_in		
is_not_in_all		
sub_sequence	@sequence	un insieme di liste di stringhe in formato csv, separate da ";"

Tabella 4.2: specifica di BASE_COND per l'operatore SEQUENCE_FILTER

Esempio 4.1 (Specifica di una CONDITION per l'operatore SEQUENCE_FILTER).

```

<CONDITION>
  <AND_COND>
    <BASE_COND op_type="greater"
      term1="@sequence_support" term2="0.8">
    <BASE_COND op_type="is_not_in" term1="@sequence"
      term2="matherboard">
    <BASE_COND op_type="is_not_in" term1="@sequence"
      term2="cpu">
    <BASE_COND op_type="is_not_in" term1="@sequence"

```

```

                                term2="ram">
<BASE_COND op_type="is_not_in" term1="@sequence"
                                term2="monitor">
<OR_COND>
  <BASE_COND op_type="subsequence" term1="@sequence"
                                term2="hard_disk,cooler;cooler">
  <BASE_COND op_type="subsequence" term1="@sequence"
                                term2="hard_disk;cooler">
</OR_COND>
</AND_COND>
</CONDITION>

```

Questa condizione restituisce tutte le sequenze con supporto maggiore dell'80%, che non contengono gli items "matherboard", "cpu", "ram" o "monitor", e che siano super-sequenze di "hard_disk,cooler;cooler" o "hard _disk;cooler".

4.4.3 SEQUENCE_MAXIMAL_FILTER

DTD

```

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

```

Dato un insieme di sequenze, restituisce tutte le sequenze massimali, i.e. tutte e sole le sequenze che non sono contenute in altre (sequenze).

Signature

$f_{\langle \text{SEQUENCE_MAXIMAL_FILTER} \rangle} : \text{sequence} \rightarrow \text{sequence}.$

4.4.4 SEQUENCE_AGGREGATE_FILTER

DTD

```

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

```

Dato un insieme di sequenze ed una tabella, l'operatore restituisce tutte le sequenze che soddisfano una data condizione.

Una condizione rappresenta un vincolo, o una composizione booleana di vincoli.

Un vincolo è una funzione booleana, definita come:

$op_type(f_{agg}(table_attr), term),$

dove op_type è la funzione (vincolo), f_{agg} è una funzione di aggregazione, $table_attr$ l'attributo della tabella su cui calcolare la funzione d'aggregazione e $term$ una costante.

In questo operatore, un vincolo è calcolato su una aggregazione di item in una sequenza. Le funzioni di aggregazione sono: *sum*, *avg*, *max*, *min*, *standard deviation*. Si possono esprimere condizioni del tipo: $sum(@price) > 100 \wedge min(@quantity) \geq 3$.

Introduzione

1. Data una sequenza $s = \langle s_1, \dots, s_n \rangle$ e una data- sequence $d = \langle d_1, \dots, d_m \rangle$, d supporta s se:

$$\exists 1 \leq i_1 < i_2 < \dots < i_n \leq m, \text{ tali che } s_1 \subseteq d_{i_1}, \dots, s_n \subseteq d_{i_n}.$$

2. d_{i_1}, \dots, d_{i_n} sono le transazioni della data-sequence d che supportano gli elementi di s .
3. $X_j = d_{i_j} \cap s_j$, è l'insieme degli item in d_{i_j} contenuti in s_j , $j = 1, \dots, n$.
4. $T = \{X_j, j = 1, \dots, n\}$, è l'insieme delle transazioni (e dei rispettivi items) di d , che supportano gli elementi di s .
5. $f_{attr} : X \rightarrow V$, data una transazione X , f_{attr} ritorna, per ogni item in X , il valore dell'attributo $attr$; V è l'insieme di questi valori.
6. $V_s = \{f_{attr}(X_j), j = 1, \dots, n\}$, è l'insieme dei valori - rispetto all'attributo $attr$ - degli items nella sequenza s , calcolati nelle transazioni di d che supportano s .

Data una data-sequence d , una sequenza s , ed un vincolo (come appena definito), d supporta s , se:

1. $s \sqsubseteq d$, e
2. $op_type(f_{agg}(V_s), term2)$ restituisce true

Quest'operatore ridefinisce il processo di *support-counting*, vincolando come una data-sequence supporta una sequenza. L'insieme delle sequenze restituito dall'operatore è composto da tutte quelle il cui supporto - ridefinito sulla condizione specificata - è maggiore del supporto minimo.

Consideriamo, ad esempio, la data-sequence nella tabella 4.3, e la sequenza $\langle (spaghetti)(tomato)(spaghetti, wine) \rangle$. L'insieme di transazioni che supportano la sequenza sono quelle in mostrate in tabella 4.4. Applicando il vincolo $sum(@price) > 11$, si ottiene $sum(3, 2, 6, 7) > 11$, quindi la sequenza continua ad essere supportata.

Signature

$f_{\langle SEQUENCE_AGGREGATE_FILTER \rangle} : sequence \times table \times condition \rightarrow sequence.$

seq_id	timestamp	item	price	quantity
id_1	10	spaghetti	3	5
id_1	11	tomato	2	2
id_1	11	wine	4	1
id_1	20	spaghetti	6	4
id_1	20	tomato	3	2
id_1	20	wine	7	1

Tabella 4.3: data-sequence

seq_id	timestamp	item	price	quantity
id_1	10	spaghetti	3	5
id_1	11	tomato	2	2
id_1	20	spaghetti	6	4
id_1	20	wine	7	1

Tabella 4.4: transazioni

Specifica della condizione

Per specificare una condizione, viene utilizzato l'elemento `CONDITION` (figura 2.5) messo a disposizione dal linguaggio.

I vincoli vengono espressi utilizzando l'elemento `BASE_COND`; in particolare il vincolo viene identificato tramite l'attributo `op_type`.

I vincoli esprimibili sulle sequenze sono:

- **equal, not_equal, greater, greater_or_equal, less, less_or_equal** permettono di specificare vincoli su una aggregazione di item in una sequenza (e.g. solo le sequenze dove la media dei prezzi degli item è maggiore di 100);

La tabella 4.5 contiene il tipo dei termini da usare per ogni `op_type`.

OpType	Term 1	Term 2	Term 3
equal, not_equal greater, greater_or_equal less, less_or_equal	sum, avg max, min stddev	numeric_attribute_name	reale positivo

Tabella 4.5: specifica di `BASE_COND` per l'operatore `SEQUENCE_AGGREGATE_FILTER`

Esempio 4.2.

```
<CONDITION>
  <AND_COND>
```

```

<BASE_COND op_type="greater" term1="avg" term2="@price"
                                     term3="10"/>
<BASE_COND op_type="greater_or_equal" term1="min"
                                     term2="@quantity" term3="2"/>
</AND_COND>
</CONDITION>

```

Questa condizione è una composizione in AND di vincoli. Il primo vincolo, richiede che la media dei prezzi degli item, in tutti gli elementi di una sequenza, sia maggiore di 100; il secondo, che ciascun item, in tutti gli elementi di una sequenza, sia stato acquistato almeno 2 volte.

4.4.5 SEQUENCE_TIMESTAMP_FILTER

DTD

```

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

```

Quest'operatore raggruppa due vincoli, *duration* e *gap*. Entrambi i vincoli operano su un fattore temporale, il *timestamp* delle transazioni in input.

In generale il timestamp definisce un ordinamento parziale tra le transazioni; questo operatore richiede che l'attributo sia di tipo numerico.

Dato un insieme di sequenze ed una tabella, l'operatore restituisce tutte le sequenze che soddisfano il dato vincolo.

Gli attributi *interval_closure*, *lower_bound* e *upper_bound*, definiscono un intervallo I .

L'attributo *constr_type* è un vincolo sulla durata della sequenza (il tempo complessivo in cui le transazioni che supportano la sequenza sono occorse), o un vincolo sul tempo intercorso tra due transazioni che supportano due elementi consecutivi di una sequenza (ad es. solo le sequenze in cui ogni elemento occorre a distanza di un giorno da quelli adiacenti).

Duration impone dei limiti alla durata delle transazioni che supportano la sequenza.

Data una data-sequence $d = \langle d_1, \dots, d_n \rangle$, dove $d_i.timestamp$ identifica il timestamp della transazione d_i , ed una sequenza $s = \langle s_1, \dots, s_m \rangle$, d supporta s , se:

1. $s \sqsubseteq d$, e

$$2. d_{s_m}.timestamp - d_{s_1}.timestamp \in I$$

dove d_{s_1} e d_{s_m} corrispondono, rispettivamente, alla prima ed all'ultima transazione, della data-sequence d , tra quelle che supportano la sequenza s .

Gap impone dei limiti sul tempo intercorso tra due transazioni - in una data-sequence - che supportano due elementi consecutivi di una la sequenza.

Data una data-sequence $d = \langle d_1, \dots, d_n \rangle$, dove $d_i.timestamp$ identifica il timestamp della transazione d_i , ed una sequenza $s = \langle s_1, \dots, s_m \rangle$, d supporta s , se:

$$1. s \sqsubseteq d, \text{ e}$$

$$2. \forall 1 < i \leq m, d_{s_i}.timestamp - d_{s_{i-1}}.timestamp \in I$$

dove d_{s_i} è la transazione che contiene i -esimo l'elemento della sequenza s .

Come per l'operatore `SEQUENCE_AGGREGATE_FILTER`, quest'operatore ridefinisce il processo di *support-counting*, vincolando come una data-sequence supporta una sequenza.

