



Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea Specialistica in Informatica

Tesi di laurea specialistica

**KDDML: Estensione alla
fase di Preprocessing**

Candidato
Sandra Zimei

Relatore
Prof. Franco Turini

Controrelatore
Prof.ssa Alessandra Di Pierro

Anno Accademico 2003-2004

A mamma, papà, Orlando e Alessio
per tutto l'amore che mi danno.

Introduzione

Negli ultimi anni il processo di estrazione di nuova conoscenza su grosse quantità di dati, noto come processo KDD (Knowledge Discovery in Database), ha conquistato campi di applicazione svariati e innovativi. La motivazione principale sta nel vertiginoso aumento dei dati disponibili in molti settori dovuto, soprattutto, alla diffusione della modalità di comunicazione via web. Il processo KDD ha motivo d'esistere proprio in condizioni di disponibilità di una quantità considerevole di dati che non sarebbero utili ai fini dell'estrazione di nuove informazioni non banali, potenzialmente utilizzabili e comprensibili, se non accuratamente preparati. L'intero processo KDD è composto da diverse fasi iterative che in letteratura sono state individuate con nomi che variano a seconda delle interpretazioni, ma che sempre riferiscono tre categorie di elaborazione dei dati: preprocessing, data mining, postprocessing. Nonostante la fase più rappresentativa sia quella di data mining, il cui nome spesso sostituisce quello dell'intero processo, in termini di tempo la fase più importante è quella di preprocessing. Questa fase, infatti, comporta un dispendio di risorse molto elevato soprattutto a causa delle modalità di esecuzione che fanno pesantemente riferimento a conoscenze del dominio. L'importanza aumenta se si tiene conto che i risultati ottenuti da questa fase saranno vincolanti per le fasi successive al punto che un preprocessing non accurato porta ad avere, al termine dell'intero processo, dei risultati

inconsistenti o poco realistici. In questi ultimi anni, presso il Dipartimento di Informatica dell'Università di Pisa è stato sviluppato un ambiente generale di sviluppo per applicazioni di estrazione di conoscenza dai dati. Tale sistema, denominato KDDML (KDD Markup Language), è nato con l'obiettivo principale di integrare le varie fasi del processo KDD. Si è riconosciuta innanzitutto la necessità di rappresentare, in modo uniforme, i dati in ingresso, i meta-dati, i modelli di conoscenza, e gli stessi problemi di data mining. Questo requisito, indispensabile per fornire l'interoperabilità tra le diverse fasi del processo, così come tra i diversi modelli di conoscenza, è stato soddisfatto attraverso l'utilizzo di un linguaggio standard per la rappresentazione e lo scambio dei dati: il linguaggio di mark-up XML. Esso costituisce la base per la realizzazione del linguaggio di mark-up KDDML del sistema, il quale è interamente implementato in Java. In un secondo momento è stato definito e integrato nel sistema un nuovo linguaggio di interrogazione, che rendesse l'utilizzo dell'ambiente KDDML più semplice [22]. Tale linguaggio, denominato Mining Query language (MQL) [23] si pone ad un livello di astrazione superiore, conservando la flessibilità, l'estendibilità e l'espressività tipiche del linguaggio KDDML. Tramite esso è possibile specificare un problema di estrazione di conoscenza come un processo di interrogazione sui dati. Il linguaggio, infatti, permette di combinare dati e modelli di conoscenza (regole di associazione, alberi di classificazione, clusters) attraverso degli operatori secondo uno stile algebrico. Il risultato ottenuto con KDDML-MQL [22] è stato quindi un ambiente in cui l'utilizzatore interagisce inserendo queries che rispettano la grammatica definita in MQL, mentre rimane compito del sistema provvedere all'interfacciamento con KDDML per la loro esecuzione vera e propria. Il query language MQL è stato successivamente completato con nuovi costrutti che modellano funzionalità a supporto della fase di preproces-

sing (DPL [9]). Il linguaggio DPL è costituito principalmente da costrutti che rappresentano trasformazioni di “tabelle estese” a livello di istanza e a livello di schema. Le tabelle estese sono tabelle relazionali in cui si individuano due componenti principali: quella relativa ai dati veri e propri e quella, di pari dimensioni, contenente metainformazioni riguardo ciascun dato in posizione corrispondente. La parte delle metainformazioni è dedicata all’inserimento implicito o esplicito di codici di marcatura per mantenere traccia di operazioni di preprocessing che operano a livello di istanza. Il preprocessing per sua natura è, infatti, “*user guided*” nel senso che le decisioni su quale tipo di operazioni applicare e con quali parametri, vengono prese tenendo conto del tipo di conoscenza che si vuole arrivare ad estrarre e ponderando le scelte sulla base della conoscenza del dominio. Tenendo conto di questa considerazione le metainformazioni inserite in modo implicito risolvono effetti indesiderati causati da scelte fatte sulla base di dati non omogenei ottenuti da trasformazioni di preprocessing. L’inserimento esplicito di codici di marcatura, invece, serve all’utente che guida il processo di estrazione di conoscenza per individuare tipologie di dati sospetti. Partendo dal sistema KDDML-MQL è nata quindi la necessità di integrarlo con il DPL in modo completo. Questo lavoro di tesi ha come obiettivo quello di rendere il sistema abile a modellare e risolvere le funzionalità principali della fase di preprocessing del processo KDD. A tale scopo il lavoro mira ad agire sulle componenti principali del sistema: il compilatore, il linguaggio KDDML, l’interprete del sistema e il suo supporto a run-time. Il compilatore interposto tra query language e linguaggio KDDML è realizzato sfruttando il *framework* ANTLR. Con il DPL nasce l’esigenza di integrazioni nella grammatica del linguaggio sorgente per la generazione di alberi sintattici. A fronte di tali nuovi alberi sintattici generabili, va esteso il modulo del compilatore sia per l’analisi statica che per

la generazione di codice in linguaggio destinazione. La generazione di codice fa riferimento al linguaggio KDDML nel quale bisogna inserire nuovi tipi validi per il sistema. Le definizioni della grammatica del linguaggio KDDML sono espresse mediante DTD (*Document Type Definition*) che rappresentano lo schema logico dei documenti .XML contenenti dati eseguibili dal sistema. L'estensione del linguaggio KDDML consiste, quindi, nell'integrazione delle DTD per modellare le "tabelle estese" e le loro trasformazioni previste dal DPL. Ciascuna query passata contenente operazioni di preprocessing deve essere sottoposta a *parsing* per verificarne la validità e poi usarla per creare e istanziare una classe che realizza l'interprete. L'estensione del supporto a run-time costituisce sicuramente il lavoro più impegnativo. Essa riguarda l'introduzione da una parte di nuove strutture dati necessarie per le tabelle estese, e dall'altra di nuovi operatori. Gli operatori in particolare devono essere realizzati tenendo conto di una caratteristica che è la principale nel processo KDD: l'elevata quantità di dati. Proprio per questo, quando possibile i dati da manipolare vanno trattati suddividendoli in blocchi evitando di sovraccaricare la memoria. Già in fase di progettazione della parte di estensione del sistema le operazioni devono essere modellate in classi, tenendo conto sia della possibilità di soddisfare questo requisito di "non sovraccaricamento" della memoria, sia della necessità di rispettare la natura del tipo di trasformazione che ciascun operatore rappresenta (a livello di istanza e a livello di schema). La realizzazione di tutte queste estensioni porta alla costituzione di un ambiente che funge da supporto per la specifica e l'esecuzione di ogni fase dei processi di KDD mediante l'uso di query ad alto livello che meglio si adattano all'uso che ne fa l'utente.

Organizzazione della tesi

- Il capitolo 1 è dedicato alla descrizione del processo KDD nei suoi obiettivi e nei mezzi, distinguendo questi ultimi in base alla fase di appartenenza di ciascuna tecnica.
- Il capitolo 2 presenta una rassegna di tutte le tecniche di preprocessing individuate in letteratura atte a risolvere le problematiche di “qualità dei dati”.
- Nel capitolo 3 viene esposto il sistema KDDML-MQL nelle funzionalità di cui disponeva precedentemente a questo lavoro di tesi.
- Nel capitolo 4 viene presentata l’estensione DPL al query language MQL e le carenze del sistema KDDML-MQL in vista della possibilità di utilizzare i nuovi costrutti di preprocessing proposti per la sottomissione di query al sistema.
- Il capitolo 5 propone le estensioni apportate al sistema KDDML a livello più basso. Vengono definiti i nuovi tipi del linguaggio KDDML e vengono descritte modifiche apportate al supporto a run-time; in particolare sono illustrate le nuove strutture dati introdotte per supplire alle necessità di manipolazione delle tabelle estese, e gli operatori implementati per la risoluzione e la generazione dei risultati delle nuove tipologie di query.
- Il capitolo 6 mostra le modifiche apportate al compilatore per permettere il riconoscimento delle query DPL come tipi validi del linguaggio KDDML.
- Nel capitolo 7 viene riportato un caso d’uso del sistema esteso al preprocessing per la generazione di un dataset consolidato.

- Un capitolo finale conclude la tesi, proponendo ulteriori possibili miglioramenti al sistema KDDML.

Indice

Introduzione	iii
1 Il Processo KDD	1
1.1 Data Cleaning	1
1.2 Data Integration	3
1.3 Data Transformation	4
1.4 Data Reduction	4
1.5 Data Mining	5
1.6 Interpretazione e Valutazione	6
2 Tecniche di Preprocessing	7
2.1 Tecniche di Preprocessing dipendenti dal dominio	9
2.1.1 Problemi dipendenti dal dominio in caso di singola sorgente	10
2.1.2 Problemi dipendenti dal dominio in caso di sorgente multipla	12
2.1.3 Normalizzazione	13
2.1.4 Discretizzazione	14
2.1.5 Campionamento	16
2.1.6 Eliminazione dei Duplicati	21

2.1.7	Trattamento dei Dati Mancanti	24
2.2	Tecniche di Preprocessing indipendenti dal dominio	29
2.3	La Clusterizzazione	30
2.3.1	Clustering per variabili numeriche	32
2.3.2	Clustering per variabili binarie	33
2.3.3	Clustering per variabili categoriche: nominali, ordinali	34
2.3.4	Clustering per variabili di tipo misto	35
2.4	La Classificazione	36
3	Il Sistema KDDML-MQL	39
3.1	XML per la rappresentazione dei documenti.	40
3.2	Gerarchia del Sistema KDDML-MQL	42
3.2.1	Classi della gerarchia del Sistema: PMML_MODEL	42
3.2.2	Classi della gerarchia del Sistema: KDD_QUERY	44
3.2.3	Classi della gerarchia del Sistema: ITEM_HIERARCHY	45
3.2.4	Classi della gerarchia del Sistema: DATA_SOURCES	47
3.3	Architettura del Sistema KDDML-MQL	48
3.3.1	Repository layer	48
3.3.2	Intermediate layer	51
3.3.3	Operators and algorithms layer	52
3.3.4	Interpreter layer	53
3.3.5	GUI layer	53

4	Il Query Language DPL	55
4.1	Sintassi del DPL	55
4.1.1	Esempi guida dell'effetto dei costrutti del DPL	63
4.2	DPL nel Sistema KDDML-MQL	68
5	Estensione del linguaggio KDDML e del supporto a run-time	73
5.1	Estensione del linguaggio KDDML	73
5.1.1	Gli operatori DPL nel linguaggio KDDML	74
5.2	Estensione del supporto a run-time	93
5.2.1	Necessità di nuove strutture dati nel supporto a run-time	93
5.2.2	Introduzione di nuovi operatori e modifica dell'inter- prete nel supporto a run-time	98
5.2.3	Operatori a livello di schema	100
5.2.4	Operatori a livello di istanza indipendenti dai valori assunti da tutte le istanze	108
5.2.5	Operatori a livello di istanza dipendenti dai valori assunti da tutte le istanze	117
5.2.6	Operatori di passaggio di fase nel processo KDD: TABLE2PPTABLE e PTABLE2TABLE	125
6	Estensione del compilatore	129
6.1	Analizzatore lessicale	133
6.1.1	Implementazione del file <code>lexer.g</code>	133
6.1.2	File generati da ANTLR per l'analisi lessicale	139
6.2	Analizzatore sintattico	139
6.2.1	Implementazione del file <code>parser.g</code>	142
6.2.2	File generati da ANTLR per l'analisi sintattica	147

6.3	Analizzatore dei tipi	147
6.3.1	Implementazione del file <code>typeChecker.g</code>	149
6.3.2	File generati da ANTLR per l'analisi dei tipi	151
6.4	Generazione del Codice	152
7	Esempio completo di preprocessing nel sistema KDDML	157
8	Conclusioni	179
	Bibliografia	185

Elenco delle figure

1.1	Processo di KDD	2
2.1	Preprocessing	9
2.2	Classificazione dei problemi sulla qualità dei dati	10
2.3	Es. problemi di singola sorgente a livello di schema	11
2.4	Es. problemi di singola sorgente a livello di istanza	11
2.5	Es. problemi di sorgente multipla a livello di Schema e di Istanza	13
2.6	Metodi di valutazione dell'accuratezza di classificatori.	17
2.7	Esempio dei record contenuti in una finestra di dimensione w , in due passi successivi, durante la fase di individuazione dei duplicati.	23
2.8	Matrice di contingenza di attributi binari di due oggetti i e j	33
2.9	Prima fase di classificazione: Costruzione di un modello.	36
2.10	Seconda fase di classificazione: Uso del modello.	37
3.1	Gerarchia degli oggetti in KDDML-MQL.	42
3.2	Struttura del sistema KDDML-MQL.	50
4.1	Esempio di uniformazione dei formati dell'attributo "Data" mediante operatori del DPL [9].	66
4.2	Esempio di eliminazione di duplicati mediante operatori del DPL [9].	67

4.3	Tabelle Estese nel DPL.	69
4.4	Schematizzazione dei contributi della tesi.	71
5.1	Schema UML delle tabelle estese	94
5.2	Gerarchia delle classi del modulo di PreProcessing	100
5.3	Gerarchia delle classi degli operatori a livello di schema	101
5.4	Gerarchia delle classi degli operatori a livello di istanza, con risultati indipendenti dai valori assunti da tutte le istanze . . .	109
5.5	Gerarchia delle classi degli operatori a livello di istanza, con risultati dipendenti dai valori assunti da tutte le istanze	118
6.1	Compilatore: La struttura standard per la traduzione del lin- guaggio MQL-DPL in KDDML	130
6.2	Esempio di analisi lessicale su una query in DPL.	134
6.3	ANTLR per l'analisi lessicale.	135
6.4	Esempio di analisi sintattica su una query in DPL.	141
6.5	ANTLR per l'analisi sintattica.	142
6.6	ANTLR per l'analisi dei tipi.	151
6.7	Esempio di visita Depth-first dell'albero sintattico nella gene- razione di codice.	155
7.1	Dataset da sottoporre a preprocessing.	158
7.2	Risultati di marcatura dei duplicati sull'attributo SSN.	160
7.3	Metadati della tabella estesa prodotta dall'unificazione delle istanze con valori duplicati per l'attributo SSN.	161
7.4	Prima pagina dei risultati prodotti dall'unificazione delle istan- ze con valori duplicati per l'attributo SSN.	163
7.5	Dati modificati della riscrittura della data di nascita nel for- mato corretto.	167

- 7.6 Porzione della pagina principale dei risultati prodotti dall'operazione di marcatura dei valori negativi dell'attributo "Age". . . 169
- 7.7 Porzione della pagina principale dei risultati prodotti dalla query per il trattamento dell' attributo "Age". 173
- 7.8 Porzione della pagina principale dei risultati prodotti dalla query per il trattamento dell' attributo "Region". 177

Elenco delle tabelle

3.1	Esempio di DTD per l'operatore di merging.	41
3.2	PMML-DTD: Modello di cluster.	44
3.3	KDDML_Object: elementi della KDD_QUERY.	46
3.4	Data_Sources : Esempio di file in formato .ARFF.	48
3.5	Data_Sources : Esempio del file .XML e .DATA per la rappresentazione di una tabella.	49
3.6	KDDML-MQL : estensione del sistema KDDML sulla base degli operatori definiti in MQL.	52
4.1	Esempio: dati da uniformare nel formato mediante operatori del DPL.	63
4.2	Esempio: dati da de-duplicare mediante operatori del DPL.	65
5.1	DTD che descrivono un tipo tabella o tabella estesa in KDDML.	75

Capitolo 1

Il Processo KDD

Si definisce il processo Knowledge Discovery in Databases (KDD) come: [2] “*Il processo non-banale di identificazione di modelli di dati validi, originali, potenzialmente utili ed in fine comprensibili*”. Da questa definizione si deduce la possibilità di individuare misure quantitative per valutare i modelli trovati; concetti come originalità e comprensibilità sono, chiaramente, molto più soggettivi. Un aspetto importante, che combina validità, originalità, utilità e semplicità, è definito come *interestingness*, ed è generalmente assunto come misura generale del valore di un modello. Il processo KDD è un processo *interattivo* tra l’utente, il software utilizzato e gli obiettivi, che devono essere costantemente inquadrati, ed *iterativo* nel senso che in ogni fase del processo si può prevedere sempre un ritorno alle fasi precedenti. Le fasi previste nel processo sono schematizzate in fig. 1.1 [1]:

1.1 Data Cleaning

Le informazioni del mondo reale a disposizione per attuare un processo di KDD si presentano come dati inconsistenti, dati che rappresentano *rumore* o dati incompleti. Il risultato di una estrazione di conoscenza su questi valori

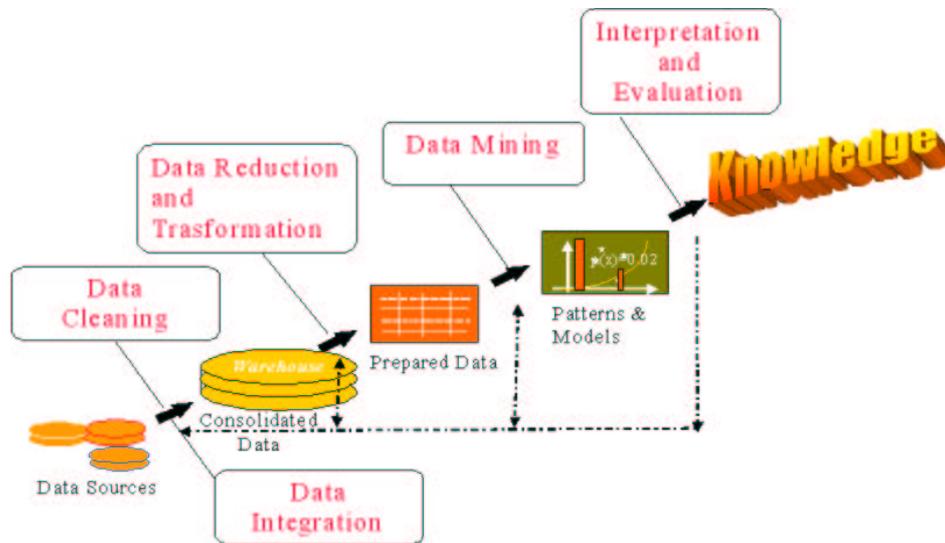


Figura 1.1: Processo di KDD

sarebbe poco realistico. La necessità primaria è quella di formare un data warehouse [3, 4] che risulti abile nella definizione di modelli validi. Il data warehouse richiede e fornisce un supporto esteso della applicazione di tecniche specifiche di *Cleaning*. Nel data warehouse si caricano e continuamente si aggiornano grandi quantità di dati che con elevata probabilità risultano “sporchi”. Nella raccolta di dati, infatti, si possono presentare valori inconsistenti causa la violazione di alcuni vincoli di integrità, talvolta aggravata dalla presenza di sinonimie. Problemi di questo tipo, solitamente causati da errori nella fase di inserimento, possono essere riconosciuti e risolti

- manualmente, riferendosi ad un consulente esterno, oppure
- automaticamente, con l’ausilio di strumenti ad hoc per l’individuazione di quei particolari valori che risultano in contraddizione con alcune dipendenze funzionali sussunte dalla base di conoscenza.

Alcuni dati introdotti per errore o riferiti a casi particolari presentano valori che rispetto ad una particolare misura risultano molto distanti rispetto agli altri, tanto da essere causa di confusione nel processo di KDD. Tali valori formano del *rumore* nella base di conoscenza e devono pertanto essere individuati e scartati o sostituiti (*clustering, sampling, binning*). Ai fini della conoscenza che si vuole prevedere, sono necessarie informazioni che spesso, viste le modalità e i tempi di raccolta dati, possono non essere completamente disponibili oppure presentarsi solo come una delle componenti sintetizzate in un unico valore aggregato. Per procedere correttamente nel processo bisogna quindi introdurre valori stimati e dedurre valori parziali con criteri che mantengono realistico l'insieme di dati.

1.2 Data Integration

Per effettuare una fase di analisi accurata, l'insieme dei dati deve essere coerente e senza informazioni ripetute. I dati devono essere *integrati*

- risolvendo le possibili inconsistenze, dovute a nomi differenti usati per rappresentare la stessa informazione,
- e eliminando le ridondanze.

La molteplicità dei files o dei database da cui si prelevano i dati porta alla presenza di sinonimie fra informazioni che possono essere riconosciute con l'uso dei metadati e risolte rendendo omogenea la nomenclatura a cui fare riferimento. L'elevata quantità di dati raccolti spesso è alimentata dalla presenza di valori che possono essere direttamente derivati da altri. Tra due dati si può misurare un livello di correlazione che, se elevato, è indice del fatto che uno dei due può reputarsi ridondante.

1.3 Data Transformation

Per un'analisi più accurata e per il successo del processo di estrazione di conoscenza, è necessario un passo di trasformazione che consiste principalmente nella normalizzazione di dati e nell'inserimento di dati aggiuntivi utili. Data l'eterogeneità delle fonti, i dati hanno valori inconfondibili. Le cause di incompatibilità tra valori della stessa informazione, sono sia l'influenza personale data dal valutatore che ha raccolto i dati, sia la scala di riferimento con differenza di range di valori a disposizione oppure differenza di granularità. Per ottenere risultati coerenti nell'analisi svolta i dati vanno resi omogenei sottoponendoli ad una fase di normalizzazione secondo criteri che variano a seconda degli scopi del processo. Nell'insieme di informazioni disponibili, ai fini della conoscenza ricercata, ci sono valori che vengono usati ripetutamente nel processo perché sfruttati per ottenere un attributo che è il risultato di una qualche loro combinazione. In particolare possono esserci dati binari spesso testati in *and* ad altri, oppure dati numerici frequentemente utilizzati per essere sottoposti ad operazioni aritmetiche che li combinano tra loro. Conoscere informazione sul ruolo e sull'influenza nel processo di tali valori combinati arricchisce la conoscenza pertanto è utile costruire attributi da aggiungere all'insieme dato.

1.4 Data Reduction

Quando l'insieme dei dati è estremamente numeroso, l'analisi e l'estrazione di conoscenza diventano impraticabili. Sottoponendo i dati ad una riduzione che ne mantiene l'integrità è possibile che il processo migliori nell'efficienza e produca gli stessi risultati che avrebbe ottenuto sull'intero insieme

di valori iniziali. In particolare si può ricorrere a strategie applicabili al momento della realizzazione del data cube:

- applicazione di operazioni di aggregazione e proiezione ai dati che compongono il data cube,
- eliminazione di dimensioni irrilevanti o ridondanti (*forward selection*, *backward elimination* e *decision tree induction* mediante algoritmi quali [5]ID3 e [6]C4.5) ,
- compressioni di dati che permettono comunque successivamente, l'esatta ricostruzione dei dati originari, oppure solo una loro approssimazione,
- riduzione della numerosità dei dati con tecniche
 - *parametriche* per mantenere l'informazione memorizzando solo i parametri da cui dipende (ad es modelli di *regressione* e modelli *log-linear*)
 - *non-parametriche* per memorizzare solo rappresentazioni ridotte dei dati (*istogrammi, clustering, sampling, binning*)),
- discretizzazione e generazione di un concetto di gerarchia per poter astrarre a livelli più generali attributi della base di dati,
- il campionamento.

1.5 Data Mining

Con una certa frequenza, in letteratura, i termini KDD e data mining sono usati come sinonimi; in questa presentazione il Data Mining rappresenta solo una fase particolare in questo processo, quella fase in cui si opera

l'applicazione di algoritmi specifici per estrarre modelli significativi dai dati. Sono gli altri passi del processo KDD, come la preparazione, la selezione, la pulizia dei dati, l'integrazione ed infine la corretta interpretazione dei risultati del mining, che assicurano che l'informazione sia significativa. La cieca applicazione dei metodi di Data Mining può essere una attività pericolosa, che conduce sovente alla scoperta di modelli privi di senso. . In questa fase vengono applicati gli algoritmi per l'esplorazione e lo studio dei dati.

1.6 Interpretazione e Valutazione

Dopo avere estratto conoscenza utilizzando algoritmi appropriati per il Data Mining , si ha a disposizione una serie di modelli che possono costituire un valido supporto alle decisioni. I modelli scoperti vanno interpretati e verificati cioè si cerca di valutare in che misura questi modelli possono essere utili. Da questa fase può nascere la possibilità, alla luce di risultati non perfettamente soddisfacenti, di rimandare il processo ad uno dei passi precedenti per migliorare o correggere i modelli. Al raggiungimento di un buon risultato, questa fase include la generazione di forme di visualizzazione dei modelli estratti comprensibili agli utenti finali. Si passa quindi all'utilizzo della conoscenza acquisita, con l'incorporamento e la fusione di informazioni ed esperienze relative ai sistemi in uso fino a quel momento.

Capitolo 2

Tecniche di Preprocessing

I dati a disposizione dal mondo reale prima di essere sottoposti ad una fase di data mining devono essere elaborati per risolvere

- problemi relativi alla qualità dei dati,
- problemi relativi alla loro rappresentazione ridotta ma significativa nel data warehouse.

Tali problemi si affrontano durante le fasi di:

- cleaning
- integrazione
- trasformazione e
- riduzione

nel complesso queste attività sono note come Preprocessing [1] fig.2.1. Il Preprocessing rappresenta la parte più dispendiosa, in termini di tempi di lavoro, nell'intero processo di KDD. Nel preprocessing non esiste una vera sequenza temporale da seguire per le operazioni che lo compongono, ciascuna

infatti può modificare la rappresentazione dei dati e richiedere di sottoporre nuovamente i dati ad una fase già affrontata , fino ad ottenere un data warehouse consolidato . Le tecniche di preprocessing utilizzate variano a seconda del tipo di problema da risolvere e non a seconda della fase che si esegue [7]. Esse infatti possono essere

- Tecniche che si basano sulla conoscenza del dominio:
 - verifica della consistenza,
 - eliminazione delle ridondanze,
 - omogeneizzazione delle misure fatte su scale differenti per range o per granularità,
 - stima dei dati mancanti;

- Tecniche che sono indipendenti dal dominio:
 - individuazione del rumore,
 - classificazione.

E' importante notare che le tecniche sono generali e “a sé stanti”, la stessa tecnica pertanto può essere usata per affrontare fasi diverse del preprocessing, quello che cambia è solo l'uso che viene fatto dei risultati ottenuti. Le tecniche di individuazione del rumore e di classificazione, ad esempio, possono essere usate sia in fase di cleaning per eliminare i dati irrilevanti ai fini del processo, sia in fase di riduzione della numerosità dei dati per avere una rappresentazione mediante rappresentanti di uno stesso insieme di dati simili.

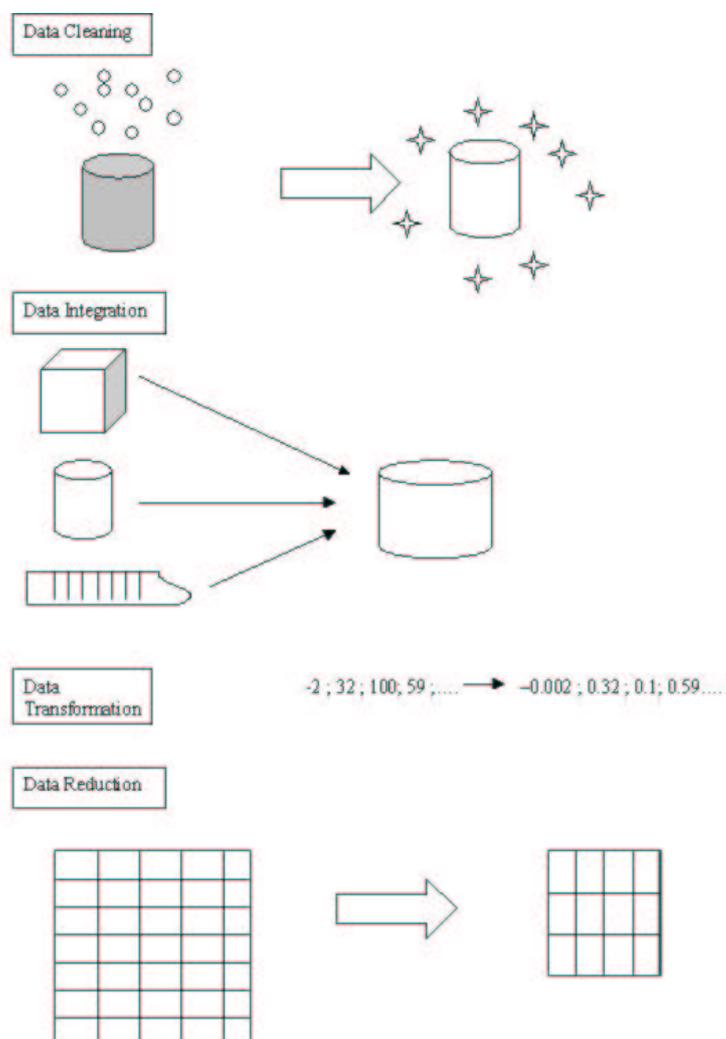


Figura 2.1: Preprocessing

2.1 Tecniche di Preprocessing dipendenti dal dominio

Le tecniche che si basano sulla conoscenza del dominio si preoccupano della qualità dei dati. I problemi di qualità dei dati si distinguono tra problemi derivanti da singola sorgente o derivanti da molteplici sorgenti e problemi a

livello di schema o a livello di istanza[8](vedi fig.2.2) Le tecniche utili per

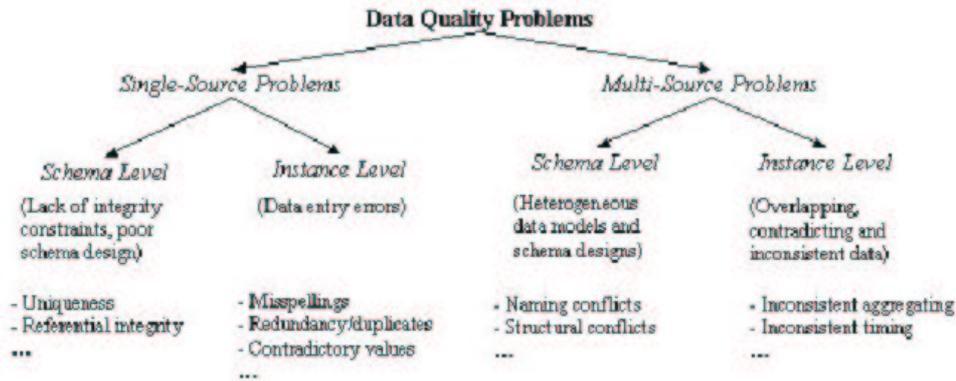


Figura 2.2: Classificazione dei problemi sulla qualità dei dati

garantire la qualità dei dati sono, ad esempio:

- discretizzazione,
- normalizzazione,
- binarizzazione,
- campionamento,
- eliminazione di anomalie e errori,
- eliminazione dei duplicati (*merge/purge problem*[9, 11, 12, 16, 15])

2.1.1 Problemi dipendenti dal dominio in caso di singola sorgente

In caso di singola sorgente tipicamente si fa riferimento ad un file che non ha uno schema, non ha controllo sull'integrità dei dati immessi e che non

permette molte restrizioni sugli inserimenti. La probabilità di valori errati e di inconsistenze è pertanto elevata vedi fig. 2.3 e fig. 2.4. La comunità di

Scope/Problem		Dirty Data	Reasons/Remarks
Attribute	Illegal values	bdate=30.13.70	values outside of domain range
Record	Violated attribute dependencies	age=22, bdate=12.02.70	age = current year - birth year should hold
Record type	Uniqueness violation	emp ₁ =(name="John Smith", SSN="123456"); emp ₂ =(name="Peter Miller", SSN="123456")	uniqueness for SSN (social security number) violated
Source	Referential integrity violation	emp=(name="John Smith", deptno=127)	referenced department (127) not defined

Figura 2.3: Es. problemi di singola sorgente a livello di schema

Scope/Problem		Dirty Data	Reasons/Remarks
Attribute	Missing values	phone=9999-999999	unavailable values during data entry (dummy values or null)
	Misspellings	city="Lipzig"	usually typos, phonetic errors
	Cryptic values, Abbreviations	experience="B"; occupation="DB Prog."	
	Embedded values	name="J. Smith 12.02.70 New York"	multiple values entered in one attribute (e.g. in a free-form field)
	Misfielded values	city="Germany"	
Record	Violated attribute dependencies	city="Redmond", zip=77777	city and zip code should correspond
Record type	Word transpositions	name ₁ ="J. Smith", name ₂ ="Miller P."	usually in a free-form field
	Duplicated records	emp ₁ =(name="John Smith", ...); emp ₂ =(name="J. Smith", ...)	same employee represented twice due to some data entry errors
	Contradicting records	emp ₁ =(name="John Smith", bdate=12.02.70); emp ₂ =(name="John Smith", bdate=12.12.70)	the same real world entity is described by different values
Source	Wrong references	emp=(name="John Smith", deptno=17)	referenced department (17) is defined but wrong

Figura 2.4: Es. problemi di singola sorgente a livello di istanza

ricerca si è occupata intensamente di traduzione e di integrazione di schema, ma non si riscontra un grande interesse nei confronti della pulizia dei dati, nonostante l'elevato numero di strumenti commerciali testimoni l'importanza e la difficoltà di questo problema. Problemi tipici che possono essere individuati ed ispezionati solo sulla base della conoscenza del dominio, sono ad esempio:

Valore spezzato Es. Indirizzo1="Via Curtatone e Monta" Indirizzo2="nara"

Valore non corrispondente all'attributo richiesto Es. Città="Italia"

Errore di digitazione Es. Stato="Itallia"

Valori fuori dal dominio Es. Data di Nascita="30/13/1976"

Valori mancanti o sconosciuti

Inconsistenze, vincoli d'integrità Es. Codice Fiscale = duplicato, Età="22
Anno di Nascita="1950"

Formato e/o unità cambiate nel tempo Es. Qualità prodotto="3" vs
Qualità prodotto="C"

Duplicati Es. Nome="Sandra Zimei" vs Nome="S.Zimei".

2.1.2 Problemi dipendenti dal dominio in caso di sorgente multipla

I problemi presenti in caso di singola sorgente sono aggravati nel caso di sorgente multipla. Dopo aver sottoposto ogni singola sorgente ad operazioni di analisi e ripulitura così come visto nel paragrafo precedente, bisogna riuscire ad integrare i dati provenienti da sorgenti eterogenee in modo appropriato sempre facendo riferimento a conoscenze del dominio. A livello di istanza ci possono essere:

diverse rappresentazioni dello stesso attributo: ad es. sesso= "M/F"
VS sesso="1/0"

possibili duplicati: ad es. nome= "Kristen Smith" vs nome="Kris Smith"

A livello di schema si possono presentare:

sinonimie: ad es. attributo “Customer” VS attributo “Client”

conflitti strutturali: ad es. diverse rappresentazioni per l’attributo che modella l’indirizzo “street” + “city” VS “address”

(vedi esempio fig. 2.5) [8]. Tali problemi possono essere risolti mediante la

Customer (source 1)				
CID	Name	Street	City	Sex
11	Kristen Smith	2 Hurley Pl	South Fork, MN 48503	0
24	Christian Smith	Hurley St 2	S Fork MN	1

Client (source 2)					
Cno	LastName	FirstName	Gender	Address	Phone/Fax
24	Smith	Christoph	M	23 Harley St, Chicago IL, 60633-2394	333-222-6542 / 333-222-6599
493	Smith	Kris L.	F	2 Hurley Place, South Fork MN, 48503-5998	444-555-6666

Customers (integrated target with cleaned data)											
No	LName	FName	Gender	Street	City	State	ZIP	Phone	Fax	CID	Cno
1	Smith	Kristen L.	F	2 Hurley Place	South Fork	MN	48503-5998	444-555-6666		11	493
2	Smith	Christian	M	2 Hurley Place	South Fork	MN	48503-5998			24	
3	Smith	Christoph	M	23 Harley Street	Chicago	IL	60633-2394	333-222-6542	333-222-6599		24

Figura 2.5: Es. problemi di sorgente multipla a livello di Schema e di Istanza

verifica delle condizioni di “similitudine” e la conversione in formati uniformi.

2.1.3 Normalizzazione

In presenza di dati numerici, può essere utile prevedere una fase preliminare di *standardizzazione*. Spesso, infatti, tali variabili numeriche rappresentano misure *interval-scaled* derivanti da rilevazioni fatte basandosi su scale differenti che le rendono inconfondibili. Le tecniche di standardizzazione possono essere scelte fra le tecniche di normalizzazione (utili anche in fase di trasformazione del processo KDD). La **normalizzazione min-max**, ad

esempio, preserva le relazioni fra i valori originali e mappa un valore x_i nel nuovo valore x'_i in un range $[\text{newMin}, \text{newMax}]$:

$$x'_i = \frac{x_i - \text{min}}{\text{max} - \text{min}}(\text{newMax} - \text{newMin}) + \text{newMin}$$

dove min e max sono i valori di minimo e di massimo che l'attributo assumeva nei dati originali. Se tali due valori fossero sconosciuti, si può ricorrere alla **normalizzazione z-score** calcolando z_i come:

$$z_i = \frac{x_i - \bar{A}}{\sigma_A}$$

dove \bar{A} rappresenta la media

$$\bar{A} = \frac{x_1 + \dots + x_n}{n}$$

e σ_A rappresenta la deviazione standard. Alternativamente σ_A può essere sostituito dalla *deviazione assoluta media*

$$s_f = \frac{1}{n}(|x_1 - \bar{A}| + \dots + |x_n - \bar{A}|)$$

che, non prevedendo l'elevazione al quadrato, riduce i problemi dovuti agli outlier.

2.1.4 Discretizzazione

La discretizzazione consiste nel ridurre i valori assunti da un attributo ad un insieme di valori meno numeroso. La discretizzazione a seconda del tipo di attributo può essere:

- numerica,
- categorica.

Il processo di discretizzazione di attributi numerici è una fase fondamentale per la maggior parte degli algoritmi di DM. È importante sottolineare come tale processo sia critico dal punto di vista della qualità della conoscenza acquisita dall'algoritmo di data mining, infatti, la semplice scelta del numero di intervalli di discretizzazione per un attributo può avere conseguenze considerevoli:

- se il numero di intervalli è troppo piccolo si potrebbero perdere informazioni utili (tanto varrebbe eliminare l'attributo);
- se il numero di intervalli è troppo grande, al contrario, si potrebbe ricadere in un problema di sovra-addestramento e l'algoritmo potrebbe non essere in grado di estrarre conoscenza significativa, o quanto meno interpretabile, dai dati.

Purtroppo non esistono delle regole generali valide per guidare questa fase. Gli algoritmi utilizzati per la discretizzazione degli attributi possono suddiversi in due categorie: gli algoritmi naïf e gli algoritmi basati sulla teoria dell'informazione. Gli algoritmi naïf sono i più semplici da un punto di vista concettuale tra quelli presenti in letteratura, ciò nonostante sono quelli che meglio si adattano alle situazioni in cui la quantità dei dati da processare è notevole:

- EWD (Equal Width Discretization): genera intervalli di ampiezza uguale;
- EFD (Equal Frequency Discretization): ogni intervallo contiene approssimativamente lo stesso numero di elementi;
- Discretizzazione mediante clustering: algoritmo di clustering (ad esempio il K-Means) per raggruppare i valori numerici, quindi si desumono gli intervalli dalle estensioni dei vari cluster.

La discretizzazione categorica consiste :

- nel sostituire il valore categorico assunto dall'attributo, spostandosi ad un livello di astrazione superiore. Questa tecnica è applicabile quando si dispone di una gerarchia dalla quale poter determinare la categoria di appartenenza di ciascun valore assunto dalla variabile categorica;
- nel sostituire il valore categorico con il suo rango numerico, quindi nell'applicare una discretizzazione numerica a tali nuovi valori. Questo metodo può essere applicato alle variabili **ordinali**, cioè variabili che assumono valori che rappresentano stati per i quali esiste una sequenza significativa.