L'insieme delle sequenze restituito dall'operatore è composto da tutte quelle il cui supporto - ridefinito sulla condizione specificata - è maggiore del supporto minimo.

Consideriamo, ad esempio, la data-sequence nella tabella 4.3 e la sequenza $s = \langle (spaghetti)(tomato)(spaghetti, wine) \rangle$. L'insieme di transazioni che supportano la sequenza sono mostrate in tabella 4.4. Applicando il vincolo $gap(s) \in [1, 7]$, si ottiene $gap(10, 11, 20) \in [1, 7]$, quindi la sequenza non è più supportata dalla data-sequence, perchè la differenza di timestamp tra le transazioni che supportano, rispettivamente, $(tomato)$ e $(spaghetti, wine)$ viola il vincolo specificato.

Esempio 4.3.

```
<SEQUENCE_TIMESTAMP_FILTER
    constr_type="duration"
    interval_closure="open_open"
    upper_bound="30">
    ...
    ...
</SEQUENCE_TIMESTAMP_FILTER>
```

Il vincolo specificato, richiede che l'insieme di transazioni, in ogni data-sequence che contribuisce al supporto del pattern, siano avvenute entro 30 giorni. Questo significa che, ad esempio, la sequenza di acquisti, rappresentata dal pattern, è avvenuta entro 30 giorni.

Signature

$f_{\langle \text{SEQUENCE_TIMESTAMP_FILTER} \rangle} : \text{sequence} \times \text{table} \rightarrow \text{sequence}.$

Attributi Obbligatori

- `constr_type`: il vincolo da applicare, (*duration* o *gap*)
- `interval_closure`: il tipo d'intervallo, (*open_open*, *closed_closed*, *open_closed* o *closed_open*). `lower_bound` e `upper_bound` definiscono i limiti dell'intervallo.

Attributi Opzionali ²

- `lower_bound`: limite inferiore, reale $\in (0, \infty]$. Se omissso, si assume $-\infty$.
- `upper_bound`: limite superiore, reale $\in (0, \infty]$. Se omissso, si assume $+\infty$.

²almeno uno tra `lower_bound` e `upper_bound` deve essere specificato

Capitolo 5

Estensione del sistema: Livello Core

Come introdotto nel capitolo 2, il livello *Core* racchiude tutte le funzionalità per la gestione/accesso di/ai dati/modelli.

A livello di sistema, integrare un nuovo modello, significa definire:

1. un formato appropriato per la rappresentazione dei dati in input (se quelli presenti non ne offrono uno);
2. un formato per la sua rappresentazione;
3. dei moduli di accesso per i livelli superiori.

5.1 Input: formato e supporto

Di seguito verrà presentato il formato adottato per la rappresentazione dei dati in input, e le strutture di supporto per la sua gestione.

5.1.1 Formato dei dati

I dati vengono rappresentati, internamente, sotto forma di tabelle relazionali. Una tabella KDDML è composta da:

- uno *schema*, che definisce il tipo degli attributi e include alcune statistiche sui valori degli attributi;
- i *dati fisici*, in formato csv (comma separated value).

I dati fisici possono essere rappresentati in diversi formati: *relazionale*, *transazionale*, e *timestamp*.

I formati precedentemente presenti (relazionale e transazionale), non sono in grado di fornire un input adeguato agli operatori per pattern sequenziali, per questo motivo è stato necessario estendere la rappresentazione dei dati, aggiungendo il formato *timestamp*.

Le tabelle 5.1 e 5.2, mostrano, rispettivamente, una tabella in formato relazionale ed una in formato timestamp.

transaction	timestamp	spaghetti	pane	vino	pomodoro	...
id_1	10	1	0	0	0	0 ...
id_1	15	1	0	1	0	0 ...
id_1	20	1	1	1	1	0 ...

Tabella 5.1: tabella in formato relazionale

transaction	timestamp	event
id_1	10	spaghetti
id_1	15	spaghetti
id_1	15	vino
id_1	20	spaghetti
id_1	20	pomodoro
id_1	20	vino
id_1	20	pane

Tabella 5.2: tabella in formato timestamp

Supponiamo di voler rappresentare le transazioni degli utenti di un supermercato. Rappresentando i dati utilizzando il formato relazionale, ogni record dovrebbe contenere tutti i prodotti in vendita, quando i prodotti effettivamente acquistati sono decisamente inferiori.

Nel formato transazionale, invece, le transazioni a livello utente sono uniche, ossia non esiste la possibilità di associare transazioni distinte ad un stesso utente; inoltre, non esiste un *ordinamento temporale* tra le transazioni e gli eventi che vi sono occorsi.

Il formato *timestamp*, costituisce la soluzione naturale alle problematiche appena descritte. Questo formato, come descritto in precedenza, estende il formato transazionale stabilendo un ordinamento parziale tra gli eventi in una transazione.

È importante notare che, per quanto concerne i pattern sequenziali, il termine *transazione* è utilizzato scorrettamente, in quanto concettualmente, l'attributo **transaction**, rappresenta un generico *oggetto* su cui, in un dato momento, occorrono un insieme di eventi. Nel capitolo precedente, come in quelli successivi, il termine adottato per identificare una *transazione* nel formato timestamp, è *data-sequence*. Qualora fosse necessario utilizzare il termine *transazione*, verrà opportunamente specificato il suo significato.

Negli esempi 5.1 e 5.2, viene dato un possibile significato degli attributi del formato timestamp, nel caso in cui si vogliano rappresentare rispettivamente, le transazioni degli utenti di un supermercato e le cartelle cliniche dei pazienti ricoverati.

Esempio 5.1 (market basket analysis).

- `transaction`, *identificatore utente*,
- `timestamp`, *data dell'acquisto*,
- `event`, *articolo acquistato*.

Esempio 5.2 (medical record analysis).

- `transaction`, *identificatore paziente*,
- `timestamp`, *data del controllo medico*,
- `event`, *sintomo registrato*.

5.1.2 Estensione del supporto

Una volta definito un formato adeguato per la rappresentazione dei dati, è stato necessario estendere il supporto per la sua gestione nel sistema.

La figura 5.1 mostra la rappresentazione dei vari formati all'interno del sistema. La classe ***TimestampInternalTable*** rappresenta una tabella in formato timestamp.

Oltre alle operazioni base (accesso allo schema, ai singoli record ed alle statistiche sugli attributi) fornite (ereditate) dalla classe ***InternalTable*** (l'astrazione principale per la rappresentazione dei dati), questa classe fornisce l'accesso ad un'intera sequenza di dati (*data-sequence*), i.e. tutti i record con lo stesso identificatore di transazione (l'attributo `transaction` del formato) ordinati rispetto al `timestamp`; inoltre, vista la necessità di accedere ad attributi opzionali associati agli items, fornisce alcune operazioni per caricare tali attributi come *features* degli items.

DataSequence

La figura 5.2 mostra la rappresentazione di una data-sequence nel sistema. Un oggetto ***DataSequence*** rappresenta tutti i record con lo stesso identificatore di transazione (attributo `transaction`), ed è composto da un insieme ordinato di ***TimestampedTransaction***. Un oggetto ***TimestampedTransaction***, rappresenta tutti i record con lo stesso `timestamp`, ed è composto da un insieme di ***Item***. Un oggetto ***Item*** rappresenta il singolo dato (attributo `event`), ed opzionalmente, può avere associate delle *features*. Una *feature*, rappresentata dall'oggetto ***ItemFeature***, è un attributo opzionale associato ad un item (ad es. il prezzo); di ogni attributo, vengono specificati *nome*, *tipo* e *valore*.

TimestampInternalTable

Le operazioni principali di questa classe, sono:

- ***void initFeatures(String[] names, int feature_type)***
specifica quali attributi opzionali devono essere caricati per ogni item, ossia quali *features* associare.