2.1.5 Campionamento

Il campionamento consiste nella traduzione dei dati in un insieme rappresentativo. Questa operazione è un passo preliminare importante che risulta di supporto alla fase della stima dell'accuratezza dei classificatori. La stima dell'accuratezza dei classificatori è un passo importante che permette di sapere quanto accuratamente un classificatore dato, etichetterà dati che non sono stati trattati al momento della costruzione del modello. Usare un *training set* per costruire il modello di classificazione e poi usare lo stesso insieme per misurare l'accuratezza del modello, può dare risultati di stime troppo ottimistiche. Metodi di valutazione del modello sono :

- Hold-out,
- k-fold cross-validation,
- Leave-one-out,
- Bootstrap;

la scelta del metodo di valutazione varia a seconda della quantità dei dati disponibili, essa può seguire le linee guida indicate in fig.2.6 [22]. Quando

Pattern	Solution
Model Evaluation	<ol style="list-style-type: none"> 1. If a separate testing set is available apply <i>Hold-Out</i>. 2. Otherwise: <ul style="list-style-type: none"> - If more than 100 examples are available, apply <i>K-fold Cross-validation</i> with $K = 10$. - With fewer than 100, apply <i>Leave-One-Out</i>. - With fewer than 50 and <i>Leave-One-Out</i> resulting in an estimate for the error rate less than 0.632, apply <i>Bootstrap</i>.

Figura 2.6: Metodi di valutazione dell'accuratezza di classificatori.

si ha a disposizione una gran quantità di dati (large dataset), si applica il metodo *Hold Out* che prevede la suddivisione in due insiemi: *Training Set* (con una cardinalità di circa $\frac{2}{3}$ del totale dati a disposizione) and *Test Set* ($\frac{1}{3}$ del totale). Il training set sarà la base per la costruzione dei classificatori, mentre il test set verrà usato per stimarne l'accuratezza. Per insiemi di dati di dimensione media si ricorre al metodo *K-fold cross-validation* che stima l'accuratezza di un modello M , procedendo con i seguenti passi:

1. divide il data set (di cardinalità n) in k sottoinsiemi disgiunti S_1, \dots, S_k di dimensione approssimativamente uguale,
2. usa $k-1$ sottoinsiemi come training set per costruire un modello M_i , e un sottoinsieme come test set per valutare l'accuratezza di M_i ottenendo l'errore che esso comporta (e_i),

3. ripete il passo 2) k volte usando come test set un insieme sempre diverso ottenendo gli errori di valutazione $e_1, ..e_k$ per ogni modello,
4. calcola l'accuratezza del modello M come media $e = \frac{\sum_{i=1}^k e_i}{k}$

Quando i dati a disposizione si presentano come uno Small-Set, pertanto le tecniche più adatte risultano essere il *Leave-one-out* e il *Bootstrap*. Il *Leave-one-out* porta buoni risultati in caso di data set con di cardinalità media, questo non è altro che un caso particolare del *K-fold cross validation*, con k pari al numero n degli elementi nel data set. I passi previsti per la valutazione di M sono:

1. costruisce un modello M_i sul training set composto da $n - 1$ elementi, e testa M_i sull'elemento rimanente i (di classe c_i) ottenendo $j_i = 0$ se il modello assegna correttamente i a c_i e $j_i = 1$ altrimenti,
2. Ripete il passo 1) n volte fino ad ottenere stime degli errori per tutti gli n modelli,
3. calcola la stima dell'errore per il modello M come $e = \frac{\sum_{i=1}^n j_i}{n}$

La tecnica di Bootstrap, è basata su procedure statistiche di campionamento con rimpiazzo per formare il training set. Un dataset di n elementi è campionato n volte , con rimpiazzo, ottenendo un altro dataset di n elementi (training set) [21]. Dato che ci saranno alcuni elementi del data set iniziale che non verranno mai presi per formare il *training set* (che conterrà qualche elemento ripetuto con alta probabilità), saranno proprio questi gli elementi il *test set*. La probabilità che un elemento non venga scelto per formare il training set, è pari a :

$$\left(1 - \frac{1}{n}\right)^n \approx e^{-1} = 0.368$$

da qui il nome del metodo *0.632 Bootstrap*. Risultati sperimentali hanno dimostrato che l'uso di tale tecnica è consigliato con dataset piccoli (di cardinalità pari a circa 50), e su cui il metodo *leave-one-out* porta ad un tasso di errore e minore di 0.632 [22]. Questo metodo di valutazione dell'accuratezza del modello prevede:

1. costruire un campione (training set) di n elementi prelevandoli (con rimpiazzo) dal dataset iniziale D di cardinalità n .
2. usare questo campione per costruire un modello M_i ,
3. calcolare il tasso di errore sul training set ($e(\text{training})_i$),
4. usare gli elementi di D non inclusi nel training set per formare il test set, quindi calcolare il tasso di errore $e(\text{test})_i$ che si ha usando tale insieme per valutare M_i ,
5. calcolare l'errore complessivo di valutazione di M_i come

$$e_i = 0.632 e(\text{training})_i + 0.368 e(\text{test})_i$$

6. ripeti i passi 1),2),3),4), e 5) n volte per ottenere i valori delle stime di errori e_1, \dots, e_n ,
7. calcola la stima del modello M come media

$$e = \frac{\sum_{i=1}^n e_i}{n}$$

L'uso della combinazione lineare

$$0.632 e(\text{training})_i + 0.368 e(\text{test})_i$$

proposto in [20], è inteso come il bilanciamento delle stime ottimistiche, $e(\text{training})$, e pessimistiche, $e(\text{test})$, del tasso di errore. Questo metodo diminuisce l'influenza che si avrebbe sui risultati a causa di un insieme di dati disponibili

poco numeroso. Esso permette di avere training set e test set abbastanza consistenti da poter valutare l'accuratezza di un modello scelto per la classificazione dei dati o, comunque, di scegliere il modello che da risultati migliori. La prima fase per la costruzione di un modello accurato, resta comunque la costituzione di un training set che sia un insieme campione rappresentativo per il dataset. Le tecniche di campionamento più utilizzate sono:

Campionamento casuale: in cui la costituzione dell'insieme campione avviene considerando ogni elemento dell'insieme con eguale probabilità di essere estratto. Tale metodo in generale, garantisce delle buone prestazioni sia dal punto di vista delle risorse computazionali necessarie, sia dal punto di vista della qualità del campione estratto.

Campionamento su clustering: si procede con un pre-raggruppamento dei dati in base ad un opportuno criterio, ricadendo, così, nel problema più generale del clustering. Quindi si estraggono, sempre casualmente, da ogni cluster un numero di campioni proporzionale alla dimensione del cluster sull'insieme di partenza. All'interno di ogni cluster ogni campione ha la stessa probabilità di essere estratto. Se il clustering viene effettuato in modo opportuno, ovvero scegliendo in modo oculato l'algoritmo e la tecnica di calcolo della distanza, vi è la possibilità di ridurre sostanzialmente il rumore dei dati in quanto esempi anomali e poco significativi possono venire esclusi dall'algoritmo di clustering e pertanto avranno una probabilità nulla di essere estratti. Un altro tipo di campionamento su clustering consiste nell'estrarre casualmente un sottoinsieme dell'insieme dei clusters di partenza, mantenendo inalterate le correlazioni inter-clusters.

Campionamento stratificato: nel caso di attributi categorici nominali è

possibile estrarre un insieme campione prelevando una proporzione degli elementi del dataset di partenza per ciascun valore assunto dall'attributo. Ogni gruppo di elementi con lo stesso valore dell'attributo rappresenta uno strato, per la costituzione dell'insieme campione ciascuno strato si considera equiprobabile.

2.1.6 Eliminazione dei Duplicati

Le fonti da cui si attinge per la composizione di un data warehouse, sono diverse e vanno integrate in un unico dataset. Dopo aver risolto problemi di :

- integrazione di schema,
- errori di sintassi,
- errori topografici,
- eliminazione di sigle e abbreviazioni,

vanno eliminati tutti i possibili duplicati per costruire un data warehouse con valori significativi per le successive fasi di analisi. Il problema, noto in letteratura come *merge/purge problem*, riguarda l'identificazione di record che forniscono informazione riguardanti la stessa entità. In [18] viene proposto un metodo SNM (Sorted Neighborhood Method) per individuare duplicati approssimati, ordinando il dataset sulla base di una chiave specifica. La chiave può essere un sottoinsieme degli attributi, o sottostringhe che compongono gli attributi stessi, essa è scelta in modo da essere sufficientemente discriminante per individuare i possibili duplicati. Il metodo SNM può essere schematizzato nei seguenti passi:

1. Creazione della chiave,

2. Ordinamento dei dati sulla base della chiave creata al passo 1,
3. Unificazione dei dati simili.

Sottoponendo l'intero database dei dati su cui basare la creazione del data warehouse, a questo processo si ottiene un dataset in cui i dati di un oggetto che si presentano come più di un'istanza vengono unificati astruendo da problemi che li rendono apparentemente diversi come ad esempio errori di inserimento, l'uso di abbreviazioni, l'incompletezza delle informazioni su una delle istanze.

Creazione della chiave e ordinamento

L'efficacia del processo di de-duplicazione SNM dipende pesantemente dalla scelta della chiave in base alla quale verranno ordinati i record supponendo che record vicini saranno con alta probabilità simili.

Unificazione di dati simili

Partendo dalla lista ordinata di record, l'unificazione di dati simili prevede di far scorrere di un record alla volta, una finestra di dimensione w sulla lista e di confrontare i record che sono inclusi nella finestra in un passo, con quelli inclusi nella finestra al passo precedente (vedi fig.2.7). Il confronto fra record è una fase molto complessa che non si limita a considerare gli attributi della chiave. Supponiamo, ad esempio che due record siano quasi identici negli attributi chiave ,ad eccezione di una piccola differenza come ad esempio la data di fine progetto. In questo caso si può decidere che i due record siano un caso di informazione duplicata, ma potrebbe anche essere il caso di due progetti distinti che la stessa azienda ha portato avanti contemporaneamente. Se nessuna altra informazione porta a poter prendere

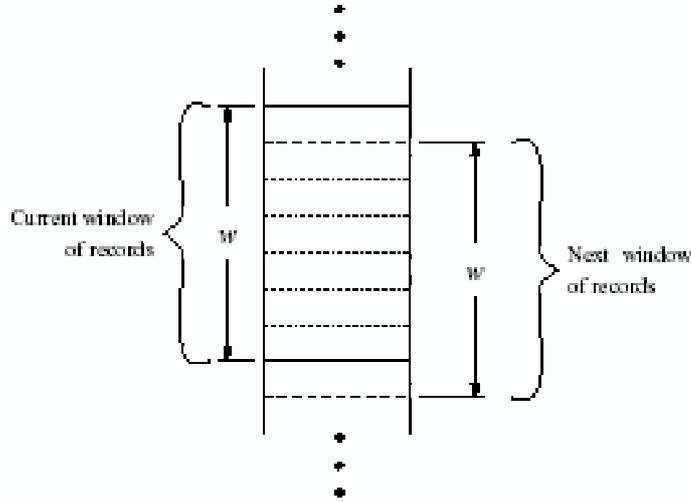


Figura 2.7: Esempio dei record contenuti in una finestra di dimensione w , in due passi successivi, durante la fase di individuazione dei duplicati.

una decisione opportuna si assume che i record siano distinti. La similitudine fra valori di un attributo può essere calcolata suddividendo ciascun valore in *token* (individuato come l'insieme di caratteri tra due spazi bianchi. Dati due record X e Y e i valori X_A e Y_A assunti dello stesso attributo A , X_A e Y_A vengono suddivisi in t_{xA1}, \dots, t_{xA_n} e t_{yA1}, \dots, t_{yA_m} , ogni token t_{xA_i} viene confrontato con tutti i t_{yA_j} (con $1 \leq j \leq m$). Siano $DoS(t_{xA1}), \dots, DoS(t_{xA_n})$ e $DoS(t_{yA1}), \dots, DoS(t_{yA_m})$ i massimi gradi di similitudine calcolati per ogni token, la similitudine per l'attributo A relativa alle istanze X e Y è data da :

$$Sim_A(X, Y) = \frac{\sum_{i=1}^n DoS(t_{xA_i}) + \sum_{j=1}^m DoS(t_{yA_j})}{n + m}$$

Il grado di similitudine fra due token può essere calcolato come

$$(1 - EDIT(t_i, t_j))$$

dove $EDIT(t_i, t_j)$ è la distanza di edit tra t_i e t_j cioè il minimo numero di operazioni (inserzione, cancellazione, sostituzione) necessarie per trasformare un token nell'altro normalizzato sulla somma delle lunghezze dei token. La similitudine fra due record X e Y che includono k attributi $(A_1, ..A_k)$, supponendo che a ciascun campo A_i venga dato un peso $W(A_i)$ (in base all'importanza che il campo ha per poter determinare l'uguaglianza fra due record), è data da:

$$SIM(X, Y) = \sum_{i=1}^k Sim_{A_i}(X, Y) * W(A_i).$$

Per la natura dei dati, a volte ha poco senso l'applicazione di tecniche di matching approssimato, più calzante invece è l'uso di matching esatto prevedendo solo risposte 0, 1 per i casi di valori diversi o uguali rispettivamente, su ogni singolo campo dei record. E' sufficiente, quindi, sommare gli esiti dei confronti su ogni attributo e normalizzare sul numero degli attributi che compongono i record per poter ottenere il grado di similitudine tra i due (se si ottenesse 1, i record sono identici). Calcolato il grado di similitudine fra due record, se questo supera una soglia minima stabilita (con un valore ragionevole per poter decidere che i due sono duplicati), si passa all'unificazione.

2.1.7 Trattamento dei Dati Mancanti

Un attributo può avere un valore mancante su una istanza per ragioni diverse:

- l'intervistato decide di non rispondere,
- l'intervistato non risponde perché non esiste fra le possibili risposte messe a disposizione una risposta che sia inerente al suo caso,

- mancanza di documentazione da cui rilevare il dato,
- errori in fase di inserimento di dati.

In generale un valore mancante di un attributo Y può essere [19]:

OAR (Observed at Random) se il dato mancante su Y è indipendente da altri attributi X , detto in altri termini se X determina la mancanza di Y allora il dato non è OAR. Ad esempio se ciascun record del mio dataset contiene valori relativi ad attributi quali Reddito ed Età, volendo dimostrare che i dati di reddito mancanti sono dati OAR posso suddividere il dataset in due sottoinsiemi : quello con i record che hanno il valore reddito significativo, e quello con i record con il valore reddito mancante. Osservando tali sottoinsiemi, se i valori assunti dall'attributo età, non sono sostanzialmente differenti tra i due, allora possiamo dire che il dato mancante Reddito è OAR. Se invece, ad esempio, si osserva che il titolo di studio determina la mancanza di dati relativi al reddito (le persone con livello culturale più elevato non danno informazioni sul loro reddito) allora il reddito non è OAR.

MAR (Missing at Random) se il dato mancante su Y è indipendente da Y stesso. Supponiamo di avere solo due attributi X e Y , X sempre osservato e Y con dei valori mancanti. Si dice che Y è MAR se la probabilità di avere un valore mancante su Y , dati X e Y , è la stessa probabilità di avere un valore mancante su Y dato X :

$$P(Y_{missing}|Y, X) = P(Y_{missing}|X).$$

Ad esempio la mancanza di dati sul reddito Reddito potrebbe essere dipendente dal livello culturale di una persona ma indipendente dal reddito stesso, allora il dato è MAR. D'altra parte se invece la mancanza

di un dato di reddito fosse legata al Reddito (ad esempio è possibile che chi non fornisce i suoi dati di reddito abbia un reddito alto) allora il dato non è MAR. E' chiaro che sapere se un dato è MAR non è possibile visto che bisognerebbe poter confrontare il dataset con i dati mancanti con il dataset completato dai valori effettivi che si sarebbero dovuti avere anche per quei dati, è possibile però fare delle ipotesi basate sulla conoscenza del dominio.

MCAR (Missing Completely at Random) se il dato mancante su Y è completamente indipendente dai valori che Y stesso assume e da tutti i valori degli altri attributi X nel dataset. In altre parole $MCAR = MAR + OAR$. Se questa assunzione potesse essere fatta su tutti gli attributi del dataset, si potrebbe considerare l'insieme dei record completi (cioè senza attributi con valori mancanti), come un sottoinsieme completamente *random* rispetto all'insieme originario delle osservazioni. E' importante notare che la definizione di un valore di un attributo come MCAR nulla ha a che fare con la mancanza di valori di altri attributi, è possibile quindi che la mancanza di valori su Y sia legata alla mancanza di valori su un altro attributo X . Ad esempio, anche se le persone che si rifiutano di fornire il loro reddito, non forniscono neanche la loro età, questa non è una condizione sufficiente per poter assumere che i valori mancanti per l'attributo Reddito e l'attributo Età non siano valori MCAR. Al contrario se invece si ha che le persone che si rifiutano di fornire il loro reddito sono più giovani di quelle che forniscono il loro reddito, allora la condizione di MCAR sarebbe violata. La condizione di MCAR è una condizione molto forte e difficilmente verificabile, ma ci sono situazioni in cui se l'acquisizione di dati per alcuni attributi fosse troppo costosa, si può decidere di raccogliere informazioni solo per un

sottoinsieme scelto a caso degli individui da includere nel dataset, e considerare il valore per quell'attributo come MCAR per gli individui rimanenti del dataset completo.

Ignorabili sono i dati mancanti MCAR o almeno MAR.

Non-Ignorabili sono i dati mancanti che non sono MAR.

I metodi convenzionali per il trattamento dei dati mancanti sono:

Listwise Deletion è l'approccio più semplice, esso prevede il mantenimento dei soli record completi. Ovviamente questo approccio porta alla perdita di numerose informazioni ed è fortemente sconsigliato soprattutto quando i dati mancanti sono molto numerosi nel dataset a disposizione. Altro aspetto negativo è l'ipotesi poco realistica, che questo approccio prevede, secondo la quale i dati siano MCAR se non lo fossero, l'eliminazione dei record con dati mancanti porterebbe a risultati non significativi.

Pairwise Deletion è un approccio simile al *Listwise Deletion*, ma che prevede di mantenere un record con dati mancanti solo quando nell'analisi che si sta svolgendo il dato necessario non è mancante, altrimenti tale record non viene considerato. Oltre al fatto che tale metodo porta a risultati non significativi in caso di dati non MAR, questo metodo risulta povero di fondamenti statistici per poterne calcolare gli errori dei valori risultanti.

Mean Substitution questo metodo porta alla sostituzione di tutti i valori mancanti con il valore medio che ciascun attributo assume sulla base dei valori noti. E' chiaro che questa soluzione porta a una errata visione dei dati che risulterebbero rappresentati da una distribuzione distorta.

Decision Tree Imputation prevede una prima fase di discretizzazione dei valori degli attributi numerici e successivamente si procede come segue:

1. divide il dataset S in due sottoinsiemi S_k e S_m che rappresentano rispettivamente l'insieme dei record completi e quello dei record con valori mancanti,
2. Per ogni attributo A in S_k che ha valori mancanti in S_m :
 - si costruisce un albero di decisione allo scopo di prevedere il valore di A mediante tutti gli attributi ad eccezione di A ,
3. Per ogni record I di S_m (in cui ci sono sicuramente attributi con valori mancanti)
 - sostituisce i valori mancanti per ogni attributo basandosi sul corrispondente albero di decisione costruito nel passo 2.

E' chiaro che tale soluzione può essere accettabile solo quando il numero dei record completi è tale da poter avere un insieme di base S_k sufficiente per la base costruzione di alberi di decisione. Inoltre è da sottolineare come tale metodo sia poco efficace in caso di dataset con molti record con diversi attributi mancanti, questa situazione infatti porta alla necessità di fare previsioni di un valore in una istanza della quale si hanno poche altre informazioni e pertanto non si può seguire con certezza il flusso di un albero di decisione ma si può solo assumere l'appartenenza ad una classe con una probabilità ≤ 1 .

HotDecking prevede il rimpiazzo di un'osservazione mancante di un attributo di un record con il valore che tale attributo assume su un record *simile* (identificandone la similitudine tramite clustering o classificazione). Se i record simili fossero più d'uno si può decidere di prendere

come valore di riferimento per la sostituzione, la media oppure la moda (il più frequente). Questo metodo prende come ipotesi che i record possano essere organizzati in classi con piccole differenze intraclasse.

Imputation by Regression sviluppa una equazione di regressione basata sui dati completi, viene imputato un valore ad un dato mancante su un attributo basandosi sui valori degli altri attributi che vengono considerati indipendenti per ipotesi (MCAR). Il problema più grave di tale tecnica è dovuto al fatto che i dati disponibili possono portare alla stima di equazioni differenti per la regressione contro l'ipotesi assunta che l'equazione di imputazione che modella al meglio i dati sia solo una.

Multiple Imputation risolve il problema che si pone nell'approccio descritto precedentemente (Imputation by Regression) prevedendo la possibilità di imputare diversi valori ad ogni valore mancante producendo un array di dataset completi. A questo punto usando tutti i dataset e calcolandone i valori medi si ottiene il dataset che costituirà l'unico insieme di dati completi su cui basarsi per le analisi successive. Oltre la complessità dell'implementazione di tale tecnica, un aspetto che spesso la rende inapplicabile è l'assunzione che essa prevede: i dati mancanti devono essere MAR.

2.2 Tecniche di Preprocessing indipendenti dal dominio

Nel processo KDD sono indispensabili tutte quelle funzioni di trasformazione che sono applicate nell'ultimo stadio della preparazione. Si tratta delle trasformazioni che adattano le rappresentazioni delle informazioni alle

specifiche degli algoritmi di Data Mining. Queste tecniche si attuano sui dati già integrati e ripuliti dalle anomalie che ne influenzavano la qualità, riservandosi, comunque, di fornire risultati che necessitano ancora di essere ispezionati da tecniche di preprocessing dipendenti dal dominio. A queste appartengono la *classificazione* e il *clustering*. Esiste una sostanziale differenza, che è opportuno chiarire, tra le tecniche di classificazione e le tecniche di raggruppamento o clustering. Tramite la classificazione si specifica la caratteristica chiave (“*supervised learning*”), di cui i membri del gruppo devono essere dotati, e non ci si occupa di nessun altro attributo che i membri del gruppo possono avere in comune. Nel clustering si parla di “*unsupervised learning*” non esiste, cioè, una conoscenza a priori delle classi in cui suddividere i dati. In tal caso si creano dei gruppi sulla base di tutti gli attributi presenti nel data warehouse, in modo che ogni gruppo sia caratterizzato da elementi “*simili*” in termini degli attributi descritti nel data warehouse e che due elementi appartenenti a gruppi diversi siano sufficientemente “*distanti tra di loro*”. Entrambe le tecniche di clustering e classificazione, basate su criteri di valutazione di similitudine necessitano di trasformazioni a livello di istanza che rendano confrontabili i valori assunti dai dati. Le operazioni preliminari che risultano fondamentali sono ad esempio:

- la normalizzazione,
- la discretizzazione.

2.3 La Clusterizzazione

La clusterizzazione è una tecnica finalizzata al raggruppamento di dati simili tra loro in *clusters*. Concettualmente si pensa al clustering come ad un processo con due componenti principali [1]:

1. individuazione delle classi (clusters) che rappresentano l'intero insieme di dati,
2. modellazione di *descrizioni* per ogni classe individuata.

I più importanti approcci per il clustering sono [17]:

algoritmi di Partizionamento costruiscono un numero prestabilito di partizioni e quindi le valuta secondo qualche criterio,

algoritmi di Gerarchia creano una decomposizione gerarchica secondo qualche criterio e con un numero di livelli che varia a seconda di una condizione di terminazione stabilita,

algoritmi basati su Grid basati su una struttura di granularità a multi-livello,

algoritmi basati su modelli si basano su un modello ipotizzato per ogni cluster e cercano la migliore collocazione dei dati in uno dei modelli dati,

algoritmi basati sulla densità basati su funzioni di connettività e densità.

Un buon metodo di clustering produce risultati di alta qualità se porta alla costruzione di clusters con un alto grado di *somiglianza* intra-classe, e un basso grado di *somiglianza* inter-classe degli elementi collezionati. In generale la qualità delle tecniche di clustering varia in base alla scelta ed implementazione della misura di somiglianza, oltre che in base alla capacità di prevedere parti nascoste del modello. La scelta della misura di somiglianza è importante e varia a seconda dei tipi di dati che bisogna clusterizzare:

- variabili numeriche;

- variabili binarie;
- variabili categoriche: ordinali, nominali;
- variabili di tipo misto.

2.3.1 Clustering per variabili numeriche

In presenza di attributi che assumono valori numerici, per calcolare la distanza tra due oggetti può essere utile prevedere una fase preliminare di *standardizzazione* come quelle viste in 2.1.3. La tecnica di clustering più largamente usata per la manipolazione di oggetti descritti da attributi numerici è quella del clustering partizionale [13], in cui si prevede il raggruppamento di oggetti simili con attributi numerici in k clusters, basandosi sul calcolo della distanza tra gli oggetti e il rappresentante di ciascun cluster. Tale distanza può essere individuata come la distanza tra ogni attributo (standardizzato o non) di ciascun oggetto, e calcolata in diversi modi, ad esempio come **distanza Euclidea**

$$d(i, j) = \sqrt{(|x_{i1} - x_{j1}|)^2 + \dots + (|x_{ip} - x_{jp}|)^2}.$$

Nel clustering gerarchico è necessaria, invece, una misura di distanza fra clusters (C_i, C_j); le più ampiamente usate in presenza di attributi numerici

sono :

$$d_{mean}(C_i, C_j) = | m_i - m_j | \quad (\text{dove } m_i \text{ e } m_j \text{ sono le medie dei clusters } C_i \text{ e } C_j);$$

$$d_{ave}(C_i, C_j) = \frac{1}{n_i n_j} \sum_{p \in C_i} \sum_{p' \in C_j} | p - p' | \quad (n_i \text{ e } n_j \text{ numero di oggetti in } C_i \text{ e in } C_j);$$

$$d_{max}(C_i, C_j) = \max_{p \in C_i, p' \in C_j} | p - p' |;$$

$$d_{min}(C_i, C_j) = \min_{p \in C_i, p' \in C_j} | p - p' |;$$

$$dist(C_i, C_j) = \min_{p \in i.rep, q \in j.rep} d(p, q) \quad (\text{dove } i.rep \text{ e } j.rep \text{ denotano gli insiemi dei rappresentanti di } C_i \text{ e } C_j) [?].$$

2.3.2 Clustering per variabili binarie

In presenza di attributi che assumono valori binari, per calcolare la distanza tra due oggetti si può ricorrere ad una matrice di contingenza [1]: e

		Object j		
		1	0	sum
Object i	1	a	b	a+b
	0	c	d	c+d
sum		a+c	b+d	p

Figura 2.8: Matrice di contingenza di attributi binari di due oggetti i e j.

calcolare la distanza fra i due oggetti ricorrendo al **coefficiente di matching semplice** (per variabili simmetriche):

$$d(i, j) = \frac{b+c}{p},$$

oppure al **coefficiente di Jaccard** (per le variabili asimmetriche):

$$d(i, j) = \frac{b + c}{a + b + c}.$$

Nel caso di oggetti con attributi binari, le tradizionali tecniche di clustering partizionali, basate sulla distanza euclidea e sul calcolo del centroide come media degli elementi [13], risultano poco soddisfacenti nel caso di **large database**. E' possibile, infatti, che un oggetto sia molto distante dal centroide di un cluster, ma che sia in realtà strettamente legato da molti **link** con gli elementi che appartengono al cluster stesso e pertanto sarebbe, comunque, desiderabile includerlo. Più adatte risultano le tecniche di clustering gerarchico agglomerativo, in particolare quelle che fanno uso del concetto di **link** (come numero di oggetti **neighbor** in comune) per costruire i clusters [10].

2.3.3 Clustering per variabili categoriche: nominali, ordinali

Le variabili **nominali** non sono altro che una generalizzazione di una variabile binaria i cui stati possono essere anche più di 2. La distanza fra due oggetti i e j può essere calcolata tenendo conto del numero delle variabili per le quali i e j assumono lo stesso stato (m):

$$d(i, j) = \frac{p - m}{p}$$

dove p rappresenta il numero totale delle variabili. confrontate. Vista l'analogia con le variabili binarie, ogni variabile nominale può essere codificata mediante l'introduzione di una nuova variabile binaria simmetrica per ciascuno degli stati che essa può assumere. In questo modo è possibile utilizzare le stesse tecniche di clustering e di calcolo della distanza viste in 2.3.2. Le variabili **ordinali** sono simili alle variabili nominali, tranne per il fatto che

gli stati, in questo caso, sono ordinati in una sequenza significativa. E' possibile assegnare a ciascun valore assunto x_{if} dalle variabili ordinali un rango numerico

$$r_{if} \in \{1, \dots, M_f\}$$

dove M_f è il numero di valori diversi assunti da x_{if} . Dato che ogni variabile ordinale può avere un numero di stati diverso, è necessario mappare ciascun valore in $[0,1]$ per dare lo stesso peso ad ogni variabile:

$$x_{if} \mapsto z_{if} = \frac{r_{if} - 1}{M_f - 1}.$$

A questo punto si possono trattare con le stesse tecniche di calcolo della distanza e di clustering, viste per le variabili numeriche in 2.3.1.

2.3.4 Clustering per variabili di tipo misto

Nei casi reali è molto alta la probabilità di avere oggetti con attributi di tipo diverso (numerico, binario, nominale, ordinale). La distanza fra due oggetti i e j che hanno p attributi di tipo misto, può essere definita come [1]:

$$d(i, j) = \frac{\sum_{f=1}^p \delta_{ij}^{(f)} d_{ij}^{(f)}}{\sum_{f=1}^p \delta_{ij}^{(f)}}$$

dove $\delta_{ij}^{(f)} = 0$ se :

1. x_{if} o x_{jf} è un valore mancante oppure
2. $x_{if} = x_{jf} = 0$ e f è una variabile binaria asimmetrica

altrimenti $\delta_{ij}^{(f)} = 1$. La distanza $d_{ij}^{(f)}$ rappresenta il contributo di ciascun attributo f alla dissimilarità globale fra i due oggetti i e j ; tale distanza è calcolata a seconda del tipo dell'attributo:

- f è binario o nominale se $x_{if} = x_{jf} \Rightarrow d_{ij}^{(f)} = 0$, altrimenti $d_{ij}^{(f)} = 1$;

- f numerico , $d_{ij}^{(f)} = \frac{|x_{if} - x_{jf}|}{\max x_{hf} - \min x_{hf}}$ (dove h varia fra tutti i valori non mancanti);
- f ordinale, si calcola il rango r_{if} e $z_{if} = \frac{r_{if}-1}{M_f-1}$, quindi si tratta z_{if} come numerica.

2.4 La Classificazione

La Classificazione è un processo a due fasi [1]:

1. costruzione di un modello di classificazione (regole di classificazione) fig. 2.9;
2. utilizzo del modello costruito per la classificazione di nuovi valori fig. 2.10.

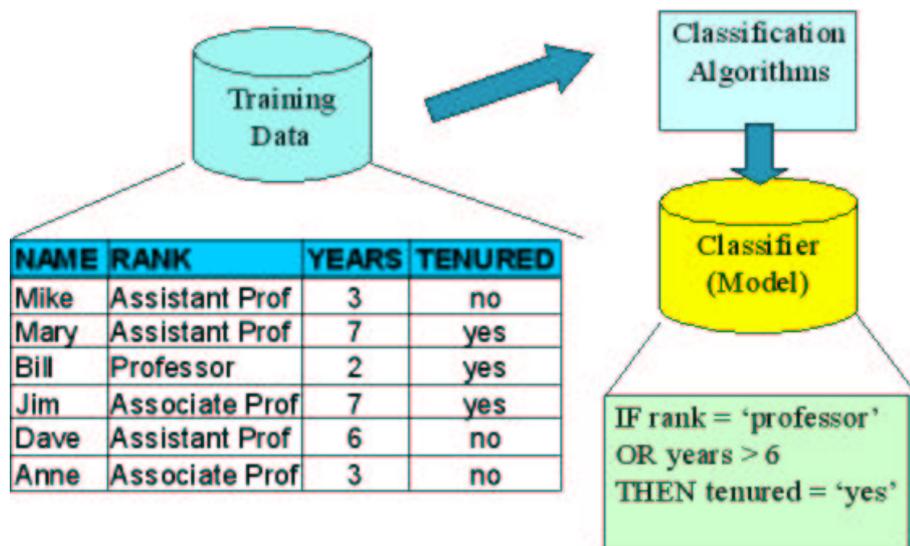


Figura 2.9: Prima fase di classificazione: Costruzione di un modello.

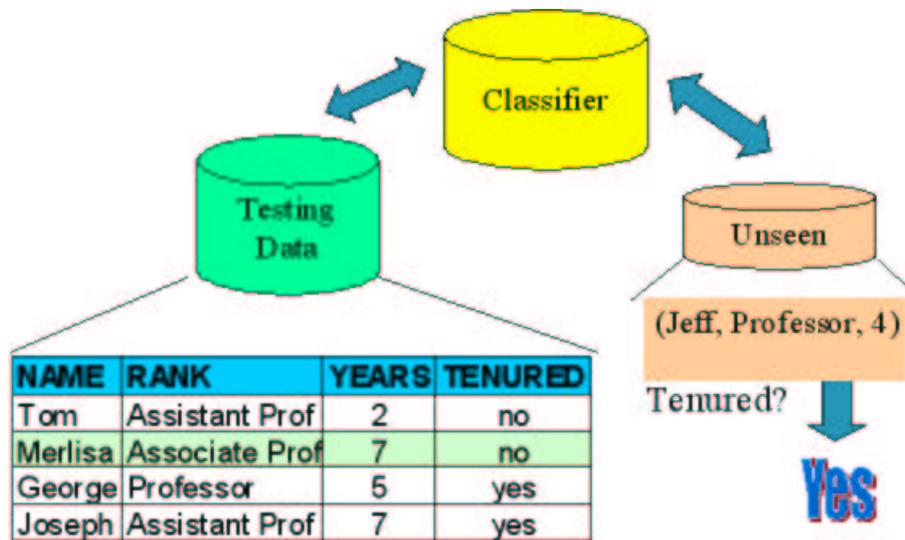


Figura 2.10: Seconda fase di classificazione: Uso del modello.

Nella prima fase si applicano algoritmi al *training set* (insieme di informazioni) che, sulla base di **un** attributo (attributo di etichetta di classe), classificano ciascuna tupla (insieme di attributi che descrivono un dato). Gli algoritmi più utilizzati sono [1]:

- classificatori Bayesiani,
- algoritmi basati su alberi di decisione,
- approcci con l'utilizzo di logica fuzzy.

Nella seconda fase si utilizzano i modelli per inserire in uno dei clusters creati quegli oggetti che si aggiungono al training set in un momento successivo o che non hanno informazioni riguardo l'attributo etichetta tali da permetterne il giusto inserimento in una delle classi. Lo scopo delle tecniche di classificazione può essere quello di ottenere una suddivisione dei dati in classi che fanno riferimento non ai singoli valori assunti dai dati, ma ad una loro generalizzazione. Ciascun valore generalizzato rappresenta un livello gerarchico

superiore rispetto a quello reale rappresentato dal dato. Tale generalizzazione è ricavabile solo se preventivamente vengono attuate tecniche di preprocessing basate sulla conoscenza del dominio , come la discretizzazione.

Capitolo 3

Il Sistema KDDML-MQL

In questi ultimi anni, presso il Dipartimento di Informatica dell'Università di Pisa è stato sviluppato un ambiente generale di sviluppo per applicazioni di estrazione di conoscenza dai dati. Tale sistema, denominato KDDML (KDD Markup Language) [14, 23], è nato con l'obiettivo principale di integrare le varie fasi del processo KDD. Si è riconosciuta innanzitutto la necessità di rappresentare, in modo uniforme, i dati in ingresso, i meta-dati, i modelli di conoscenza, e gli stessi problemi di data mining. Questo requisito, indispensabile per fornire l'interoperabilità tra le diverse fasi del processo, così come tra i diversi modelli di conoscenza, è stato soddisfatto attraverso l'utilizzo di un linguaggio standard per la rappresentazione e lo scambio dei dati: il linguaggio di mark-up XML. Esso costituisce la base per la realizzazione del linguaggio di mark-up KDDML del sistema, il quale è interamente implementato in Java. In un secondo momento è stato definito e integrato nel sistema un nuovo linguaggio di interrogazione, che rendesse l'utilizzo dell'ambiente KDDML più semplice [23]. Tale linguaggio, denominato Mining Query language (MQL) [24] si pone ad un livello di astrazione superiore, conservando la flessibilità, l'estendibilità e l'espressività tipiche del linguaggio

KDDML. Tramite esso è possibile specificare un problema di estrazione di conoscenza come un processo di interrogazione sui dati. Il linguaggio, infatti, permette di combinare dati e modelli di conoscenza (regole di associazione, alberi di classificazione, clusters) attraverso degli operatori secondo uno stile algebrico. Il risultato di un'operazione può costituire l'ingresso di un altro operatore, realizzando in questo modo il principio di chiusura. Il risultato ottenuto con KDDML-MQL [23] è stato quindi un ambiente in cui l'utilizzatore interagisce inserendo queries che rispettano la grammatica definita in MQL, mentre rimane compito del sistema provvedere all'interfacciamento con KDDML per la loro esecuzione vera e propria. In KDDML-MQL [23], è stato implementato il linguaggio MQL sfruttando come motore esecutivo delle queries l'interprete di KDDML. Il linguaggio XML, di fatto, resta comunque la base per la rappresentazione delle queries e degli oggetti restituiti dagli operatori di data mining.

3.1 XML per la rappresentazione dei documenti.

Lo standard XML (eXtensible Markup Language), utilizzato per la rappresentazione dei dati semi-strutturati, fornisce informazioni di tipo semantico e strutturale dei dati. XML è stato approvato dal gruppo W3C [25] per facilitare lo scambio elettronico di dati interpretabili da macchine sulla rete; si distacca quindi dall'HTML che invece si propone come supporto per lo scambio di informazioni interpretabili dall'utente. I contributi più importanti che l'XML porta rispetto all'HTML sono :

- l'utente può definire i tag a seconda delle sue necessità;

```
<!ELEMENT PPMERGING_QUERY ((%kdd_query_PPtable;),ATTR_TO_MERGE,NEW_ATTR_NAME)>
<!ATTLIST PPMERGING_QUERY xml_dest CDATA # IMPLIED>
<!ELEMENT ATTR_TO_MERGE (ATTR+)>
<!ELEMENT NEW_ATTR_NAME (ATTR)>
<!ELEMENT ATTR EMPTY>
<!ATTLIST ATTR name CDATA # REQUIRED>
```

Tabella 3.1: Esempio di DTD per l'operatore di merging.

- le strutture dei documenti possono essere annidate ad ogni livello;
- ogni documento XML può contenere una descrizione opzionale della propria grammatica DTD (Document Type Definition), per poter sviluppare metodi di validazione strutturali.

Le DTD rappresentano la grammatica di un documento XML valido, espressa tramite espressioni regolari, modellando le produzioni di ciascun elemento e la descrizione degli attributi (vedi es. 3.1). Il gruppo W3C ha proposto un insieme di interfacce per accedere, modificare e rappresentare i documenti XML. Questo insieme di interfacce noto come DOM (Document Object Model) in letteratura [26], è stato implementato interamente in Java (XML4J [28]) dal un gruppo di ricerca IBM Alpha Works. Nel sistema KDDML-MQL si fa uso delle funzionalità fornite dalla JDK 1.4 della SUN per la manipolazione dei documenti XML. Un'altra tecnologia, proposta dal W3C, è l'Extensible Stylesheet Language (XSL) che include un linguaggio di trasformazione per associare i documenti XML ai propri fogli di stile che definiscono la loro visualizzazione sul browser. La trasformazione è realizzata da un modulo software chiamato processore XSL, che legge un documento XML e i fogli di stile associati, e restituisce il documento HTML relativo. Un processore XSL ampiamente usato è LotusXSL [27], sviluppato da IBM, che supporta le specifiche XSL proposte dal W3C. Nel sistema KDDML-MQL si fa uso della libreria LotusXSL per presentare i documenti all'utente.

3.2 Gerarchia del Sistema KDDML-MQL

Il sistema KDDML-MQL è stato sviluppato e implementato per coprire le fasi di data mining e post-processing del processo KDD. KDDML-MQL definisce la gerarchia degli oggetti (vedi fig.3.1) seguendo l'approccio suggerito da Imielinsky e Mannila in [29], nel quale il processo KDD è definito come un *query process* basato su un query language appropriato.