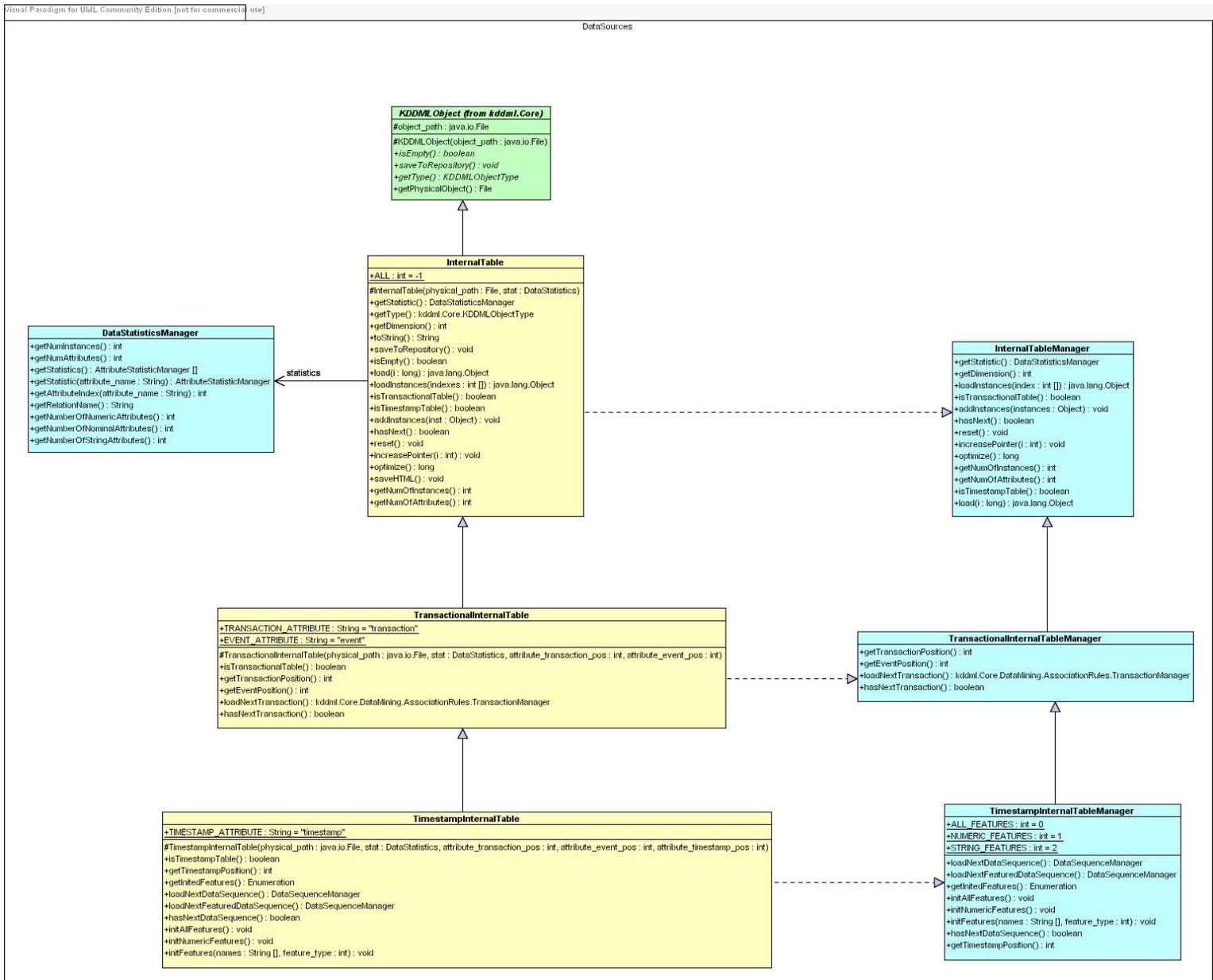


Figura 5.1: Package DataSources

- *names* è una lista con i nomi degli attributi da caricare, eventualmente vuota;
- *type*, il tipo.

Ad es. *initFeatures(null, NUMERIC)* specifica che devono essere caricati tutti gli attributi di tipo numerico contenuti nella tabella.

- *boolean hasNextDataSequence()*

controlla se esistono data-sequence nella tabella che non siano state ancora caricate.

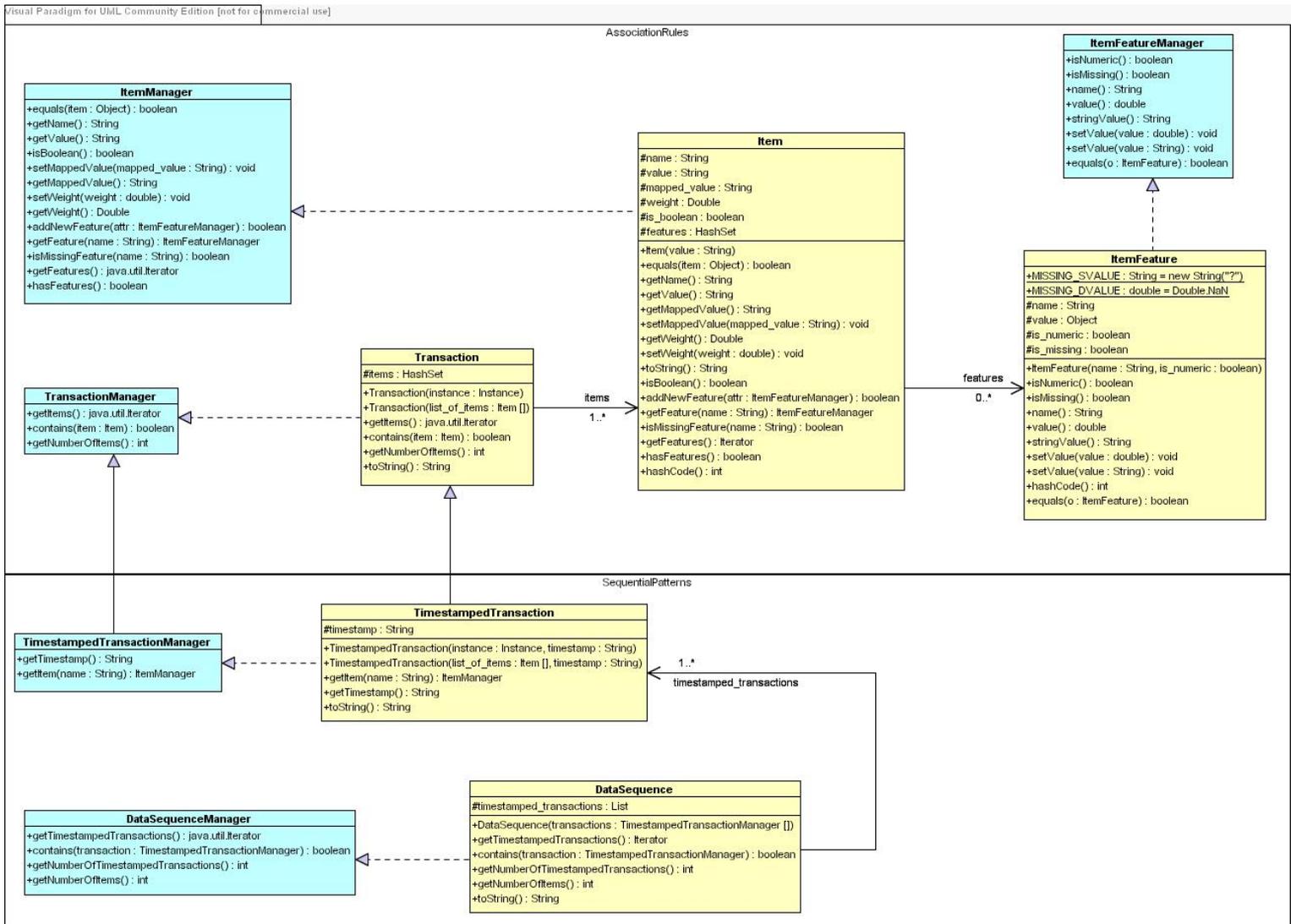


Figura 5.2: DataSequence

- *DataSequenceManager loadNextDataSequence()*
carica la data-sequence corrente dalla tabella.
- *DataSequenceManager loadNextFeaturedDataSequence()*
come *loadNextDataSequence()*, ad eccezione del fatto che per ogni item, vengono caricati gli attributi specificati in *initFeatures()*.
Consideriamo ad esempio la tabella 5.3. Richiamando *initFeatures({price, quantity}, ALL)*, ogni item, avrà due *feature* associate, contenenti i valori degli attributi *price* e *quantity*; ad es. l'item *spaghetti* nella transazione con *timestamp=10*, avrà associati, *price=.5* e *quantity=3*.

transaction	timestamp	event	price	quantity
id_1	10	spaghetti	0.5	3
id_1	15	spaghetti	0.5	2
id_1	15	vino	5	1
id_1	20	spaghetti	0.55	2
id_1	20	pomodoro	1.5	3
id_1	20	vino	6	2
id_1	20	pane	3	1

Tabella 5.3: tabella in formato timestamp con attributi opzionali

```
<!ELEMENT SequenceModel (Extension*, MiningSchema, Item*, Itemset*,
                          SetPredicate*, Sequence+,
                          SequenceRule*, Extension*)>
```

Figura 5.3: PMML Sequence Model

5.2 Modello: rappresentazione ed integrazione

I modelli estratti, sono rappresentati internamente come estensione dei modelli *PMML*.

Il primo passo per la rappresentazione di un modello per pattern sequenziali è stato lo studio del `SequenceModel` definito nel linguaggio PMML. Nella figura 5.3 viene riportato un frammento del DTD. Per semplicità è stato omesso il DTD degli attributi, presentati in seguito.

Un modello per pattern sequenziali è costituito principalmente da:

- un `MiningSchema`, fornisce informazioni sugli attributi utilizzati per l'estrazione del modello
- `Sequence`, l'insieme dei pattern sequenziali estratti
- `SequenceRule`, le regole sequenziali calcolate sui pattern estratti

Nell'implementazione corrente, il modello è un sotto-insieme di quello PMML. Di seguito, verrà descritto il modello implementato, e quali funzionalità del PMML non sono supportate.

5.2.1 Integrazione del nuovo modello

SequenceModel in KDDML

Il primo passo per integrare un nuovo modello è l'estensione della classe astratta *MiningModel*. Questa classe, rappresenta un generico modello, ed estende la classe *KDDMLObject*, che è la radice di tutti gli oggetti del sistema.

La figura 5.4 mostra la rappresentazione del modello nel sistema. Il modello è composto da un insieme di:

- *SequenceManager*, le sequenze (pattern sequenziali) estratte;
- *SequenceRuleManager*, le regole calcolate;
- *SequenceElementManager*, gli elementi che compongono le sequenze;
- *ItemManager*, gli items che compongono gli elementi delle sequenze.

La figura 5.5 mostra in dettaglio come regole, sequenze ed elementi sono composti e relazionati tra di loro.