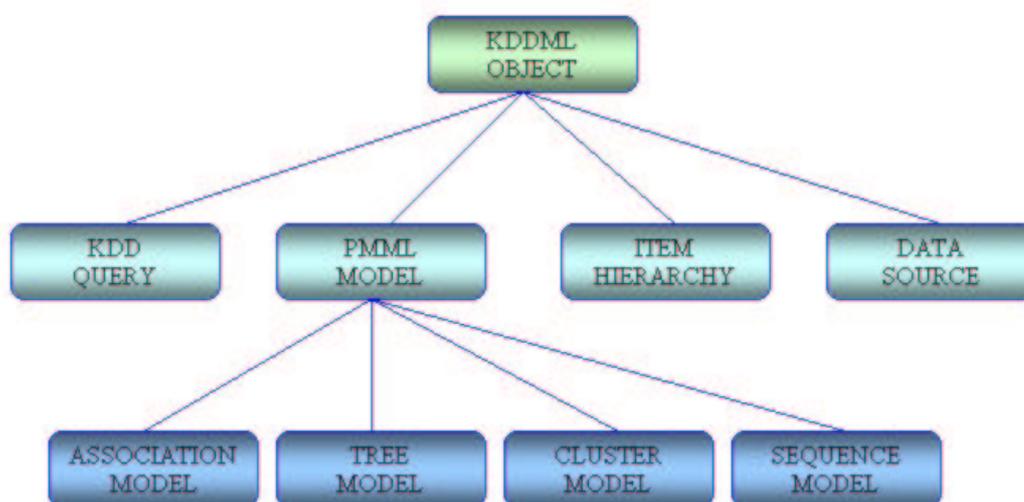


Figura 3.1: Gerarchia degli oggetti in KDDML-MQL.

3.2.1 Classi della gerarchia del Sistema:

PMML_MODEL

Predictive Model Markup Language (PMML) [30] è uno standard per la rappresentazione di modelli come documenti XML, sviluppato dal DMG (Data Mining Group), un consorzio di industrie ed organizzazioni formato per facilitare la creazione e lo sviluppo di utili standards per la comunità di DM. Questo Markup Language è composto dai seguenti elementi:

Data Dictionary definisce gli attributi di input per i modelli e specifica il tipo e il range di valori ammissibili per ogni attributo;

Mining Schema ogni modello contiene un *mining schema* nel quale sono elencati gli attributi usati nel modello. Questi attributi sono un sottoinsieme degli attributi nel Data Dictionary. Il mining schema contiene informazioni specifiche per certi modelli, mentre il data dictionary contiene le definizioni di dati che non cambiano in base al modello. Per esempio il mining schema specifica il tipo di uso che si fa di un attributo, che può essere active (un input del modello), predicted (un output del modello), oppure supplementary (riguardante informazioni descrittive e ignorato dal modello);

Transformation Dictionary definisce attributi derivati. Gli attributi derivati possono essere definiti mediante normalizzazione, discretizzazione o aggregazione;

Model Statistics contiene statistiche riguardanti gli attributi usati nel modello

Models modelli della conoscenza estratta.

Il PMML mediante una DTD chiamata PMML-DTD, descrive un'ampia gamma di modelli, compresi alberi di decisione, cluster, regole associative, reti neurali, classificazioni per regressione e naive-Bayes (vedi esempio in tab,3.2). E' da sottolineare il fatto che PMML è uno standard di rappresentazione della conoscenza estratta e non del processo di estrazione di conoscenza. Nella gerarchia delle classi del sistema KDDML-MQL, l'elemento PMML_MODEL rappresenta l'insieme di modelli che rappresentano i risultati degli algoritmi di Data Mining. I modelli considerati nel sistema sono :

```

<!ELEMENT ClusteringModel ( Extension*, MiningSchema,
ModelStats?, ComparisonMeasure, ClusteringField*, CenterFields?,
Cluster+, Extension* ) >
<!--ATTLIST ClusteringModel modelName CDATA #IMPLIED
functionName %MINING-FUNCTION; #REQUIRED
algorithmName CDATA #IMPLIED
modelClass ( centerBased | distributionBased ) #REQUIRED
numberOfClusters %INT-NUMBER; #REQUIRED -->
<!ELEMENT CenterFields ( DerivedField+ ) >

```

Tabella 3.2: PMML-DTD: Modello di cluster.

- regole associative,
- alberi di decisione,
- pattern sequenziali,
- clusters.

3.2.2 Classi della gerarchia del Sistema: KDD_QUERY

L'elemento KDD_QUERY rappresenta una generica query come un predicato che ritorna un insieme di oggetti PMML oppure data-source. La struttura delle query può essere annidata, realizzando in questo modo il principio di chiusura. La validità delle query è basata sul controllo dei tipi dei risultati prodotti dagli operatori invocati nelle sotto-query. Per controllare l'annidamento delle sotto-query, nel sistema sono state raggruppate le invocazioni agli operatori che ritornano lo stesso tipo di conoscenza nella stessa classe, mediante un'entità XML. Le entità ottenute a seguito di tale raggruppamento sono (vedi tab.3.3):

Kdd_query_clusters invocazioni a algoritmi di Data Mining e operatori che restituiscono un insieme di clusters,

Kdd_query_hierarchy invocazioni degli operatori che restituiscono una gerarchia di items,

Kdd_query_empty invocazioni di operatori che non restituiscono nessun oggetto ma effettuano ad esempio invocazioni esterne,

Kdd_query_sequence invocazioni degli operatori che restituiscono un pattern sequenziali,

Kdd_query_table invocazioni degli operatori che restituiscono un tabella,

Kdd_query_trees invocazioni degli operatori che restituiscono un albero,

Kdd_query_rda invocazioni degli operatori che restituiscono un insieme di regole associative.

3.2.3 Classi della gerarchia del Sistema:

ITEM_HIERARCHY

Per rendere possibile l'estrazione di conoscenza a diversi livelli di astrazione [31], KDDML-MQL permette la definizione e l'uso di gerarchie su insiemi di items. L'elemento Item_Hierarchy è la rappresentazione in formato XML di una gerarchia di elementi; tale gerarchia può essere di supporto ad algoritmi che restituiscono, ad esempio, un sottoinsieme di regole di associazione, che preservano la loro appartenenza gerarchica rispetto alle regole di associazione note per un livello di astrazione superiore (PRESERVED_RULES). In termini di KDD_QUERY si ha ad esempio:

```

<!--===== Dichiarazione della radice di una KDDML_QUERY =====>
<!ELEMENT KDDML_OBJECT (KDD_QUERY)+>
<!-- KDDML_OBJECT e' la radice di tutti gli oggetti esprimibili
in KDDML ed e' composto da uno qualsiasi degli oggetti
esprimibili nel linguaggio stesso.-->
<!ELEMENT KDD_QUERY (%kdd_operator;)>
<!ATTLIST KDD_QUERY name CDATA # REQUIRED>
<!-- KDD_QUERY rappresenta un problema di KDD espresso con il
linguaggio KDDML ed e' composto da un elemento che rappresenta
un generico operatore del linguaggio.KDD_QUERY ha un attributo
''nome'' che esprime il nome della query stessa. -->
<!ENTITY % kdd_operator (%kdd_query_clusters;|%kdd_query_rules;|
%kdd_query_sequence;|%kdd_query_table;|%kdd_query_trees;|
%kdd_query_hierarchy;|%kdd_query_empty;)>
<!-- kdd_operator si riferisce all'or dei nomi degli elementi che
rappresentano un qualsiasi operatore del linguaggio KDDML. -->

```

Tabella 3.3: KDDML_Object: elementi della KDD_QUERY.

```

<KDD_QUERY name="hierarchy.xml">
  <PRESERVED_RULES xml_dest="preserved_rules.xml">
    <MINE_RULE>
      <FILE_ARFF file_name="data.arff"/>
      <ALGORITHM name="DCI">
        <PARAM name="min_confidence" value="0.7">
        <PARAM name="min_support" value="0.4">
        <PARAM name="number_of_rules" value="10">
      </ALGORITHM>
    </MINE_RULE>
    <FILE_RDA file_name="generalized_rules.xml"/>
    <FILE_HIERARCHY file_name="hierarchy.xml"/>
  </PRESERVED_RULES>
</KDD_QUERY>

```

Questa DTD modella la query che prevede di ottenere un sottoinsieme di regole associative controllando che nel secondo insieme di regole ci siano quelle riferite agli stessi elementi ma con un livello di astrazione superiore. L'elemento PRESERVED_RULES ha infatti tre elementi così come è definito dalla DTD :

```
<!ELEMENT PRESERVED_RULES (%kdd_query_rules;,%kdd_query_rules;,  
                             HIERARCHY_FILE)>  
<!ATTLIST PRESERVED_RULES xml_dest CDATA #REQUIRED>
```

3.2.4 Classi della gerarchia del Sistema: DATA_SOURCES

Gli elementi Data_Sources sono rappresentazioni di dataset sotto forma di tabelle. Le sorgenti dei dati possono essere :

- file in formato .ARFF,
- file XML,
- Tabelle SQLServer 2000 o Oracle,
- file prolog,

che fisicamente vengono tradotti in un file .DATA e un file .XML. Il formato .ARFF è il formato proprietario di WEKA (Waikato Environment for Knowledge Analysis) [32]. Essi sono infatti files di testo ASCII in cui le tuple sono separate da un carattere di caporiga, e i valori all'interno di una singola tupla sono separati da una virgola. Nei files .ARFF è prevista anche un'intestazione che contiene il nome ed il tipo degli attributi; informazioni che sono indispensabili per applicare gli algoritmi di data mining. Gli attributi in WEKA possono assumere tre diversi tipi: numeric, enumerated, string. Un file .ARFF inizia dunque con una riga in cui si dichiara il nome della relazione, attraverso la specifica *@relation nome*. Le righe successive sono del tipo *@attribute nome tipo*, ed indicano il nome ed il tipo degli attributi che compongono la relazione. Infine seguono i dati che costituiscono il contenuto della base di dati, i quali sono preceduti dalla parola chiave *@data*.

<pre> @relation PlayTennis @attribute Outlook Sunny, Overcast, Rainy @attribute Temperature numeric @attribute Humidity numeric @attribute Windy True, False @attribute PlayTennis Yes, No </pre>
<pre> @data Sunny,85,85,False,No Sunny,80,90,True,No Overcast,83,86,True,Yes Rainy,70,96,False,Yes </pre>

Tabella 3.4: Data_Sources : Esempio di file in formato .ARFF.

(vedi esempio in tab.3.4). A partire dalle tabelle nel formato originario della sorgente, in KDDML-MQL si passa alla loro rappresentazione mediante un file .XML e uno .DATA per lo schema dei dati e per i dati , rispettivamente. Un esempio della rappresentazione delle tabelle è illustrato nella tab. 3.5.

3.3 Architettura del Sistema KDDML-MQL

Il Ssistema è implementato in codice Java e organizzato in packages, uno per ogni componente della struttura. Il Sistema si pone fra l'utente, che interagisce per mezzo di un'interfaccia grafica, e la sorgente dei dati. L'architettura (schematizzata in fig.3.2) è strutturata a livelli, ciascuno dei quali implementa una funzionalità specifica e fornisce un'interfaccia al livello superiore.

3.3.1 Repository layer

A livello più basso della gerarchia si trova il livello del sistema che si fa carico di:

schema	dati
<pre> <?xml version=1.0 encoding=UTF-8 ?> <!DOCTYPE KDDML_TABLE (View Source for full doctype...)> - <KDDML_TABLE> <SCHEMA name=weather file_name=take_by_if.xml numberOfAttributes=5 numberOfInstances=15> - <ATTRIBUTE name=outlook numberOfMissedValues=2 numberOfMissedValuesPerc=13% type=nominal> <VALUE value=sunny cardinality=5 cardinalityPerc=33% /> <VALUE value=overcast cardinality=4 cardinalityPerc=27% /> <VALUE value=rainy cardinality=4 cardinalityPerc=27% /> </ATTRIBUTE> - <ATTRIBUTE name=temperature numberOfMissedValues=0 numberOfMissedValuesPerc=0% type=numeric mean=74.0 variance=42.857142857142854 /> - <ATTRIBUTE name=humidity numberOfMissedValues=3 numberOfMissedValuesPerc=20% type=numeric mean=80.0 variance=88.36363636363636 /> - <ATTRIBUTE name=windy numberOfMissedValues=5 numberOfMissedValuesPerc=33% type=nominal> <VALUE value=TRUE cardinality=0 cardinalityPerc=0% /> <VALUE value=FALSE cardinality=10 cardinalityPerc=67% /> </ATTRIBUTE> - <ATTRIBUTE name=play numberOfMissedValues=0 numberOfMissedValuesPerc=0% type=nominal> <VALUE value=yes cardinality=10 cardinalityPerc=67% /> <VALUE value=no cardinality=5 cardinalityPerc=33% /> </ATTRIBUTE> </SCHEMA> </KDDML_TABLE> </pre>	<pre> sunny,85,85,?,no sunny,80,90,FALSE,no overcast,83,86,FALSE,yes rainy,70,?,FALSE,yes rainy,68,80,FALSE,yes ?,65,70,?,no overcast,64,65,?,yes sunny,72,95,FALSE,no sunny,69,?,FALSE,yes rainy,75,80,?,yes sunny,75,70,FALSE,yes overcast,72,90,FALSE,yes overcast,81,75,?,yes rainy,71,?,FALSE,no ?,80,74,FALSE,yes </pre>

Tabella 3.5: Data_Sources : Esempio del file .XML e .DATA per la rappresentazione di una tabella.

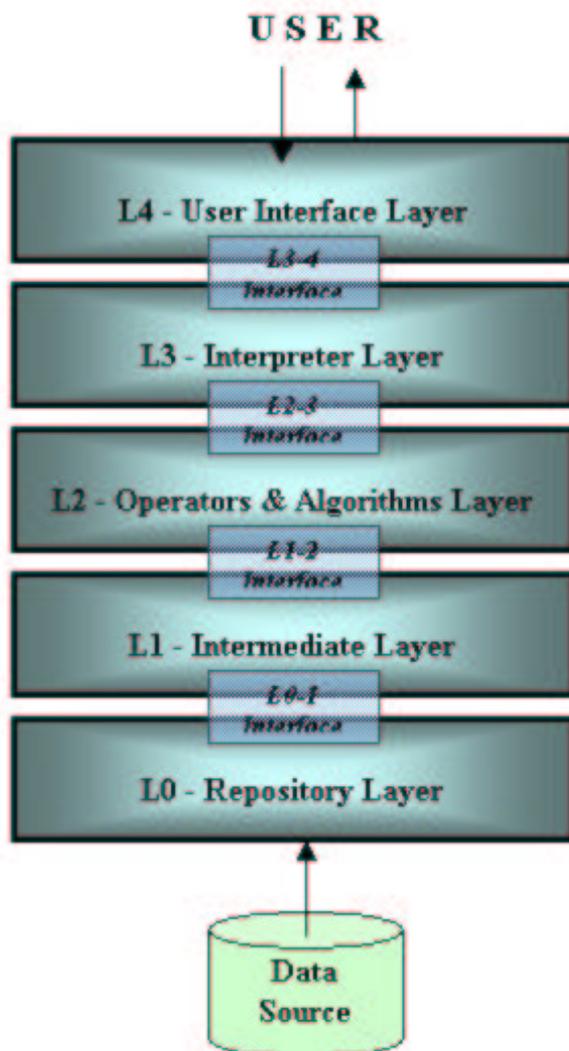


Figura 3.2: Struttura del sistema KDDML-MQL.

- sia della traduzione dei dati nei diversi formati sorgenti in rappresentazioni interne (.XML e .DATA) dei dati e del loro schema,
- che del salvataggio, dei file creati, nel repository.

3.3.2 Intermediate layer

Il livello intermedio della struttura del Sistema si preoccupa del passaggio da rappresentazioni fisiche (.XML e .DATA) a rappresentazioni interne e viceversa. Le rappresentazioni interne sono la descrizione del *KDDMLObject* (vedi fig.3.1) mantenuto in memoria principale durante l'esecuzione di operazioni sui dati. Questo livello, da una parte trasferisce i dati e gli schemi dal repository verso le altre componenti del Sistema, mentre dall'altra riceve le rappresentazioni interne di tabelle e modelli dall'interprete e le memorizza nel repository rispettando la loro rappresentazione fisica. Per le rappresentazioni interne delle tabelle, KDDML fa uso del pacchetto di WEKA [32] *weka.core*, mentre i modelli si rappresentano come alberi DOM [26]. Nel dettaglio, le funzionalità offerte dal livello intermedio sono:

- traduce un insieme di istanze del data repository (.xml e .data), nella loro rappresentazione interna;
- prende la rappresentazione interna di un insieme di istanze ottenuta dal query executor e memorizza i dati nel repository;
- traduce un modello PMML del Repository Layer in un albero DOM;
- traduce una rappresentazione DOM di un modello ottenuto dal query executor, in formato XML per la rappresentazione dello schema dei dati nel repository;
- fornisce un'interfaccia al livello superiore (Operators layer) che offre metodi ad alto livello per manipolare tabelle e modelli (parzialmente coperti dalla libreria di WEKA [32]).

Operatore MQL	Operatore KDDML
CreateRules	MINE_RULE
CreateGenRules	GENERALIZE
Filter-if	Non supportato → <i>Esteso</i>
PreservedRule	PRESERVED_RULES
CreateTree	MINE_TREE
ExecAnd	AND_TREE
ExecOr	OR_TREE
ExecComitato	COMMITTEE_OP
CreateCluster	MINE_CLUSTER
CreateCluster-Using	Non supportato → <i>Esteso</i>
Take-by-if	Non supportato → <i>Esteso</i>
Use-with	CLASSIFY
RuleSupport	RULE_SUPPORT
RuleException	RULE_EXCEPTION
ClusterNumber	N_CLUSTER_OF
ClusterMax	MAX_OF
Misclassified	Non supportato → <i>Esteso</i>
ExtractAll	Non supportato → <i>Esteso</i>

Tabella 3.6: KDDML-MQL : estensione del sistema KDDML sulla base degli operatori definiti in MQL.

3.3.3 Operators and algorithms layer

Il livello *Operators and algorithms* comprende le implementazioni degli operatori e degli algoritmi di supporto ad alcune fasi del processo KDD, e definiti dal linguaggio del sistema KDDML-MQL. L'interprete del Sistema invoca l'esecuzione dell'operatore necessario e ne riceve il risultato nella sua rappresentazione logica (*KDDMLObject*). Le fasi del KDD supportate dagli operatori e algoritmi implementati in KDDML-MQL sono quelle di Data Mining e Post-Processing. Gli operatori disponibili comprendono, oltre a quelli già presenti nella prima versione del sistema KDDML, gli operatori illustrati in fig.3.6 aggiunti successivamente in [23] sulla base del query language MQL definito in [24].

3.3.4 Interpreter layer

Il livello dell'interprete rappresenta la componente chiave della struttura di KDDML-MQL. E' a questo livello che avviene l'interazione fra l'albero DOM della KDD_QUERY, ottenuto mediante GUI oppure mediante la compilazione di una query in formato MQL, e la creazione del documento HTML relativo ai risultati. La classe centrale di questo livello è il *query executor* che lavora ricorsivamente sull'albero DOM della query, invoca gli operatori appropriati, recupera i risultati, ne testa la validità e la consistenza. L'interprete è strutturato in modo da non dover apportare modifiche per l'aggiunta di nuovi operatori, modelli o algoritmi. I metodi invocati per la risoluzione degli operatori sono progettati in modo da restituire un oggetto valido per il sistema, rispettando un'interfaccia definita a livello intermedio (*kddml.core.KDDMLObject.java*). La chiamata al metodo relativo a ciascun operatore avviene mediante la tag del file XML relativo alla query in analisi, pertanto è sufficiente estendere le DTD delle KDD_QUERY con la definizione del nuovo costrutto e chiamare la classe Java che lo implementa con lo stesso identificatore utilizzato nelle DTD.

3.3.5 GUI layer

Il livello dell'interfaccia grafica si preoccupa dell'interazione con l'utente rendendo disponibili le seguenti funzionalità :

- aprire una KDD_QUERY esistente,
- creare una nuova query guidato da un editor che ne controlla la sintassi,
- eseguire una KDD_QUERY e vedere la visualizzazione dei risultati mediante browser come una pagina HTML grazie all'uso di XSL [27] per la conversione dei file XML.

Capitolo 4

Il Query Language DPL

Il linguaggio DPL (Data Preparation Language) presentato in [9] è un'estensione alla sintassi MQL [24], il linguaggio di interrogazione data mining sviluppato per il sistema KDDML-MQL [23]. Lo scopo della formalizzazione del DPL è stato quello di coprire tutte le funzionalità necessarie per supportare la fase di preprocessing del processo KDD.