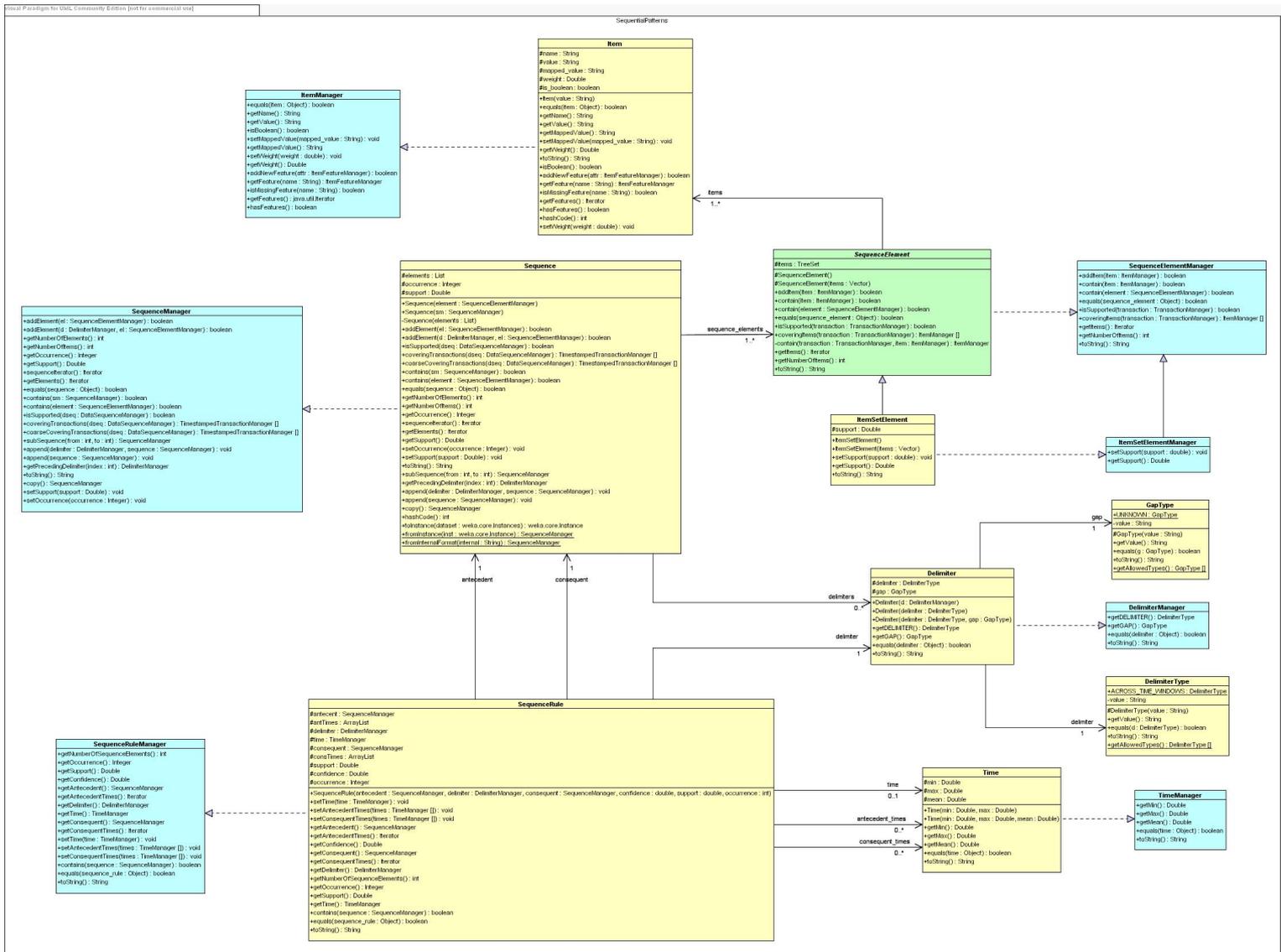


Figura 5.5: Rule, Sequence e SequenceElement Manager

Caratteristiche del modello PMML (KDDML):

- ***numberOfTransactions***
il numero di transazioni (data-sequence)
- ***maxNumberOfItemPerTransaction***
il massimo numero di items per transazione (data-sequence)
- ***avgNumberOfItemPerTransaction***
il numero medio di items per transazione (data-sequence)
- ***minimumSupport***
il supporto minimo di una sequenza
- ***minimumConfidence***
la confidenza minima di una regola
- ***lengthLimit***
la massima lunghezza di una sequenza
- ***timeWindowWidth***
è utilizzato per separare gli items in una transazione (data-sequence) in eventi discreti (timestamped-transaction), solo se non esiste una *chiave* per distinguere gli eventi (un evento nel PMML è definito come un insieme di items). Due items consecutivi, devono occorrere in un intervallo minore della soglia specificata per essere considerati parte dello stesso evento (essere associati allo stesso timestamp).
- ***minimumTime***
il minimo tempo tra gli items come definito in *timeWindowWidth*
- ***maximumTime***
il massimo tempo tra gli items come definito in *timeWindowWidth*

Bisogna notare che, *timeWindowWidth*, *minimumTime* e *maximumTime* sono utilizzati per adattare i dati in input all'estrazione dei pattern. Anche se presenti, l'implementazione corrente, non supporta tali caratteristiche, in quanto tutti gli operatori utilizzano come input tabelle nel formato timestamp. Tale formato, definisce una chiave, l'attributo `timestamp`, per distinguere gli "eventi" (da non confondere con l'attributo `event` del formato timestamp).

Principali funzionalità fornite:

- ***boolean addRule(RuleManager rule)***
aggiunge un nuova regola al modello; per essere aggiunta, la nuova regola, deve rispettare il minimo supporto e la minima confidenza del modello
- ***boolean removeRule(RuleManager rule)***
rimuove una regola dal modello
- ***boolean addSequence(SequenceManager sequence)***
aggiunge un nuova sequenza al modello; per essere aggiunta, la nuova sequenza, deve rispettare il minimo supporto del modello

- ***boolean removeSequence(SequenceManager sequence)***
rimuove una sequenza dal modello; rimuovendo una sequenza, vengono rimosse tutte le regole che la contengono
- ***SequenceModelManager clone(File output_path)***
restituisce una “bassa” copia del modello; tutti gli oggetti contenuti nel modello (regole, sequence, elementi ed item) non vengono clonati (riferiscono gli stessi oggetti del modello clonato)
- ***Object sequencesToInstances()***
trasforma le sequenze contenute nel modello in istanze (un oggetto ***Instances*** della libreria *weka*).
Ogni istanza (***Instance***) ha cinque attributi:
 - *sequence* una rappresentazione sotto forma di stringa,
 - *sequence_support* il supporto,
 - *sequence_cardinality* la lunghezza,
 - *sequence_distinct_items_cardinality* il numero di items distinti,
 - *max_number_of_items_per_set* il massimo numero di items per ogni elemento di una sequenza.

Le altre funzionalità fornite riguardano l’accesso ai vari attributi del modello ed a gli oggetti che lo compongono.

SequenceElement

Nel PMML gli elementi che compongono una sequenza possono essere sia itemset (***Itemset***: un insieme di items con, opzionalmente, il relativo supporto), sia predicate-set (***SetPredicate***: un’espressione booleana composta da un operatore - *supersetOf* -, un campo - il “soggetto” dell’espressione - ed il valore da confrontare - un array di items).

Un oggetto ***SequenceElement*** rappresenta un generico elemento di una sequenza, ossia un insieme di items. Questa generalizzazione, permette facilmente di estendere il sistema per supportare l’elemento ***SetPredicate*** definito nel PMML.

L’unica rappresentazione per un elemento di una sequenza, attualmente, è data dall’oggetto ***ItemSetElement***, che implementa un itemset.

Funzionalità fornite dalla classe ***SequenceElement***:

- ***boolean addItem(ItemManager item)***
aggiunge un nuovo item
- ***boolean contains(ItemManager item)***
controlla se un dato item è contenuto nell’elemento
- ***boolean contains(SequenceElementManager el)***
controlla se un dato elemento è contenuto nell’elemento; se è un sotto-insieme

- *boolean equals(Object el)*
confronta due elementi; due elementi sono uguali se contengono gli stessi items
- *Iterator getItems()*
restituisce un iteratore su gli items nell'elemento
- *boolean isSupported(TransactionManager transaction)*
controlla se l'elemento è supportato da una transazione, i.e. se gli items nella transazione contengono quelli nell'elemento
- *ItemManager[] coveringItems(TransactionManager transaction)*
se l'elemento è supportato dalla transazione, restituisce tutti e soli gli items nella transazione che sono contenuti nell'elemento, altrimenti restituisce `null`.

Delimiter

Un oggetto *Delimiter* definisce le relazioni tra due elementi di una sequenza o tra la sequenza antecedente e conseguente di una regola; è composto da un *DelimiterType* ed un *GapType*

DelimiterType Un oggetto *DelimiterType*, rappresenta l'attributo `delimiter` dell'elemento *Delimiter* definito nel PMML. Tale attributo stabilisce se i due elementi *SetPredicate* separati dall'elemento *Delimiter*, sono occorsi nello stesso evento (hanno lo stesso *timestamp*) o *periodo di tempo*, come definito dall'attributo *timeWindowWidth* del modello.

I possibili valori sono:

- *sameTimeWindow*, i due elementi sono occorsi nel stesso evento o periodo di tempo;
- *acrossTimeWindow*, altrimenti.

La classe *DelimiterType*, è una collezione dei tipi di delimiter ammessi. Nell'implementazione corrente, l'unico valore ammesso è `acrossTimeWindow`.

GapType Un oggetto *GapType*, rappresenta l'attributo `gap` dell'elemento *Delimiter*. Tale attributo, stabilisce la possibile esistenza di elementi *SetPredicate*, tra i due elementi separati dal *Delimiter*.

I possibili valori sono:

- *true*, indica l'esistenza di *SetPredicate* tra due elementi, $[1, \infty]$;
- *unknown*, indica la possibile esistenza di *SetPredicate* tra due elementi, $[0, \infty]$;
- *false*, indica che i due elementi separati dal delimiter, sono insiemi consecutivi nei dati in input.

La classe *GapType*, è una collezione dei tipi di gap ammessi. Nell'implementazione corrente, l'unico valore ammesso è `unknown`.

Sequence

Un oggetto *Sequence* rappresenta un pattern sequenziale; è composto da un insieme ordinato di *SequenceElement*. Ogni elemento è separato da quello successivo da un oggetto *Delimiter*, con il significato descritto in precedenza.

Oltre ad operazioni base sulla sequenza, quali:

- ***boolean addElement(SequenceElementManager el)***
aggiunge un nuovo elemento alla fine della sequenza
- ***void append(SequenceManager seq)***
aggiunge tutti gli elementi della sequenza data, alla fine della sequenza
- ***boolean contains(SequenceElementManager el)***
controlla se la sequenza contiene un dato elemento
- ***SequenceManager subSequence(int from, int to)***
restituisce una nuova sequenza comprendente tutti gli elementi nell'intervallo [from,to)
- ***boolean equals(Object seq)***
confronta due sequenze; due sequenze sono uguali se hanno gli stessi elementi nello stesso ordine
- ***SequenceManager copy()***
restituisce una copia della sequenza
- ***Iterator getElements()***
restituisce un iteratore sugli elementi della sequenza

fornisce delle operazioni utilizzate largamente dagli operatori, quali:

- ***boolean contains(SequenceManager seq)***
controlla se la sequenza contiene un'altra sequenza; è l'implementazione dell'operatore \sqsubseteq :
una sequenza a contiene una sequenza b se esistono $i_1 < i_2 < \dots < i_n$ tali che $b_1 \subseteq a_{i_1}, b_2 \subseteq a_{i_2}, \dots, b_n \subseteq a_{i_n}$
- ***boolean isSupported(DataSequenceManager ds)***
controlla se la sequenza è supportata da una data-sequence; una data-sequence d supporta una sequenza s , se e solo se, s è una sottosequenza di d ($s \sqsubseteq d$).
- ***TimestampedTransactionManager[] coveringTransactions(DataSequenceManager ds)***
se la sequenza è supportata dalla data-sequence, restituisce tutte e sole le transazioni (timestamped-transaction) nella data-sequence che supportano la sequenza, altrimenti restituisce null; ogni transazione restituita, conterrà solo gli items contenuti nell'elemento della sequenza che sta supportando:
sia d_{i_j} la transazione di d che supporta l'elemento s_j della sequenza s ($s_j \subseteq d_{i_j}$), gli items nella transazione restituita saranno: $s_j \cap d_{i_j}$

- *TimestampedTransactionManager*[/] *coarseCoveringTransactins(DataSequenceManager ds)*
 come *coveringTransactions()*, ad eccezione del fatto che le transazioni (timestamped-transaction) sono restituite “come sono”: ogni transazione conterrà l’insieme di items presenti in input

SequenceRule

Un oggetto *SequenceRule* rappresenta una regola sequenziale. Una regola sequenziale è composta da una sequenza *antecedente* ed una *conseguente* separate da un oggetto *Delimiter*. Opzionalmente, è possibile specificare il tempo (*Time*) che intercorre tra:

1. la sequenza antecedente e conseguente di una regola (*rule time*),
2. gli elementi della sequenza antecedente (*antecedent times*), e
3. gli elementi della sequenza conseguente (*consequent times*).

5.2.2 Guida all’estensione del modello

Il modello implementato, è un sotto-insieme del modello PMML. Estendere il modello per supportare altre funzionalità del PMML, non richiede, comunque, molta “fatica”.

SetPredicate Come è già stato accennato precedentemente, un elemento **SetPredicate** è composto da un’espressione booleana. L’espressione è formata da:

- un campo, che solitamente fa riferimento al **TransformationDictionary** di un documento PMML (non supportato dal sistema); in generale fa riferimento ad un attributo dei dati in input.
- un operatore, nella versione corrente del PMML l’unico operatore è **supersetOf**.
- un array di valori (items).

Aggiungere il supporto per i **SetPredicate**, equivale, principalmente, ad estendere la classe astratta **SequenceElement**, ridefinendo, se necessario alcune operazioni.

Delimiter L’estensione dei **Delimiter** ammessi, può risultare un’operazione un pò più complicata. Per quanto riguarda il loro supporto, basta aggiungere i tipi di **delimiter** e **gap** supportati, rispettivamente, nella classe **DelimiterType** e **GapType**. Come conseguenza, però, potrebbe essere necessario ridefinire alcune operazioni della classe **Sequence**, poichè cambiando le relazioni tra gli elementi di una sequenza, può cambiare anche il comportamento della sequenza stessa.

5.3 Interfacciamento con il sistema

L'accesso ai modelli avviene tramite moduli wrapper. Questi moduli, sono organizzati in classi "factory". Questo approccio, permette di separare l'accesso al modello dalla sua interfaccia di gestione.

Per quanto riguarda i pattern sequenziali, sono previsti tre moduli wrapper principali, rispettivamente per l'accesso a:

1. nuovi modelli (proprietary)
2. modelli nel repository
3. modelli non proprietari

Accesso a nuovi modelli Per ogni modello nel sistema, vengono implementate un insieme di classi factory che permettono di accedere ad un nuovo modello, parametrizzandone la creazione.

Accesso a modelli nel repository L'accesso ai modelli nel repository di sistema, avviene tramite una classe factory, che si occupa di trasformare il modello in un *SequenceModel*.

Accesso a modelli non proprietari È possibile importare modelli non proprietari nel sistema. Per ogni tipo di modello supportato, esiste una classe factory che si occupa di importare il modello e trasformarlo nella rappresentazione interna (*SequenceModel*).

La figura 5.6, mostra l'insieme dei moduli wrapper.

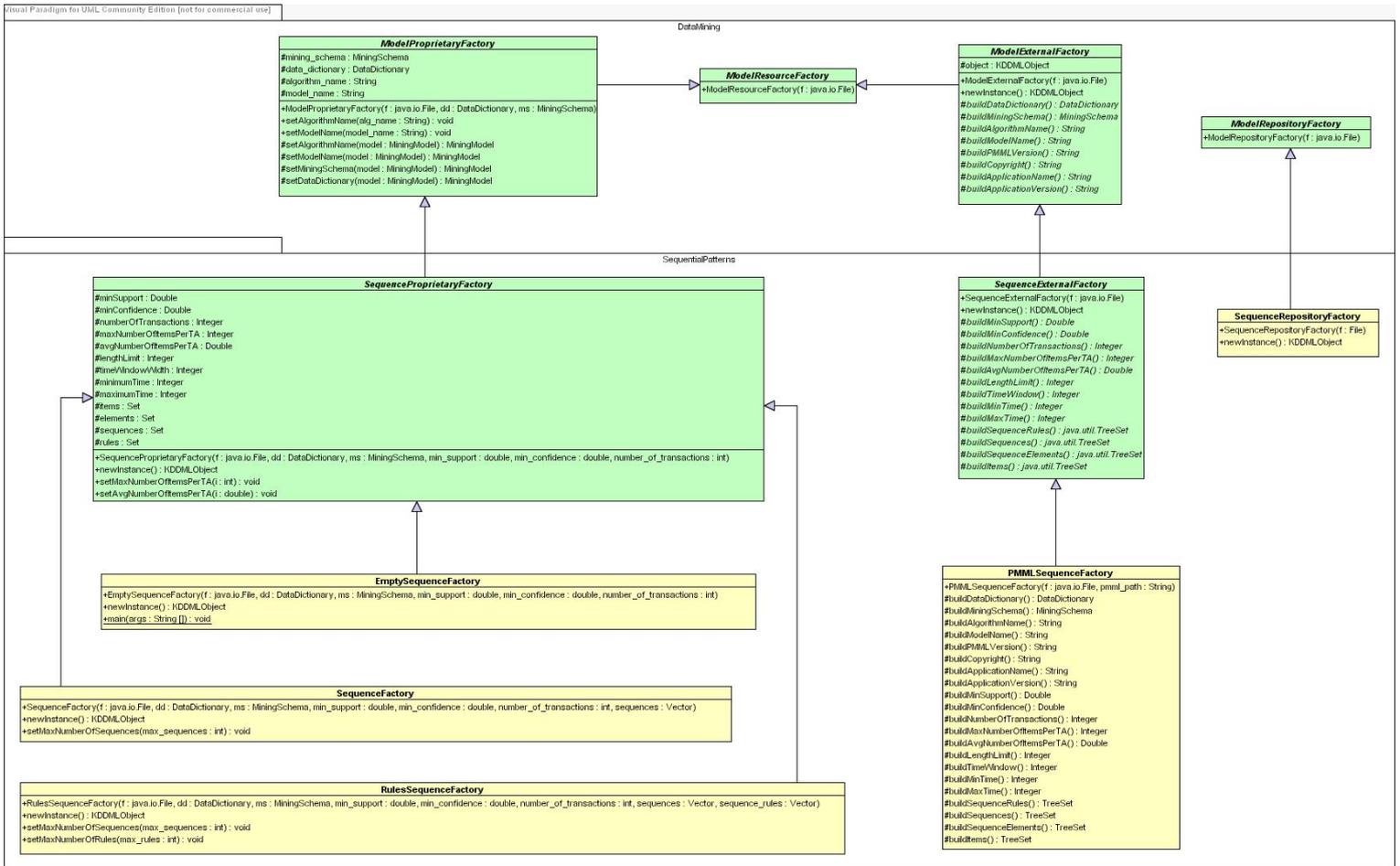


Figura 5.6: Sequence Model Factory

Capitolo 6

Estensione del sistema: Livello Operatori

6.1 Operatori a livello di sistema

A livello di sistema, un operatore `<operator_name>` è implementato come due classi Java:

1. `<operator_name>_Resolver`, e
2. `<operator_name>_Settings`,

che soddisfano rispettivamente, le classi astratte `KDDMLOperatorResolver` e `KDDMLOperatorSettings`, radici di tutti gli operatori.