4.1 Sintassi del DPL

La sintassi del query language MQL estesa al DPL è [9]:

```
MQL ::= begin query <QUERY> end query
<QUERY> ::= let <Q> in | <Q1>
<Q> ::= <Q1> <Q2>
<Q1> ::= <NAME> =
        {<RuleQuery>|<TreeQuery>|<ClusterQuery>|
        <TableQuery>| <PPTableQuery> }
<Q2> ::= ;<Q>|epsilon
```

```

<RuleQuery> ::=
    CreateRules {<T>|<TableQuery>}<AlgRules>|
    CreateGenRules <G> {<T>|(<TableQuery>)} <AlgGen>|
    filter {<R> | (<RuleQuery>)}if <cond>|
    PreservedRule {<R> |(<RuleQuery>)} <Rg> <G> <N>

<TreeQuery> ::= CreateTree { <T> | (<TableQuery> ) } <AlgTree> |
    Exec <Op>

<ClusterQuery> ::= CreateCluster {<T>| (<TableQuery> )}<AlgClus>

<TableQuery> ::= RdA2Table {<R> | (<RuleQuery>)} |
    take <Elem> by {<T>| (<TableQuery> ) } ( if <cond>)? |
    Use { <Tree> | (<TreeQuery> ) } With {<T>|(<TableQuery> )}|
    Rule{ <R> | (<RuleQuery> ) } Support {<T>| (<TableQuery> ) } |
    Rule{ <R>| (<RuleQuery> ) } Exception{<T> | (<TableQuery> )}|
    Cluster { <C> | (<ClusterQuery> ) } Number <N> |
    Max { <C> | (<ClusterQuery> ) } |
    Misclassified {<T>| (<TableQuery> )}|
    ExtractAll { <C> | (<ClusterQuery> )}|
    PPTable2Table { <P> | (<PPTaleQuery> ) }

<PPTableQuery> ::= <RewritingQuery> ( <restr> )? |
    <MarkingQuery>(<restr> )? |
    <DividingQuery> |
    <SplittingQuery> |
    <MergingQuery> |
    <RenameQuery> |
    <RemoveQuery> |
    <NewAttributeQuery> |
    <FoldingQuery> |

```

```

        <Discretization> |
        <Normalization> ( <restr> )? |
        <Sampling> |
        <SortingByFrequency> |
        <MarkDuplicates> |
        <MergeDuplicates> |
        <RemoveRow> <restr> |
        Table2PPTable {<T>| (<TableQuery>) }

*** Costrutto di riscrittura ***
<RewritingQuery> ::= APPLY RWRULES IN
<PPTable> TO <Attribute>
        BEGIN <Rewriting_rules> END

<Rewriting_rules> ::= <Rule> ; |<Rule>; <Rewriting_rules>
<Rule> ::= <RuleS> | <RuleM> | <RuleT>
<RuleS> ::= <Pattern> INTO
        <Statement> ( EXCEPTION (<Code>)?)
<Pattern> ::= <ER>

<Statement> ::= <Substitution> | IF(<rw_cond>) THEN <Statement>
        | IF (<rw_cond >) THEN <Statement> ELSE <Statement>
<Substitution> ::= 'stringa di sostituzione contenente variabili
        di raggruppamento $i e caratteri'
< rw_cond > ::= <rw_term> < compare_op > < rw_term > |
        (<rw_cond>) AND (<rw_cond>) |
        (<rw_cond>) OR (<rw_cond>) |
        NOT <rw_cond>
<rw_term> ::= 'una variabile di raggruppamento $i oppure una

```

```

        sequenza di caratteri'
<RuleM> ::= <Pattern> EXCEPTION (<Code >)?

<RuleT> ::= EVERY
        <Pattern> INTO
        <Substitution> (EXCEPTION (<Code>))?)?

*** Costrutto di marcatura ***
MarkingQuery> ::= MARK <Attribute> IN <PPTable>
        IF(<mark_cond>)(WITH CODE <Code>)?

<mark_cond> ::= <mark_term> <compare_op> <mark\_term> |
        (<mark_cond>) AND ( < mark_cond > ) |
        (<mark_cond>) OR ( < mark_cond > ) |
        NOT ( < mark_cond > )

<mark_term> ::= <term>

*** Costrutto di divisione ***
<DividingQuery> ::= DIVIDE <Attribute>
        IN <PPTable>
        IF(<div_cond>)PUT INTO <NewAttribute>

<div_cond> ::= <compare_op> <div_term> , < div_term > |
        MATCH <ER> |
        NOT < div_cond > |
        (<div_cond >) (AND|OR) (<div_cond >)

<div_term> ::= SOURCE |<constant_term>

*** Costrutto di split ***

```

```
<SplittingQuery> ::= SPLIT <Attribute> IN  
                    <PPTable> AT ( <ER> )  
                    PUT INTO <NewAttribute> AND <NewAttribute>  
                    (EXCEPTION (<Code>))?
```

*** Costrutto di fusione ***

```
<MergingQuery> ::= MERGE <Attribute> AND <Attribute> IN <PPTable>  
                    PUT INTO <NewAttribute>
```

*** Costrutto di rinominazione ***

```
<RenameQuery> ::=  
                    RENAME<Attribute> IN <PPTable>  
                    AS <NewAttribute>
```

*** Costrutto di rimozione ***

```
<RemoveQuery> ::= REMOVE <Attribute> IN <PPTable>
```

*** Costrutto di creazione di un nuovo attributo ***

```
<NewAttributeQuery> ::= NEWATTRIBUTE <NAME> IN <PPTable>  
                    ( IS <newAtt_exp> )?
```

```
<newAtt_exp> ::= <Expression> | COPY OF<Attribute>
```

*** Costrutto di ripiegamento ***

```
<FoldingQuery> ::= FOLD <Attribute> ON <Attribute>  
                    IN <PPTable>
```

*** Costrutti di discretizzazione numerica, categorica ***

```
<Discretization> ::= <Discret_num> | <Discret_categ>
```

```

<Discret_num> ::= DISCRETIZE (EQUIWIDTH | EQUIDDEPTH) <Attribute>
                IN <PPTable> WITH (WIDTH <N> | DEPTH <N> | <N> INTERVALS)
                SMOOTH BY (MEAN | MEDIAN | INF | SUP | <Enumeration>)
                (EXCEPTION((<Code>))?)?

<Discret_categ> ::= ( SPECIFY ORDERING {<T>| (<TableQuery>)} ON
                    <Discret\_num> ) |
                    ( SPECIFY HIERARCHY {<T>| (<TableQuery>)} ON
                    DISCRETIZE <Attribute> IN <PPTable> EXCEPTION (<Code>)? )

<Enumeration> ::= {<NAME> <Enum>}

<Enum> ::= ,<NAME> <Enum>|epsilon

*** Costrutto di normalizzazione ***

<Normalization> ::= NORMALIZE <norm_method> ( <attribute_seq> )
                  IN <PPTable> ( EXCEPTION ( <Code> )?)?

<norm_method> ::= ( MIN\_MAX <Re> <Re> ) | Z\_SCORE | DECIMAL |
                  ( MIN\_MAX\_L <Re> <Re> <Re> <Re> ) |
                  ( Z\_SCORE\_L <Re> <Re> )

<attribute_seq> ::= ( <Attribute> , <attribute_seq> )|
                  <Attribute>

*** Costrutto di campionamento ***

<Sampling> ::= SAMPLE <PPTable> <sample_method> USE
              <sample_algorithm>

<sample_method> ::= SIMPLE | BY CLUSTER <Attribute> |
                  STRATIFIED ON <Attribute>

<sample_algorithm> ::= (SRSWOR | SRSWR)

```


`<term_seq> ::= <term> , <term_seq> | <term>`

`<term> ::= = ATTRIBUTE <Attribute> |
 <Expression> |
 <constant_term>`

`<compare_op> ::= una qualunque operazione di confronto tra
 stringhe oppure numeri`

`<constant_term> ::= string | real | integer`

`<attribute_seq> ::= (<Attribute> , <attribute_seq>) |
 <Attribute>`

`<AlgRules> ::= <Apriori> <Re> <Re> <N>`

`<AlgGen> ::= <GenApriori> <Re> <Re> <N>`

`<AlgTree> ::= C4.5 <Re> <N> | ID3`

`<AlgClus> ::= EM <N> <N> | <NAME>`

`<Rg> ::= <R> | CreateGenRules <G> {<T> | (<TableQuery>)} <AlgGen>`

`<Op> ::= And ({ <Tree> | (<TreeQuery>) }, { <Tree> |
 (<TreeQuery>)}) |
 Or ({ <Tree> | (<TreeQuery>)}, { <Tree> | (<TreeQuery>)}) |
 Comitato({ <Tree> | (<TreeQuery>) }, { <Tree> | (<TreeQuery>)})`

Data
25/10/78
2/17/81
20th March 2003
8.09.2000
in data 2/7/18
08/19/1950
32.05.75
8/07/328

Tabella 4.1: Esempio: dati da uniformare nel formato mediante operatori del DPL.

4.1.1 Esempi guida dell'effetto dei costrutti del DPL

Uniformazione del formato

Quando le fonti sono eterogenee si ha la necessità di rendere uniformi i dati nel formato. Supponiamo ad esempio di avere un attributo contenente le date che si presentano come in tab. 4.1 [9]

Per rendere uniforme la rappresentazione delle date si procede applicando le seguenti trasformazioni:

1. Si eliminano caratteri di spaziatura e sequenze di caratteri non significativi adiacenti agli estremi della data; si sostituiscono caratteri di punteggiatura con il carattere “/”; si aggiunge uno zero al giorno e al mese quando la data non è nel formato xx/yy/“anno”. In DPL questo corrisponde a eseguire le seguenti regole di riscrittura (di tipo Rule_S):

```

APPLY RWRULES IN 'tabella' TO 'data'
BEGIN

<< \D* (\d{1,2}) [./] (\d{1,2}) [./] (\d{1,4}) \D* >>
INTO $1/$2/$3 EXCEPTION;

```

```

<< (\d) / ([\d/]*) >>
INTO 0$1/$2 ;
<< (\d\d) / (\d) / ([\d/]*) >>
INTO $1/0$2/$3 ;
<< (\d\d) / (\d\d) / (\d) >>
INTO $1/$2/0$3 ;

```

2. Si modifica la data “xx/yy/aa” in “yy/xx/aa” quando $yy > 12$. In DPL questo corrisponde a eseguire la seguente regole di riscrittura (di tipo Rule_S) in cui lo statement di sostituzione ha una condizione “*if..then..*”:

```

...

<< ( \d{2}) / ( \d{2}) / ((\d\d){1,2}) >>
INTO IF( $2 > 12 )
      THEN $2/$1/$3 EXCEPTION;

```

3. Quando la data è nel formato “xx/yy/aa” , se $aa > 30$ la si cambi in “xx/yy/19aa” e se $aa \leq 30$ la si cambi in “xx/yy/20aa”. In DPL questo corrisponde a eseguire la seguente regole di riscrittura (di tipo Rule_S) in cui lo statement di sostituzione ha una condizione “*if..then..else*”:

```

...

<< ( \d{2}/ \d{2}/ ) (\d{2}) >>
INTO IF( $2 > 30 ) THEN $119$3
      ELSE $120$3

```

Nominativo	Indirizzo
J. Smith	AddressA
Smith, John	AddressA
john smith	AddressA
Smith J.	AddressB
Smith L.	AddressB

Tabella 4.2: Esempio: dati da de-duplicare mediante operatori del DPL.

4. Infine controllare la data rispettando sempre il formato “xx/yy/aaaa” e che $xx \leq 31$. In DPL questo corrisponde a eseguire la seguente regola di riscrittura (di tipo Rule_M) in cui si marca con un’eccezione i valori che non rispettano il formato rappresentato dall’espressione regolare specificata:

```

...

<< ([0-2] [0-9] [\d/]* ) | (3 [0,1] [\d/]* ) >> EXCEPTION ;
END

```

L’effetto di tutti questi passi di riscrittura sui valori iniziali dell’attributo “Data” sono illustrati in fig. 4.1.

Trattamento dei duplicati

In presenza della necessità dell’individuazione dei duplicati, ad esempio, è possibile sfruttare le potenzialità del DPL per procedere nella loro individuazione e risoluzione secondo le modalità già viste in 2.1.6.

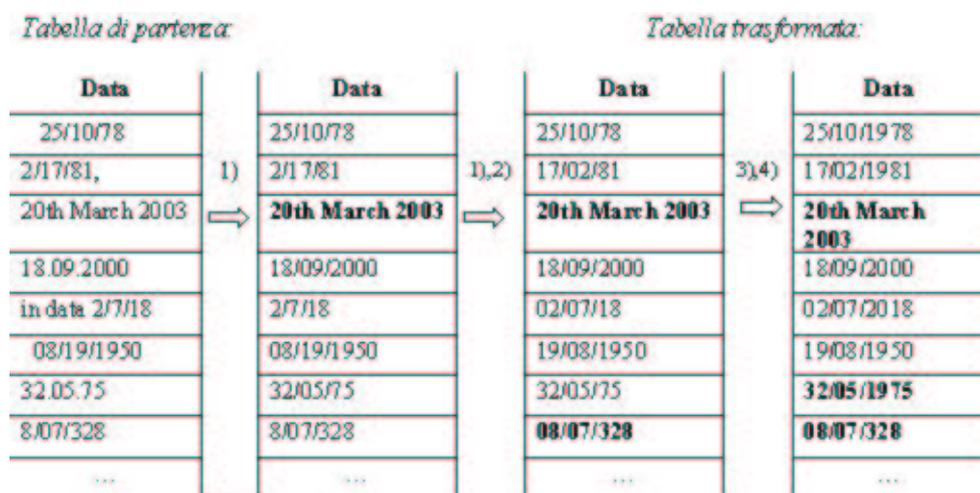


Figura 4.1: Esempio di uniformazione dei formati dell’attributo “Data” mediante operatori del DPL [9].

Supponiamo, ad esempio, di avere a disposizione i dati di tab.4.2 e di volere individuare i duplicati usando come chiave gli attributi “Nominativo” e “Indirizzo”. Innanzitutto si procede con l’applicazione di regole di riscrittura per uniformare il formato delle lettere (minuscole e maiuscole) e anticipare il nome al cognome, a questo punto si procede come illustrato in fig.4.2. Grazie alle operazioni di trasformazione applicate, possiamo decidere come plausibile la scelta di individuare i duplicati sulla base dell’iniziale del nome e del cognome piuttosto che sul “Nominativo”. A seguito della prima unificazione dei duplicati fatta strettamente sulla chiave prefissata (“Nominativo” + “Indirizzo”), siamo ancora in presenza di valori sulle istanze che

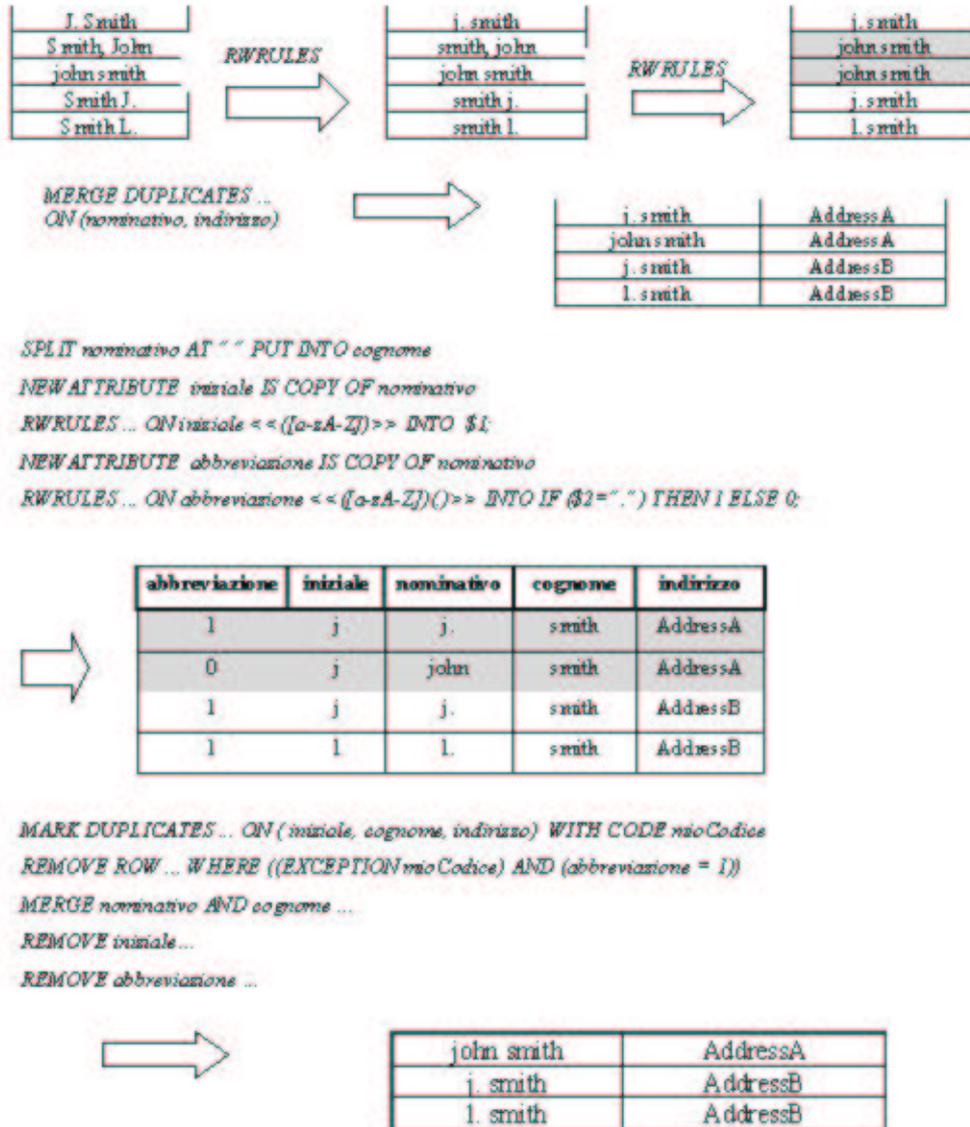


Figura 4.2: Esempio di eliminazione di duplicati mediante operatori del DPL [9].

possibilmente rappresenteranno informazioni duplicate a meno della rappresentazione eterogenea del nome (per esteso oppure come abbreviazione). Si passa, quindi, all'individuazione dei duplicati sulla base della nuova chiave "iniziale" + "cognome" + "indirizzo", marcandoli con un codice. Se reputato opportuno, a questo punto si eliminano le righe relative alle istanze appena marcate come duplicate e per le quali il nominativo si presenta come abbreviazione (per mantenere l'istanza più informativa).

4.2 DPL nel Sistema KDDML-MQL

Il linguaggio DPL è costituito principalmente da costrutti che rappresentano delle trasformazioni di tabelle. Gli ingressi e le uscite di tali trasformazioni sono oggetti di tipo "Tabella Estesa" (PPTable). L'espressione principale del linguaggio DPL è la tabella estesa, che rappresenta fondamentalmente una tabella relazionale. Rispetto alla tabella relazionale definita nel linguaggio MQL, essa ne forma un'estensione che contiene delle meta-informazioni riguardanti la storia delle trasformazioni subite dalla tabella, e contemporaneamente realizza il meccanismo di marcatura delle eccezioni. L'estensione della tabella è realizzata aggiungendo un campo informativo per ogni attributo, ossia raddoppiando il numero di colonne della tabella come in fig. 4.3. In KDDML-MQL [23] è già stato fornito un livello di astrazione più alto per la definizione di query nel formalismo MQL, che risulta più agevole per l'utente rispetto all'XML necessario per il Sistema KDDML [14]. Gli operatori MQL sono classificabili in base alla tipologia di conoscenza estratta in :

TreeQuery: operatori che restituiscono un albero di classificazione.

RuleQuery: operatori che restituiscono regole di associazione.

Attr 1	Attr 2	Attr 3	Attr 4				
						<i>Metadati</i>	

Figura 4.3: Tabelle Estese nel DPL.

ClusterQuery: operatori che restituiscono un oggetto di tipo cluster.

TableQuery: operatori che restituiscono una tabella di un database

con l'estensione del DPL vengono aggiunti:

PPTableQuery: operatori che restituiscono tabelle estese.