`KDDMLOperatorSettings` è utilizzata solo per eseguire i controlli sui parametri (attributi) dell'operatore, e richiede l'estensione del seguente metodo:

- `void applySettings(Hashtable attributes)`
esegue un controllo di correttezza sui parametri (attributi) dell'operatore.

`KDDMLOperatorResolver` è l'astrazione principale (un generico operatore), e richiede i seguenti metodi:

- `boolean runtimeCheckNeeded()`
restituisce `true` se il tipo del risultato del metodo `execute()` non è determinato a tempo di compilazione, ma deve essere controllato a run-time;
- `boolean abortIfIsEmpty()`
dice all'interprete di interrompere l'esecuzione della query se il risultato prodotto è vuoto;
- `KDDMLObjectType getParamType(int i)`
restituisce il tipo dell'*i*-esimo argomento dell'operatore;

- `KDDMLObject execute(KDDMLOperatorSettings settings, Vector arguments)`
restituisce il risultato dell'esecuzione dell'operatore sui parametri e gli argomenti dati.

La figura 6.1 mostra la gerarchia degli operatori di data mining per pattern sequenziali.

Nel paragrafo successivo, viene introdotta la rappresentazione nel sistema dell'elemento `CONDITION` definito nel linguaggio. L'elemento `CONDITION`, è implementato a livello *Core*; viene introdotto qui, in quanto utilizzato da alcuni operatori.

Rappresentazione dell'elemento `CONDITION`

Un elemento `CONDITION` [2.1], è una combinazione booleana (`AND`, `OR`, `NOT`) di casi base (elemento `BASE_COND`). definito dal linguaggio per valutare una generica condizione.

Questo elemento, è riconosciuto dall'interprete, e passato all'operatore come un oggetto **Condition**, che ha la seguente interfaccia:

- `boolean evalCond(Instance relational_instance, Instance preprocessing_instance)`
valuta la condizione sull'istanza passata.
- `void checkConditionValidity()`
esegue un controllo di correttezza sulla condizione specificata, ovvero, se l'operazione è supportata, ed i parametri dell'operazione sono corretti

Ogni operatore che utilizza come parametro un oggetto **Condition**, deve estendere la classe **BaseCondition**, che rappresenta l'elemento `BASE_COND` (una condizione primitiva). Questa classe, implementa le funzionalità base per valutare una condizione.

L'estensione della classe **BaseCondition**, richiede l'implementazione dei seguenti metodi:

- `boolean evalSimpleCase(String op, String[] values)`
valuta l'espressione rappresentata; *op* è il tipo di operatore, e *values* sono i parametri dell'operatore
- `void checkTerm1(String op_type, String term_value, boolean is_constant)`
esegue un controllo di correttezza sul primo parametro
- `void checkTerm2(String op_type, String term_value, boolean is_constant)`
esegue un controllo di correttezza sul secondo parametro
- `void checkTerm3(String op_type, String term_value, boolean is_constant)`
esegue un controllo di correttezza sul terzo parametro
- `byte getTermCardinality(String op_type)`
restituisce la cardinalità di *op_type*

6.2 SEQUENCE_MINER

La classe `SEQUENCE_MINER_SETTINGS` definisce dei parametri generali (gli attributi del modello definiti in 5.2.1) per gli algoritmi di estrazione di pattern sequenziali, come:

- minimo supporto
- massimo numero di sequenze
- minima confidenza
- massimo numero di regole
- massima lunghezza della sequenza
- dimensione del time window
- tempo minimo/massimo

Ogni algoritmo dovrà poi estendere questa classe per definire i parametri supportati.

La classe `SEQUENCE_MINER_RESOLVER` implementa il pattern del metodo `execute()` per gli algoritmi di data mining, riportato in figura 6.2

L'interfaccia `SequenceAlgorithmResolverTask` definisce le funzionalità comuni a tutti gli algoritmi per pattern sequenziali, ed è richiesta dall'operatore per implementare il pattern del metodo `execute()`.

6.2.1 PREFIX_SPAN

La classe `PREFIX_SPAN_SETTINGS` definisce i parametri supportati dall'algoritmo `prefix_span`, ed esegue i controlli sui parametri immessi. I parametri supportati sono:

min_support: minimo support di un pattern sequenziale, $(0,1]$;

max_numer_of_sequences: massimo numero di sequenze calcolate¹ $(0, \infty]$.

La classe `PREFIX_SPAN_RESOLVER` implementa, come richiesto, l'interfaccia `SequenceAlgorithmResolverTask`:

```
- void readSettings(KDDMLOperatorSettings settings)
    legge i parametri dell'algoritmo;
```

¹Nell'implementazione attuale, l'algoritmo `prefix_span` non supporta questo parametro. I pattern in eccesso, vengono rimossi dopo essere stati estratti.

```

public kddml.Core.KDDMLObject execute(KDDMLOperatorSettings settings,
                                       Vector arguments)
    throws KDDMLOperatorException, KDDMLCoreException {

    String alg_name = ((DMAgorithmSettings) settings).algorithm_name;
    SequenceAlgorithmFactory factory =
        new SequenceAlgorithmFactory(alg_name);
    factory.initialize();
    if (!factory.isSupported()) {
        throw new KDDMLOperatorException
            ("Algorithm < " + alg_name + " > not supported!", null,
             KDDMLOperatorErrorCodes.UNSUPPORTED_ALGORITHM);
    }

    SequenceAlgorithmResolverTask as = (SequenceAlgorithmResolverTask)
        factory.newResolverInstance();

    try {

        // estrae dagli argomenti la tabella
        InternalTableManager t = (InternalTableManager) arguments.get(0);

        // richiama i metodi della sottoclasse che contiene
        // l'implementazione dell'algoritmo
        as.readSettings(((DMAgorithmSettings) settings).alg_settings);
        Object obj1 = as.inputFormatting(t);
        Object obj2 = as.execute(obj1);
        KDDMLObject ris = as.outputFormatting(obj2);
        super.removeTemporaryFiles(as.getTmpFiles());
        return ris;

    }
    catch(KDDMLCoreException e1) {
        super.removeTemporaryFiles(as.getTmpFiles());
        throw e1;
    }
    catch(KDDMLOperatorException e2) {
        super.removeTemporaryFiles(as.getTmpFiles());
        throw e2;
    }
    catch(Exception e) {
        super.removeTemporaryFiles(as.getTmpFiles());
        throw new KDDMLCastingException(KDDMLObjectType.INTERNAL_TABLE);
    }
}

```

Figura 6.2: implementazione del pattern del metodo execute()

- `Object inputFormatting(InternalTableManager instances)`
formatta i dati contenuti nella tabella nel formato utilizzato dall'algoritmo `prefix_span`. La tabella in input deve essere nel formato *timestamp*. Restituisce una tabella hash contenente tutte le coppie $(item, id)$ generate durante la trasformazione dell'input.
- `Object execute(Object input_formatting_result)`
invoca l'algoritmo `prefix_span` con i parametri ottenuti in `readSettings()`. Restituisce la tabella hash passata come argomento.
- `SequenceModel outputFormatting(Object execute_result)`
trasforma il risultato prodotto dall'algoritmo, utilizzando la tabella hash passata in input per convertire gli id nel risultato negli item originali. Restituisce un `SequenceModel` contenente i pattern estratti.

Formato e trasformazione dell'input

Lo scopo di questa operazione, è di adattare il formato di rappresentazione dei dati adottato dal sistema, in quello utilizzato dall'algoritmo `prefix_span`.

L'input utilizzato dall'algoritmo, consiste di un file testo dove ogni riga rappresenta una transazione utente. Ogni riga ha il seguente formato:

`<user_id> <id_1> ... <id_n>`,

dove *user_id* è un intero che identifica un generico utente, *id_i* è un intero che identifica un item all'interno della transazione. Gli items in una transazione sono ordinati in modo crescente, $\forall i, id_i < id_{(i+1)}$. Per ogni *user_id*, sono ammesse più transazioni; l'ordine in cui si presentano stabilisce l'ordine in cui sono avvenute. Le transazioni, inoltre, devono essere raggruppate per *user_id*; diversamente, sono considerate transazioni di utenti distinti.