Ciascun operatore che opera su PPTable mira a risolvere problemi di qualità dei dati a livello di schema o a livello di istanza illustrate in 2.1 . In particolare DPL modella funzionalità quali :

- Rimozione di attributi;
- Rinominazione di attributi;
- *Split* e *Fold* dei valori assunti da un attributo;
- Aggiunta di nuovi attributi;
- *Merge* di valori di attributi;

che modificano lo schema dei dati, e operazioni di :

- Normalizzazione;
- Discretizzazione;
- Ordinamento in base alla frequenza;
- Campionamento;
- Marcatura e Unificazione dei duplicati;
- Marcatura e Riscrittura di valori che verificano delle condizioni;
- Rimozione di righe della tabella ;

che modificano le singole istanze dei dati. Con questo lavoro (vedi fig.4.4) il Sistema KDDML-MQL è stato esteso alla fase di PreProcessing in modo completo operando :

- sulla componente del compilatore interposta tra le query nel formato DPL e il linguaggio KDDML,
- nel linguaggio KDDML con la definizione di un nuovo tipo che modella le tabelle estese e di nuovi tipi di query che operano su di esse ,
- nel supporto a runtime, sia con l'estensione delle strutture dati per permettere la manipolazione delle tabelle estese, sia con l'introduzione di nuovi operatori per il calcolo e la generazione dei risultati delle nuove query introdotte.

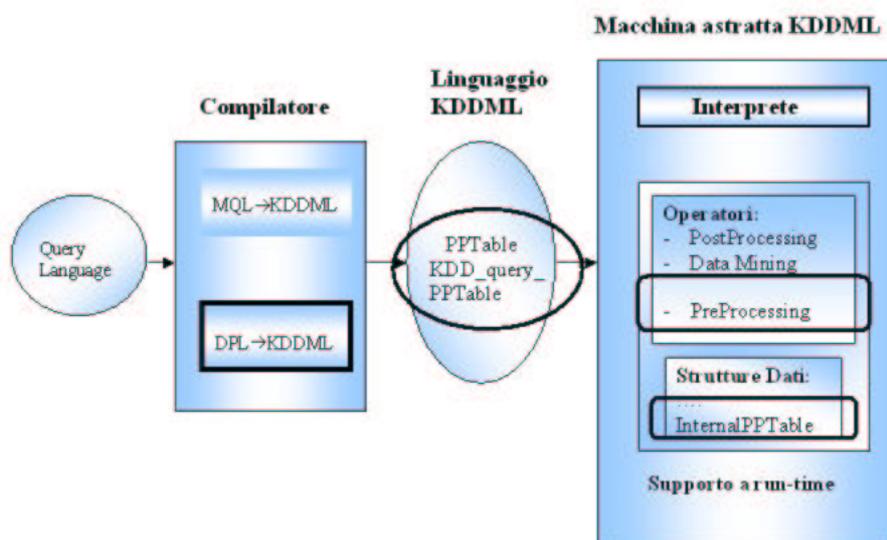


Figura 4.4: Schematizzazione dei contributi della tesi.

Capitolo 5

Estensione del linguaggio KDDML e del supporto a run-time

5.1 Estensione del linguaggio KDDML

Per realizzare l'estensione del query language al DPL proposta in [9] è stato necessario innanzitutto modificare il linguaggio KDDML introducendo:

- un nuovo tipo di conoscenza estratta che rappresenta le tabelle estese (*PPTable*)
- nuovi tipi di query che operano su *PPTable* (*KDD_query_PPTable*)

Per la rappresentazione delle *PPTable* è stata estesa la definizione delle tabelle nelle DTD 5.1. Una tabella del linguaggio KDDML, quindi, è composta da tre elementi :

- schema dei dati,

- schema dei metadati (opzionale),
- storia delle operazioni di preprocessing applicate alla tabella (opzionale).

La presenza degli ultimi due elementi opzionali implica che la natura della tabella è “PPTable”. Fisicamente le tabelle del KDDML sono rappresentate da un file .XML secondo la DTD descritta , e due file di testo “file_name”, contenenti i valori dei campi relativi agli attributi dei metadati (.METADATA) e dei dati (.DATA).

5.1.1 Gli operatori DPL nel linguaggio KDDML

Il linguaggio KDDML è stato esteso per includere le query descritte dal formalismo DPL, pertanto nelle KDD_QUERY del Sistema, descritte in fig.3.3, è stato definito il tipo *KDD_query_PPTable*. Ciascun operatore ha la corrispondente definizione del tipo nel linguaggio KDDML. Rispetto alla sintassi del DPL sono state apportate alcune modifiche di forma, per ottenere una rappresentazione che fosse il più uniforme possibile di tutte le query, e sono stati aggiunti alcuni elementi per incrementarne le funzionalità. Sono state specificate, quindi, le DTD per ciascun nuovo operatore di preprocessing grazie alle quali è possibile, a partire da un file .XML, riconoscere se è una *kdd_query_PPTable*, e pertanto considerarlo come un tipo valido per il linguaggio KDDML. La validità è data dall’esito del parsing tra il file .XML e le DTD, parsing realizzato sfruttando il package *javax.xml.parsers*. Di seguito è riportata nel dettaglio la definizione di ogni *KDD_query_PPTable* che compone il linguaggio KDDML:

DTD Table
<pre> <?xml encoding=UTF-8?> <!-- Definizione degli elementi, e relativi attributi, per la rappresentazione di un insieme di tuple di una tabella di un DataBase. --> <!ELEMENT KDDML_TABLE (SCHEMA, (METASHEMA)?,(HISTORY)?)> <!ELEMENT SCHEMA (ATTRIBUTE+)> <!ATTLIST SCHEMA name CDATA # REQUIRED> <!ATTLIST SCHEMA numberOfAttributes CDATA # REQUIRED> <!ATTLIST SCHEMA numberOfInstances CDATA # REQUIRED> <!ATTLIST SCHEMA file_name CDATA # REQUIRED> <!ELEMENT ATTRIBUTE (VALUE)*> <!ATTLIST ATTRIBUTE name CDATA # REQUIRED> <!ATTLIST ATTRIBUTE type (numeric nominal string) # REQUIRED> <!ATTLIST ATTRIBUTE numberOfMissedValues CDATA # REQUIRED> <!ATTLIST ATTRIBUTE numberOfMissedValuesPerc CDATA # REQUIRED> <!ATTLIST ATTRIBUTE mean CDATA # IMPLIED> <!ATTLIST ATTRIBUTE variance CDATA # IMPLIED> <!ELEMENT VALUE EMPTY> <!ATTLIST VALUE value CDATA # REQUIRED> <!ATTLIST VALUE cardinality CDATA # REQUIRED> <!ATTLIST VALUE cardinalityPerc CDATA # REQUIRED> <!ELEMENT METASHEMA (ATTRIBUTE+)> <!ATTLIST METASHEMA name CDATA # REQUIRED> <!ATTLIST METASHEMA numberOfAttributes CDATA # REQUIRED> <!ATTLIST METASHEMA numberOfInstances CDATA # REQUIRED> <!ATTLIST METASHEMA file_name CDATA # REQUIRED> <!ELEMENT HISTORY (PREPROCESSING_OPERATION*)> <!ELEMENT PREPROCESSING_OPERATION EMPTY> <!ATTLIST PREPROCESSING_OPERATION Operation CDATA # REQUIRED> <!ATTLIST PREPROCESSING_OPERATION Description CDATA # REQUIRED> </pre>

Tabella 5.1: DTD che descrivono un tipo tabella o tabella estesa in KDDML.

PPTABLE2TABLE

representa la chiamata ad un'operatore che trasforma una tabella estesa in una tabella

```
<!ELEMENT PPTABLE2TABLE (%kdd_query_PPtable;)>
<!ATTLIST PPTABLE2TABLE xml_dest CDATA #IMPLIED>
```

TABLE2PPTABLE

representa la query che mira a ottenere la trasformazione di una tabella in una tabella estesa

```
<!ELEMENT TABLE2PPTABLE (%kdd_query_table;)>
<!ATTLIST TABLE2PPTABLE xml_dest CDATA #IMPLIED>
```

PPREMOVE_QUERY

representa la query che mira a eliminare dalla tabella estesa uno o piu' attributi

```
<!ELEMENT PPREMOVE_QUERY ((%kdd_query_PPtable;),ATTRIBUTE_SEQ)>
<!ATTLIST PPREMOVE_QUERY xml_dest CDATA #IMPLIED>
<!ELEMENT ATTRIBUTE_SEQ ((ATTR)+)>
```

PPRENAME_QUERY

representa la query che mira a ottenere la tabella estesa con uno o piu' attributi rinominati

```
<!ELEMENT PPRENAME_QUERY ((%kdd_query_PPtable;),SUBSTITUTIONS)>
<!ATTLIST PPRENAME_QUERY xml_dest CDATA #IMPLIED>
```

```

<!ELEMENT SUBSTITUTIONS ((RENOMINATION)+)>
<!ELEMENT RENOMINATION EMPTY>
<!ATTLIST RENOMINATION old_Name CDATA #REQUIRED>
<!ATTLIST RENOMINATION new_Name CDATA #REQUIRED>

```

PPNEWATTRIBUTE_QUERY

rappresenta la query che mira a aggiungere in una tabella estesa un nuovo attributo con nome e tipo specificati, i valori assunti dall'attributo su ciascuna istanza dei dati della tabella sono calcolati come:

- valore costante;
- copia dei valori assunti da un altro attributo;
- risultato di un'espressione numerica o su stringhe.

```

<!ELEMENT PPNEWATTRIBUTE_QUERY ((%kdd_query_PPtable;),
    NEWATTR,(NEWATT_EXP)?)>
<!ATTLIST PPNEWATTRIBUTE_QUERY xml_dest CDATA #IMPLIED>
<!ELEMENT NEWATTR EMPTY>
<!ATTLIST NEWATTR attribute_name CDATA #REQUIRED
    attribute_type (enumerated|string|real)
    #REQUIRED>
<!ELEMENT NEWATT_EXP (TERMINE)>
<!ELEMENT EXPR (TERM_OPERATION,TERM_SEQ)>
<!ELEMENT TERM_OPERATION (STRING_OPERATION|NUMERIC_OPERATION)>
<!ELEMENT STRING_OPERATION EMPTY>
<!ATTLIST STRING_OPERATION op_type (concat|equal) #REQUIRED>
<!ELEMENT NUMERIC_OPERATION EMPTY>
<!ATTLIST NUMERIC_OPERATION op_type

```

```

                (sum|multiply|subtract|fract|module) #REQUIRED>
<!ELEMENT TERM_SEQ (TERMINE+)>
<!ELEMENT TERMINE (ATTR|EXPR|CONSTANT)>
<!ELEMENT CONSTANT EMPTY>
<!ATTLIST CONSTANT valore CDATA #REQUIRED>

```

PPMERGING_QUERY

rappresenta la query che mira a ottenere l'unificazione dei valori assunti da uno o piu' attributi in un nuovo attributo

```

<!ELEMENT PPMERGING_QUERY ((%kdd_query_PPtable;),
                ATTRIBUTE_SEQ,NEW_ATTR_NAME)>

<!ATTLIST PPMERGING_QUERY xml_dest CDATA #IMPLIED>
<!ELEMENT NEW_ATTR_NAME (ATTR)>

```

PPSORTING_BY_FREQUENCY

rappresenta la query che mira a ottenere l'ordinamento delle tuple e metatuple della tabella estesa sulla base della frequenza dei valori assunti di un attributo specificato

```

<!ELEMENT PPSORTING_BY_FREQUENCY ((%kdd_query_PPtable;),ATTR)>

<!ATTLIST PPSORTING_BY_FREQUENCY xml_dest CDATA #IMPLIED>

```

PPFOLDING_QUERY

rappresenta la query che mira a ottenere il ripiegamento dei valori di un attributo su un'altro attributo della tabella estesa

```

<!ELEMENT PPFOLDING_QUERY ((%kdd_query_PPtable;),ATTR,ATTR)>

```

```
<!ATTLIST PPFOLDING_QUERY xml_dest CDATA #IMPLIED>
```

PPDIVIDING_QUERY

rappresenta la query che mira a spostare i valori di un attributo della tabella estesa su un nuovo attributo, al verificarsi di una condizione

```
<!ELEMENT PPDIVIDING_QUERY ((%kdd_query_PPtable;),ATTR,
    NEW_ATTR_NAME,DIV_COND)>
```

```
<!ATTLIST PPDIVIDING_QUERY xml_dest CDATA #IMPLIED>
```

```
<!ELEMENT DIV_COND
```

```
    (OR_DIV|NOT_DIV|AND_DIV|REL_DIV|
    COMPARE_STRING_DIV|MATCH_ER_DIV)>
```

```
<!ELEMENT OR_DIV (DIV_COND,DIV_COND)>
```

```
<!ELEMENT NOT_DIV (DIV_COND)>
```

```
<!ELEMENT AND_DIV (DIV_COND,DIV_COND)>
```

```
<!ELEMENT REL_DIV (DIV_TERM,DIV_TERM)>
```

```
<!ATTLIST REL_DIV type
```

```
    (less|less_or_equal|greater|
    greater_or_equal|equal|not_equal)
    #REQUIRED>
```

```
<!ELEMENT COMPARE_STRING_DIV (DIV_TERM,DIV_TERM)>
```

```
<!ELEMENT MATCH_ER_DIV (DIV_TERM,REGULAR_EXPRESSION)>
```

```
<!ELEMENT DIV_TERM (SOURCE|CONSTANT)>
```

```
<!ELEMENT SOURCE EMPTY>
```

```
<!ELEMENT REGULAR_EXPRESSION EMPTY>
```

```
<!ATTLIST REGULAR_EXPRESSION Pattern CDATA #REQUIRED>
```

PPSPLITTING_QUERY

rappresenta la query che mira a spezzare i valori assunti da un attributo mettendo ciascuno dei due pezzi ottenuti sui rispettivi due nuovi attributi specificati. Il cutoff e' rappresentato da un' espressione regolare

```
<!ELEMENT PPSPLITTING_QUERY ((%kdd_query_PPtable;),ATTR,
    NEW_ATTR_NAME,NEW_ATTR_NAME,
    REGULAR_EXPRESSION,(EXCEPTION)?)>
```

```
<!ATTLIST PPSPLITTING_QUERY xml_dest CDATA #IMPLIED>
```

```
<!ELEMENT EXCEPTION EMPTY>
```

```
<!ATTLIST EXCEPTION marca CDATA #IMPLIED>
```

PPMARK_DUPLICATES

rappresenta la query che mira a aggiungere una marca su quegli attributi di istanze identificate come duplicate. L'individuazione dei duplicati e' basata su una chiave composta da uno o piu' attributi

```
<!ELEMENT PPMARK_DUPLICATES ((%kdd_query_PPtable;),
    ATTRIBUTE_SEQ,(WITH_CODE)?)>
```

```
<!ATTLIST PPMARK_DUPLICATES xml_dest CDATA #IMPLIED>
```

```
<!ELEMENT WITH_CODE EMPTY>
```

```
<!ATTLIST WITH_CODE marca CDATA #IMPLIED>
```

PPMERGE_DUPLICATES

rappresenta la query che mira a unificare le istanze identificate come duplicate aggiungendo una marca per mantenere traccia dell'avvenuta unificazione. L'individuazione dei duplicati e' basata su una chiave composta da uno o piu' attributi

```
<!ELEMENT PPMERGE_DUPLICATES ((%kdd_query_PPtable;),  
    ATTRIBUTE_SEQ,(WITH_CODE)?)>
```

```
<!ATTLIST PPMERGE_DUPLICATES xml_dest CDATA #IMPLIED>
```

PPSAMPLING

rappresenta la query che mira a ottenere un insieme campionato della tabella estesa con o senza rimpiazzo secondo un metodo scelto fra:

- campionamento semplice;
- campionamento stratificato ;
- campionamento su cluster.

```
<!ELEMENT PPSAMPLING ((%kdd_query_PPtable;),SAMPLE_METHOD,  
    SAMPLE_ALGORITHM)>
```

```
<!ATTLIST PPSAMPLING xml_dest CDATA #IMPLIED>
```

```
<!ELEMENT SAMPLE_METHOD (SIMPLE|BY_CLUSTER|STRATIFIED_ON)>
```

```
<!ELEMENT SIMPLE EMPTY>
```

```
<!ELEMENT BY_CLUSTER (ATTR)>
```

```
<!ELEMENT STRATIFIED_ON (ATTR)>
```

```
<!ELEMENT SAMPLE_ALGORITHM (SRSWOR|SRSWR)>
```

```
<!ATTLIST SAMPLE_ALGORITHM parameter_type (items|index) #REQUIRED  
    parameter_value CDATA #REQUIRED>
```

```
<!ELEMENT SRSWOR EMPTY>
```

```
<!ELEMENT SRSWR EMPTY>
```

PPMARKING_QUERY

rappresenta la query che mira a aggiungere dei valori nei metadati in corrispondenza di attributi con valori che verificano una condizione, e opzionalmente su un sottoinsieme di istanze che verificano la condizione di restrizione

```
<!ELEMENT PPMARKING_QUERY ((%kdd_query_PPtable;),ATTR,
    MARK_COND,(WITH_CODE)?,(RESTR)?)>
```

```
<!ATTLIST PPMARKING_QUERY xml_dest CDATA #IMPLIED>
```

```
<!ELEMENT MARK_COND
```

```
    (OR_MARK|NOT_MARK|AND_MARK|REL_MARK|
    COMPARE_STRING_MARK|IS_MISSING_MARK)>
```

```
<!ELEMENT IS_MISSING_MARK (ATTR)>
```

```
<!ELEMENT OR_MARK(MARK_COND,MARK_COND)>
```

```
<!ELEMENT NOT_MARK (MARK_COND)>
```

```
<!ELEMENT AND_MARK (MARK_COND,MARK_COND)>
```

```
<!ELEMENT REL_MARK (MARK_TERM,MARK_TERM)>
```

```
<!ATTLIST REL_MARK type
```

```
    (less|less_or_equal|greater|
    greater_or_equal|equal|not_equal)
    #REQUIRED>
```

```
<!ELEMENT COMPARE_STRING_MARK (MARK_TERM,MARK_TERM)>
```

```
<!ELEMENT MARK_TERM (ATTR|CONSTANT)>
```

PPNORMALIZATION_LOCAL

rappresenta la query che mira a normalizzare i valori di uno o piu' attributi seguendo un metodo scelto fra:

- min-max;

- z-score;

basati su valori di:

- min e max, e
- media e deviazione standard

dell'intera tabella passati come parametri e non calcolati

```
<!ELEMENT PPNORMALIZATION_LOCAL ((%kdd_query_PPtable;),LOCAL_NORM_METHOD,
    ATTRIBUTE_SEQ,(EXCEPTION)?,(RESTR)?)>
```

```
<!ELEMENT LOCAL_NORM_METHOD (MIN_MAX_LOCAL|Z_SCORE_LOCAL)>
```

```
<!ELEMENT Z_SCORE_LOCAL EMPTY>
```

```
<!ATTLIST Z_SCORE_LOCAL media CDATA #REQUIRED>
```

```
<!ATTLIST Z_SCORE_LOCAL deviazione_standard CDATA #REQUIRED>
```

```
<!ELEMENT MIN_MAX_LOCAL EMPTY>
```

```
<!ATTLIST MIN_MAX_LOCAL min_Attuale CDATA #REQUIRED>
```

```
<!ATTLIST MIN_MAX_LOCAL max_Attuale CDATA #REQUIRED>
```

```
<!ATTLIST MIN_MAX_LOCAL min CDATA #REQUIRED>
```

```
<!ATTLIST MIN_MAX_LOCAL max CDATA #REQUIRED>
```

PPNORMALIZATION

rappresenta la query che mira a normalizzare i valori di uno o piu' attributi seguendo un metodo scelto fra:

- min-max;
- z-score;
- decimale

basati su valori di:

- min e max, e
- media e deviazione standard

calcolati sull'intera tabella

```
<!ELEMENT PPNORMALIZATION ((%kdd_query_PPtable;),NORM_METHOD,ATTRIBUTE_SEQ)>
```

```
<!ATTLIST PPNORMALIZATION xml_dest CDATA #IMPLIED>
```

```
<!ATTLIST PPNORMALIZATION_LOCAL xml_dest CDATA #IMPLIED>
```

```
<!ELEMENT NORM_METHOD (MIN_MAX|Z_SCORE|DECIMAL)>
```

```
<!ELEMENT MIN_MAX EMPTY>
```

```
<!ATTLIST MIN_MAX min CDATA #REQUIRED>
```

```
<!ATTLIST MIN_MAX max CDATA #REQUIRED>
```

```
<!ELEMENT Z_SCORE EMPTY>
```

```
<!ELEMENT DECIMAL EMPTY>
```