Ogni riga del file contiene gli items di una transazione con timestamp (`TimestampedTransaction`) della tabella in input. Tutti gli items vengono rappresentati numericamente, ed il valore originale memorizzato in una tabella hash ($item_id \rightarrow item_value$). Il primo valore di ogni riga rappresenta l'identificatore della transazione (`DataSequence`).

Ad esempio, data la tabella 6.1, l'operazione di formattazione, produce l'input da passare all'algoritmo e la tabella hash contenente tutte le associazioni ($item_id \rightarrow item_value$), come mostrato nella tabella 6.2).

Dovendo scandire l'intero input, vengono calcolate alcune informazioni sulla tabella in input, quali:

- il numero di transazioni (data-sequence),
- il massimo numero di items per transazione (data-sequence),
- il numero medio di items per transazione (data-sequence),

utilizzate per costruire il modello estratto. Inoltre, vengono costruiti il `DataDictionary` - la descrizione degli attributi nella tabella - ed il `MiningSchema` - gli attributi utilizzati nell'estrazione.

transaction	timestamp	event
1	10	pasta
1	10	pane
1	10	vino
1	15	vino
2	10	pasta
2	15	latte
2	20	pane

Tabella 6.1: tabella timestamp

user_id	items_id	item_id	item_value
1	1 2 3	1	pasta
1	3	2	pane
2	1	3	vino
2	4	4	latte
2	2		

Tabella 6.2: input prodotto e relativa tabella hash

Formato e trasformazione dell'output

Quest'operazione produce il risultato inverso, rispetto a quella in input, cioè costruisce le sequenze partendo dal risultato prodotto dall'algoritmo.

L'output restituito dall'algoritmo, consiste di un file testo dove la prima riga contiene gli items frequenti nel seguente formato:

`<id_1> ... <id_n>`,

dove id_i è un intero che identifica un item.

Le righe successive contengono i (sub-)pattern sequenziali estratti; per ogni pattern, la riga successiva, contiene l'insieme dei sub-pattern.

Un pattern, ha il seguente formato:

`<itemset_1> ... <itemset_n>;<percentage_support>%`,

dove $itemset_i$ è l' i -esimo itemset che compone il pattern; un itemset ha il seguente formato:

- `<id_1>` se l'itemset è composto da un solo item,
- `(<id_1>, ..., <id_n>)` altrimenti.

L'insieme dei sub-pattern, ha il seguente formato:

`<pattern_1><pattern_2>...<pattern_n>`,

dove $pattern_i$ ha il formato descritto precedentemente.

L'insieme dei (sub-)pattern viene rappresentato tramite l'oggetto **Sequence**; gli id degli items che compongono i pattern, vengono sostituiti con il loro valore originale, tramite la tabella hash passata come argomento.

Nella tabella 6.3, viene mostrato un esempio di output dell'algorithmo.

1 5 7
1;100.0%
(1,5);66.0%
1;100.0%5;100.0%
(1,7);66.0%
1;100.0%7;66.0%
(1,7) 7;66.0%
(1,7);66.0%1 7;66.0%1;100.0%7;66.0 %7 7;66.0%
1 7;66.0%
1;100.0%7;66.0%
5;100.0%
5 7;66.0%
5;100.0%7;66.0%
7;66.0%
7 7;66.0%
7;66.0%

Tabella 6.3: output algoritmo

6.3 SEQUENCE_SATISFY

La classe **SEQUENCE_SATISFY_RESOLVER**, implementa l'operatore **SEQUENCE_SATISFY**.

Ogni **DataSequence** caricata dalla tabella, viene mantenuta nella tabella in output, solo se supporta almeno una sequenza nel modello.

```

1 procedure satisfy(table, sequences):
2   output_table := {};
3   for all data_sequence in table do
4     for all sequence in sequences do
5       if data_sequence.support(sequence) then
6         output_table.add(data_sequence);
7         break;
8       end if
9     end for
10  end for
11  return output_table;
```

6.4 SEQUENCE_EXCEPTION

La classe `SEQUENCE_EXCEPTION_RESOLVER`, implementa l'operatore `SEQUENCE_EXCEPTION`.

La classe `SEQUENCE_EXCEPTION_RESOLVER`, è implementata come estensione della classe `SEQUENCE_SATISFY_RESOLVER`. Per ogni `DataSequence` per cui l'operatore “*satisfy*” restituisce un booleano b , che indica se mantenere i dati o meno, l'operatore “*exception*” restituisce $!b$.

```
1 procedure exception(table, sequences):
2   output_table := {};
3   for all data_sequence in table do
4     add := true;
5     for all sequence in sequences do
6       if data_sequence.support(sequence) then
7         add := false;
8         break;
9       end if
10    end for
11    if add then
12      output_table.add(trans);
13    end if
14  end for
15  return output_table;
```

6.5 SEQUENCE_MAXIMAL_FILTER

La classe `SEQUENCE_FILTER_RESOLVER` implementa l'algoritmo di filtraggio dei pattern non massimali.

La struttura *maximal_sequences*, che contiene le sequenze massimali, mantiene un ordine tra le sequenze in base alla lunghezza ed il supporto; infatti, le sequenze con lunghezza maggiore e supporto minore, hanno più probabilità di essere quelle massimali. In questo modo, solo le sequenze massimali devono scandire l'intera struttura (per scoprirne di esserlo). Considerando, inoltre, che il numero di sequenze massimali è strettamente minore rispetto a quelle in input, l'approccio seguito, da ottimi risultati in termini di efficienza.

```
1 procedure maximal(sequences):
2   maximal_sequences := {};
3   for all seq in sequences do
4     add := false;
5     for all max_seq in maximal_sequences do
6       if max_seq.contains(seq) then
7         add := false;
8         break;
9     else
```

```

10         if seq.contains(max_seq) then
11             maximal_sequences.remove(max_seq);
12         end if
13     end if
14 end for
15 if add then
16     maximal_sequences.add(seq);
17 end if
18 end for
19 return maximal_sequences;

```

6.6 SEQUENCE_RULE

La classe **SEQUENCE_RULE_SETTINGS** definisce i parametri supportati dall'operatore, ed i controlli sui parametri immessi. I parametri supportati sono:

min_confidence minima confidenza di una regola, (0,1];

max_numer_of_rules massimo numero di regole calcolate (0, ∞].

La classe **SEQUENCE_RULE_RESOLVER** implementa l'algoritmo di estrazione delle regole. Di seguito vengono riportati i passi principali dell'algoritmo.

```

1  procedure compute_rules(sequences):
2  rules := {};
3  hash_table := {};
4  for all seq in sequences do
5      hash_table.add(seq, seq.support);
6  end for
7
8  for all seq in sequences do
9      for i := 1 to seq.length do
10         rule.antecedent := seq.sub_sequence(0, i);
11         rule.consequent := seq.sub_sequence(i, seq.length);
12         rule.support := seq.support;
13         rule.antecedent.support := hash_table.get(rule.antecedent);
14         rule.consequent.support := hash_table.get(rule.consequent);
15         if rule.antecedent.support & rule.consequent.support then
16             rule.confidence := rule.support / rule.antecedent.support;
17             if rule.confidence ≥ min_confidence then
18                 rules.add(rule);
19             end if
20         end if
21     end for
22 end for
23 return rules;

```

6.7 SEQUENCE_FILTER

Questo operatore, utilizza l'elemento `CONDITION` [2.1] definito dal linguaggio per valutare una generica condizione.

La classe `SEQUENCE_FILTER_CONDITION` implementa la specifica dell'elemento `CONDITION` richiesta dall'operatore.

Le sequenze nel modello, sono trasformate in *istanze* (come definito in 5.2.1), e valutate dalla condizione.

```
1 procedure filter(sequences , condition):
2   for all seq in sequences do
3     if condition.eval(seq) then
4       sequences.remove(seq);
5     end if
6   end for
7   return sequences;
```

6.8 SEQUENCE_AGGREGATE_FILTER

Questo operatore, utilizza l'elemento `CONDITION` [2.1] definito dal linguaggio per valutare una generica condizione.

La classe `SEQUENCE_AGGREGATE_FILTER_CONDITION` implementa la specifica dell'elemento `CONDITION` richiesta dall'operatore.

Per ogni attributo su cui deve essere calcolata una funzione di aggregazione, ne viene calcolato il valore per ogni item nella data-sequence che supporta un item nella sequenza (per una definizione più formale si veda la specifica dell'operatore nel capitolo 4).

L'insieme degli attributi e dei rispettivi valori, viene trasformato in una *istanza* nel seguente modo:

ogni attributo dell'istanza, corrisponderà ad un attributo su cui deve essere calcolata una funzione di aggregazione, mentre il valore associato ad ogni attributo, sarà una stringa in formato csv, che rappresenta l'insieme dei valori dell'attributo, calcolati come appena descritto.

L'istanza ottenuta, viene valutata dalla condizione.

```
1 procedure aggregate_filter(sequences , table , condition):
2   for all data_sequence in table do
3     for all seq in sequences do
4       trans := seq.getSupportingTransactions(data_sequence);
5       if trans && condition.eval(trans) then
6         seq.num_occurrences++;
7       end if
8     end for
9   end for
10
```

```

11 min_occurrences = sequences.min_support * table.num_data_sequence;
12 for all seq in sequences do
13     if seq.num_occurrences < min_occurrences then
14         sequences.remove(seq);
15     end if
16 end for
17 return sequences;

```

6.9 SEQUENCE_TIMESTAMP_FILTER

La classe **SEQUENCE_TIMESTAMP_FILTER_SETTINGS** definisce i parametri supportati dall'operatore, ed i controlli sui parametri immessi. I parametri supportati sono:

constr_type, il vincolo da applicare: *duration* o *gap*;

interval_closure, il tipo di intervallo: *open_open*, *open_closed*, *closed_closed* o *closed_open*;

lower_bound, il limite inferiore dell'intervallo;

upper_bound, il limite superiore.

La classe **SEQUENCE_TIMESTAMP_FILTER_RESOLVER** implementa l'algoritmo di filtraggio basato sul timestamp dei dati in input.

I passi principali dell'algoritmo, sono riportati di seguito.

```

1 procedure timestamp_filter(sequences, table, interval):
2 for all data_sequence in table do
3     for all seq in sequences do
4         trans := seq.getSupportingTransactions(data_sequence);
5         if trans && evalConstraint(trans, interval) then
6             seq.num_occurrences++;
7         end if
8     end for
9 end for
10
11 min_occurrences = sequences.min_support * table.num_data_sequence;
12 for all seq in sequences do
13     if seq.num_occurrences < min_occurrences then
14         sequences.remove(seq);
15     end if
16 end for
17 return sequences;

```

In base al vincolo specificato, viene eseguita una delle seguenti procedure.

```
1 procedure evalDurationConstraint(transactions , interval):
2 duration := transactions.last.time - transactions.first.time;
3 return interval.contain(duration);

1 procedure evalGapConstraint(transactions , interval):
2 for i := 1 to trans.length do
3     gap := transactions[i].time - transactions[i-1];
4     if !interval.contain(gap) then
5         return false;
6     end if
7 end for
8 return true;
```

Capitolo 7

Conclusioni

Durante il tirocinio sono state affrontate diverse problematiche, quali l'estrazione di conoscenza, la valutazione della conoscenza estratta, la sua rappresentazione attraverso un linguaggio standard, i requisiti di un sistema e di un linguaggio di supporto per l'intero processo KDD.

Le difficoltà incontrate non sono state poche, vista la complessità del problema affrontato e la mancanza delle conoscenze di base per poterlo affrontare.

Pur essendo l'estrazione di pattern sequenziali ben definita, esistono in letteratura diversi problemi relativi alla scoperta di pattern da dati sequenziali che, nonostante rispondano a problematiche differenti ed utilizzino approcci diversi, presentano molte analogie.

La *scoperta di episodi frequenti*, introdotto da Mannila e al. [8], consiste nel trovare episodi (che verificano una determinata soglia di frequenza) in una lunga sequenza di eventi (solamente), dove un episodio è definito come un insieme di eventi che occorrono con un ordine definito parzialmente e all'interno di un determinato limite di tempo (time-window).

La *formulazione universale dei pattern sequenziali* [6] consiste in una generalizzazione dei problemi introdotti, rispettivamente, da Agrawal e Srikant e da Mannila e al.

I pattern sequenziali estratti, inoltre, possono essere classificati in tre principali categorie [3]:

- pattern sequenziali continui. I dati che compongono un pattern, appaiono necessariamente in ordine consecutivo, anche nei dati in input.
- pattern sequenziali discontinui. I dati che compongono un pattern, appaiono (non necessariamente) in ordine non consecutivo nei dati in input.
- pattern sequenziali ibridi. Sono pattern formati da sotto-pattern sia continui che discontinui.

Tra queste categorie, l'estrazione di pattern sequenziali discontinui (il problema introdotto da Agrawal e Srikant) è la più popolare ed è stata intensivamente studiata vista la numerosità dei domini di applicazione.

Una comprensione completa del problema, richiederebbe uno studio molto più accurato, che, come ci si può immaginare, non rientra negli scopi di questo tirocinio.

Per quanto riguarda il sistema, una volta studiata l'architettura, l'estensione dei vari livelli, è stata, in generale, la parte meno difficile, grazie anche all'alta estensibilità del sistema. Tuttavia, per poter procedere all'estensione del sistema, è stato necessario apprendere i linguaggi utilizzati e familiarizzare con alcuni tool.

La comprensione del linguaggio XML, ed in particolare del DTD, è stata di fondamentale importanza, se si considera che il KDDML si basa completamente su XML, sia per la specifica del linguaggio che per la rappresentazione di query, dati e modelli.

In particolare, i modelli KDDML sono rappresentati come estensione dei modelli PMML. Per poter procedere nell'estensione del sistema, è stato necessario comprendere la specifica del modello PMML, cosa non molto facile, in quanto:

1. essendo uno standard, il PMML ha un alto livello di astrazione;
2. relativamente al modello specifico da rappresentare (`SequenceModel`), non esiste, a parte una descrizione generale, una documentazione completa o degli esempi guida.

Inoltre, il KDDML permette di trasformare i modelli estratti in documenti HTML, per renderne la visualizzazione più "user friendly". Per poter trasformare i modelli estratti in documenti HTML, è stato necessario apprendere le basi del linguaggio XSLT per poter definire un foglio di stile XSL, che descrive le regole per la trasformazione.

Uno strumento essenziale, sia per la comprensione dei vari moduli dell'architettura che per la documentazione del lavoro svolto, è stato *Visual Paradigm for UML* (VP-UML) [1], uno strumento CASE per UML. L'utilizzo dei diagrammi UML esistenti e la creazione, quando necessario, di diagrammi più dettagliati ha permesso una rapida comprensione dei moduli dell'architettura. Inoltre, l'utilizzo dei diagrammi UML per la documentazione del modello implementato, ha facilitato la discussione delle scelte progettuali.

L'obiettivo inizialmente stabilito, era quello di fornire un supporto per i pattern sequenziali ed alcuni operatori base da applicare ai pattern estratti. Il lavoro svolto ha permesso al sistema di supportare ampiamente il processo di estrazione e valutazione di pattern sequenziali. In particolare, nel livello *Core*, le nuove funzionalità fornite sono:

- la gestione di un nuovo formato dei dati, il formato timestmap;
- l'accesso e la gestione di un modello per i pattern sequenziali.

Nel livello *Algorithm e Operators*:

- l'estrazione di pattern sequenziali;
- il filtraggio dei pattern estratti in base a diverse proprietà, sia del pattern che dei dati da cui è stato estratto;
- il calcolo di regole sequenziali.

Tuttavia, per quanto riguarda i pattern sequenziali, esistono diversi miglioramenti che possono essere apportati al sistema, quali:

- fornire un supporto completo del PMML;
- integrare un algoritmo più efficiente (*constraint-based*);
- aggiungere nuovi operatori. Ad esempio, la classificazione di nuovi pattern [4].

Bibliografia

- [1] *Visual Paradigm for UML (VP-UML)*, 2004, <http://www.visual-paradigm.com>.
- [2] Rakesh Agrawal and Ramakrishnan Srikant, *Mining sequential patterns*, Eleventh International Conference on Data Engineering, 1995.
- [3] Yen-Liang Chen, Shih-Sheng Chen, and Ping-Yu Hsu, *Mining hybrid sequential patterns and sequential rules*, 2002.
- [4] Pedro Gabriel Ferreira and Paulo J. Azevedo, *Sequence similarity classification through relevant sequence mining*.
- [5] Daniela Iozzia, *Studio, progettazione e implementazione di un algoritmo per il calcolo di sequential pattern*, 2003, Tesi di Laurea, Dipartimento di Informatica, Univ. di Pisa.
- [6] Mahesh Joshi, George Karypis, and Vipin Kumar, *A universal formulation of sequential patterns*, Tech. report, University of Minnesota, Minneapolis, 1999.
- [7] Heikki Mannila and Hannu Toivonen, *Discovering generalized episodes using minimal occurrences*, Knowledge Discovery and Data Mining, 1996.
- [8] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo, *Discovering frequent episodes in sequences*, First International Conference on Knowledge Discovery and Data Mining, 1995.
- [9] ———, *Discovery of frequent episodes in event sequences*, Data Mining and Knowledge Discovery, 1997.
- [10] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu, *Prefixspan: Mining sequential patterns efficiently by prefix projected pattern growth*, 2001.
- [11] Jian Pei, Jiawei Han, and Wei Wang, *Mining sequential patterns with constraints in large databases*, International Conference on Information and Knowledge Management, 2002.
- [12] Ramakrishnan Srikant and Rakesh Agrawal, *Mining sequential patterns: Generalizations and performance improvements*, Tech. report, IBM Research Division, 1996.

- [13] The Data Mining Group, *Predictive Model Markup Language (PMML)*, Version 2.1, 2003, <http://www.dmg.org>.
- [14] W3C World Wide Web Consortium, *Extensible Stylesheet Language Transformations (XSLT) 1.0*, W3C Recommendation, 1999, <http://www.w3.org/TR/xslt>.
- [15] ———, *Extensible Markup Language (XML) 1.0 (second edition)*, W3C Recommendation, 2000, <http://www.w3.org/TR/REC-xml>.
- [16] ———, *Extensible Stylesheet Language (XSL)*, W3C Proposed Recommendation, 2001, <http://www.w3.org/TR/xsl>.
- [17] ———, *XML Schema, Parts 0, 1, and 2*, W3C Recommendation, 2001, <http://www.w3.org/TR/xmlschema-0>, -1, and -2.
- [18] Qiankun Zhao and Sourav S. Bhowmick, *Sequential pattern mining: A survey*, Tech. report, CAIS, Nanyang Technological University, Singapore, 2003.