PPDISCRETIZATION

rappresenta la query che mira a discretizzare i valori di uno attributo della tabella estesa seguendo un metodo scelto fra:

- discretizzazione numerica
- discretizzazione categorica

```
<!ELEMENT PPDISCRETIZATION ((%kdd_query_PPtable;),ATTR,
(NUMERIC_DISCRETIZATION|CATEGORIC_DISCRETIZATION))>
```

```
<!ATTLIST PPDISCRETIZATION xml_dest CDATA #IMPLIED>
```

```

<!ELEMENT NUMERIC_DISCRETIZATION
      (NUMERIC_DISCRETIZATION_METHODS,SMOOTH_BY)>
<!ELEMENT NUMERIC_DISCRETIZATION_METHODS (EQUIWIDHT|EQUIDEPTH) >
<!ELEMENT EQUIWIDHT EMPTY>
<!ATTLIST EQUIWIDHT having(widht|intervals) #REQUIRED>
<!ATTLIST EQUIWIDHT value CDATA #REQUIRED>
<!ELEMENT EQUIDEPTH EMPTY>
<!ATTLIST EQUIDEPTH having (depth|intervals) #REQUIRED>
<!ATTLIST EQUIDEPTH value CDATA #REQUIRED>
<!ELEMENT SMOOTH_BY (MEAN|MEDIAN|INF|SUP|ENUMERATION)>
<!ELEMENT MEAN EMPTY> <!ELEMENT MEDIAN EMPTY>
<!ELEMENT INF EMPTY> <!ELEMENT SUP EMPTY>
<!ELEMENT ENUMERATION (EXCEPTION,(ENUMERATION_ELEMENT)*)>
<!ELEMENT ENUMERATION_ELEMENT EMPTY>
<!ATTLIST ENUMERATION_ELEMENT name CDATA #REQUIRED>
<!ELEMENT CATEGORIC_DISCRETIZATION (SPECIFY)>
<!ELEMENT SPECIFY ((ORDERING|HIERARCHY))>
<!ELEMENT ORDERING (SOURCE_ORDER,NUMERIC_DISCRETIZATION)>
<!ELEMENT SOURCE_ORDER (TABLE_WITH_ORDER|USE_NOMINAL_ORDER)>
<!ELEMENT USE_NOMINAL_ORDER EMPTY>
<!ELEMENT TABLE_WITH_ORDER ((%kdd_query_table;),EXCEPTION)>
<!ELEMENT HIERARCHY ((%kdd_query_table;),EXCEPTION)>

```

PPREWRITING

rappresenta la query che mira a riscrivere i valori di uno attributo della tabella estesa applicando una o piu' regole tra:

- rule-S (se mappa una ER o se e' missing allora riscrivo)
- rule-T (per ogni sottostringa del valore che mappa una ER, riscrivo)

- rule-M (se mappa una ER o se e' missing allora aggiungo un valore nei metadati)

Rispetto alla sintassi del DPL [9], il costrutto di riscrittura del linguaggio KDDML permette di individuare, marcare e riscrivere valori mancanti mediante la condizione di applicazione delle regole "IS_MISSING_RW" piuttosto che il solo match con un'espressione regolare.

```
<!ELEMENT PPREWRITING ((%kdd_query_PPtable;),ATTR,REWRITING_RULES,(RESTR)?)>
```

```
<!ATTLIST PPREWRITING xml_dest CDATA #IMPLIED>
```

```
<!ELEMENT REWRITING_RULES (RULE+)>
```

```
<!ELEMENT RULE (RULE_S|RULE_M|RULE_T)>
```

```
<!ELEMENT RULE_S
```

```
((REGULAR_EXPRESSION|IS_MISSING_RW),STATEMENT,(EXCEPTION)?)>
```

```
<!ELEMENT STATEMENT (SUBSTITUTION|IF_THEN_ELSE)>
```

```
<!ELEMENT IF_THEN_ELSE (RW_COND,STATEMENT,(STATEMENT)?)>
```

```
<!ELEMENT SUBSTITUTION EMPTY>
```

```
<!ATTLIST SUBSTITUTION substitution_string CDATA #REQUIRED>
```

```
<!ELEMENT RULE_M ((REGULAR_EXPRESSION|IS_MISSING_RW),EXCEPTION)>
```

```
<!ELEMENT RULE_T ((REGULAR_EXPRESSION),SUBSTITUTION,(EXCEPTION)?)>
```

```

<!ELEMENT RW_COND (OR_RW|NOT_RW|AND_RW|REL_RW|COMPARE_STRING_RW)>
<!ELEMENT IS_MISSING_RW EMPTY>
<!ELEMENT OR_RW (RW_COND,RW_COND)>
<!ELEMENT NOT_RW (RW_COND)> <!ELEMENT AND_RW (RW_COND,RW_COND)>
<!ELEMENT REL_RW (RW_TERM,RW_TERM)> <!ATTLIST REL_RW type
(less|less_or_equal|greater|greater_or_equal|equal|not_equal)
#REQUIRED>

```

```

<!ELEMENT COMPARE_STRING_RW (RW_TERM,RW_TERM)>

```

```

<!ELEMENT RW_TERM(RAGGR|CONSTANT)>
<!ELEMENT RAGGR EMPTY>
<!ATTLIST RAGGR group_variable CDATA #REQUIRED>

```

PPREMOVE_ROWS

rappresenta la query che mira a rimuovere righe della tabella estesa al verificarsi di una condizione di restrizione

```

<!ELEMENT PPREMOVE_ROWS ((%kdd_query_PPtable;),RESTR)>
<!ATTLIST PPREMOVE_ROWS xml_dest CDATA #IMPLIED>
<!ELEMENT RESTR (RESTR_COND)>
  <!ELEMENT RESTR_COND
    (OR_RESTR|NOT_RESTR|AND_RESTR|REL_RESTR|
    COMPARE_STRING_RESTR|HAS_EXCEPTION|HAS_MARK)>
<!ELEMENT OR_RESTR (RESTR_COND,RESTR_COND)>
<!ELEMENT NOT_RESTR (RESTR_COND)>
<!ELEMENT AND_RESTR (RESTR_COND,RESTR_COND)>
<!ELEMENT REL_RESTR (RESTR_TERM,RESTR_TERM)>
<!ATTLIST REL_RESTR type

```

```

        (less|less_or_equal|greater|
        greater_or_equal|equal|not_equal)
        #REQUIRED>
<!ELEMENT COMPARE_STRING_RESTR (RESTR_TERM,RESTR_TERM)>
<!ELEMENT HAS_EXCEPTION (EXCEPTION,ATTR)>
<!ELEMENT HAS_MARK (WITH_CODE,ATTR)>
<!ELEMENT RESTR_TERM (ATTR|CONSTANT)>

```

Esempio guida in linguaggio KDDML

Riprendiamo dagli esempi guida già visti per il DPL in 4.1.1 il caso relativo al trattamento dei duplicati e vediamo come possono essere espresse le stesse query con tipi validi in KDDML. Per brevità, partiamo dalla tabella in cui è già stata effettuata unificazione dei duplicati sulla base della chiave “Nominativo”+“Indirizzo”. Procedendo con i passi già illustrati in fig 4.2 si esegue lo “split” dei valori contenuti nell’attributo “Nominativo” in due nuovi attributi “Nome” e “Cognome” :

```

<KDDML_OBJECT>
  <KDD_QUERY name="Split">
    <PPSPLITTING_QUERY xml_dest="splitted.xml">
      <PPTABLE_LOADER file_name="merged.xml"/>
      <ATTR name="Nominativo"/>
      <NEW_ATTR_NAME>
        <ATTR name="Nome"/>
      </NEW_ATTR_NAME>
      <NEW_ATTR_NAME>
        <ATTR name="Cognome"/>
      </NEW_ATTR_NAME>
      <REGULAR_EXPRESSION Pattern=" "/>
    </PPSPLITTING_QUERY>
  </KDD_QUERY>
</KDDML_OBJECT>

```

A questo punto si aggiunge un nuovo attributo d'appoggio "Iniziale" di tipo stringa contenente gli stessi valori del "Nome":

```
<KDDML_OBJECT>
  <KDD_QUERY name="creaIniziale">
    <PPNEWATTRIBUTE_QUERY xml_dest="newIniziale.xml">
      <PPTABLE_LOADER file_name="splitted.xml"/>
      <NEWATTR attribute_name="Iniziale" attribute_type="string"/>
      <NEWATT_EXP>
        <TERMINE>
          <ATTR name="Nome"/>
        </TERMINE>
      </NEWATT_EXP>
    </PPNEWATTRIBUTE_QUERY>
  </KDD_QUERY>
</KDDML_OBJECT>
```

Si applica, quindi, riscrittura a questo nuovo attributo in modo da avere solo la prima lettera del valore attuale di "Iniziale":

```
<KDDML_OBJECT>
  <KDD_QUERY name="rewriteIniziale">
    <PPREWRITING xml_dest="rewriteIniziale.xml">
      <PPTABLE_LOADER file_name="newIniziale.xml"/>
      <ATTR name="Iniziale"/>
      <REWRITING_RULES>
        <RULE>
          <RULE_S>
            <REGULAR_EXPRESSION Pattern="([a-zA-Z])*"/>
            <STATEMENT>
              <SUBSTITUTION substitution_string="$1"/>
            </STATEMENT>
          </RULE_S>
        </RULE>
      </REWRITING_RULES>
    </PPREWRITING>
  </KDD_QUERY>
</KDDML_OBJECT>
```

A questo punto si aggiunge un nuovo attributo d'appoggio "Abbreviazione" di tipo stringa contenente gli stessi valori del "Nome" per contenere un valore che indica se l'iniziale in realtà è l'unica informazione sul nome (cioè il valore del nome nei dati iniziali compare come abbreviazione) oppure no:

```
<KDDML_OBJECT>
  <KDD_QUERY name="creaAbbreviazione">
    <PPNEWATTRIBUTE_QUERY xml_dest="newAbbreviazione.xml">
      <PPTABLE_LOADER file_name="rewriteIniziale.xml"/>
      <NEWATTR attribute_name="Abbreviazione" attribute_type="string"/>
      <NEWATT_EXP>
        <TERMINE>
          <ATTR name="Nome"/>
        </TERMINE>
      </NEWATT_EXP>
    </PPNEWATTRIBUTE_QUERY>
  </KDD_QUERY>
</KDDML_OBJECT>
```

a questo punto mediante riscrittura sostituiamo i valori del nuovo "Abbreviazione" con un valore distinto a seconda che nel nome compaia o meno il carattere ".":

```
<KDDML_OBJECT>
  <KDD_QUERY name="rewriteAbbreviazione">
    <PPREWRITING xml_dest="rewriteAbbreviazione.xml">
      <PPTABLE_LOADER file_name="newAbbreviazione.xml"/>
      <ATTR name="Abbreviazione"/>
      <REWRITING_RULES>
        <RULE>
          <RULE_S>
            <REGULAR_EXPRESSION Pattern="([a-zA-Z])(.)"/>
          <STATEMENT>
            <IF_THEN_ELSE>
              <RW_COND>
                <COMPARE_STRING_RW>
                  <RW_TERM>
                    <RAGGR group_variable="2"/>
                  </RW_TERM>
                </RW_COND>
              </IF_THEN_ELSE>
            </STATEMENT>
          </RULE_S>
        </RULE>
      </REWRITING_RULES>
    </PPREWRITING>
  </KDD_QUERY>
</KDDML_OBJECT>
```

```

        <RW_TERM>
            <CONSTANT valore="."/>
        </RW_TERM>
    </COMPARE_STRING_RW>
</RW_COND>
<STATEMENT>
    <SUBSTITUTION substitution_string="1"/>
</STATEMENT>
<STATEMENT>
    <SUBSTITUTION substitution_string="0"/>
</STATEMENT>
</IF_THEN_ELSE>
</STATEMENT>
</RULE_S>
</RULE>
</REWRITING_RULES>
</PPREWRITING>
</KDD_QUERY>
</KDDML_OBJECT>

```

Ottenuta la nuova tabella estesa ora si procede, seguendo lo stesso principio dell'esempio visto in fig. 4.2, con la marcatura delle istanze duplicate secondo la chiave "iniziale" + "cognome" + "indirizzo" e la successiva rimozione di quelle appena marcate ma che hanno meno informazione sul nome perché presente solo come abbreviazione. Quindi si ha l'individuazione e marcatura dei duplicati:

```

<KDDML_OBJECT>
    <KDD_QUERY name="markDuplicates">
        <PPMARK_DUPLICATES xml_dest="markDuplicates.xml">
            <PPTABLE_LOADER file_name="rewriteAbbreviazione.xml"/>
            <ATTRIBUTE_SEQ>
                <ATTR name="Iniziale"/>
                <ATTR name="Cognome"/>
                <ATTR name="Indirizzo"/>
            </ATTRIBUTE_SEQ>
            <WITH_CODE marca="mioCodice"/>
        </PPMARK_DUPLICATES>
    </KDD_QUERY>

```

```
</KDDML_OBJECT>
```

e successivamente la rimozione:

```
<KDDML_OBJECT>
  <KDD_QUERY name="a">
    <PPREMOVE_ROWS xml_dest="a">
      <PPTABLE_LOADER file_name="a.xml"/>
      <RESTR>
        <RESTR_COND>
          <AND_RESTR>
            <RESTR_COND>
              <HAS_EXCEPTION>
                <EXCEPTION marca="mioCodice"/>
                <ATTR name="Iniziale"/>
              </HAS_EXCEPTION>
            </RESTR_COND>
            <RESTR_COND>
              <COMPARE_STRING_RESTR>
                <RESTR_TERM>
                  <ATTR name="Abbreviazione"/>
                </RESTR_TERM>
                <RESTR_TERM>
                  <CONSTANT valore="1"/>
                </RESTR_TERM>
              </COMPARE_STRING_RESTR>
            </RESTR_COND>
          </AND_RESTR>
        </RESTR_COND>
      </RESTR>
    </PPREMOVE_ROWS>
  </KDD_QUERY>
</KDDML_OBJECT>
```

A questo punto l'obiettivo della rimozione dei duplicati è stato raggiunto, se si desidera la tabella estesa può ulteriormente essere "ripulita" dagli attributi in eccesso usati solo come appoggio così come in 4.2.

5.2 Estensione del supporto a run-time

Una query che risulta valida per il linguaggio KDDML, viene “srotolata” nell’annidamento e risolta. L’esecuzione di ogni sottoquery consiste nella chiamata alla classe che implementa l’operatore individuato dalla TAG del documento. Come è chiaro dalla definizione del linguaggio in 5.1.1, gli operatori di PreProcessing lavorano principalmente sulle tabelle estese.

5.2.1 Necessità di nuove strutture dati nel supporto a run-time

La struttura dati e i metodi di supporto alla gestione delle tabelle estese sono state realizzate come wrapper alla classe *Instances*, tipicamente utilizzata come supporto alla gestione di un insieme di istanze. Una tabella estesa risulta a tutti gli effetti costituita da una coppia di *Instances*, necessaria per la manipolazione atomica di dati e metadati che complessivamente detengono la semantica di una tabella estesa. L’implementazione completa di una tabella estesa è presente classe *InternalPPTable.java* il cui diagramma UML è illustrato in fig.5.1 . La classe *Instances* appartiene al pacchetto *weka.core* della libreria *Weka*, software open source emesso sotto licenza GNU, che fornisce una collezione di algoritmi per il data mining e strumenti per il preprocessing, la classificazione, la regressione, il clustering e le regole associative. La classe *InternalPPTable*, che estende la classe *InternalTable* per la manipolazione dei soli dati, ha costruttori che prevedono come parametri: una coppia di *Instances* oppure una coppia di oggetti di tipo *Element* (del package *org.w3c.dom* importato) . Presi due *Element* per la creazione di una tabella estesa essi rappresentano lo schema dei dati e quello dei metadati usati, vengono quindi manipolati per definire le istanze come

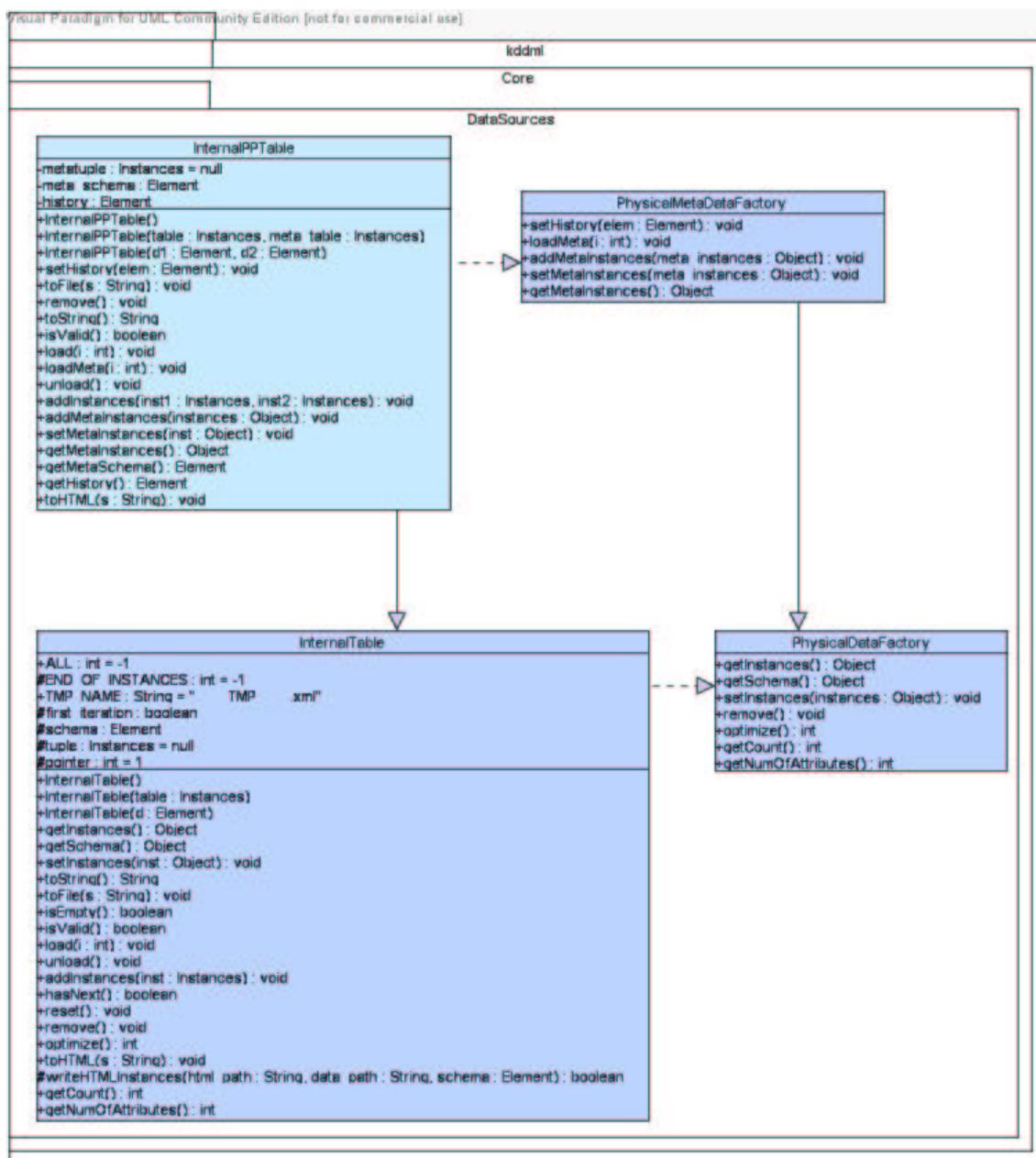


Figura 5.1: Schema UML delle tabelle estese

due Instances vuote ma con la struttura degli alberi DOM specificati. Ogni istanziamento di una tabella estesa porta all'inizializzazione di un *Element*

denominato *history* che rappresenta un albero in cui ciascun figlio ha due attributi: uno “*operation*” per l’operazione di PreProcessing a cui i dati e i metadati sono stati sottoposti, e un altro “*description*” contenente una breve descrizione sulle modalità di applicazione dell’operatore. Tra i metodi della classe, i metodi *load* e *unload* sono quelli più specifici per le problematiche che sussistono nella risoluzione di operatori che fanno uso di tabelle estese. Il problema della grossa quantità di dati da importare in memoria è infatti sicuramente quello che, per la natura del processo di Knowledge Discovery in generale, deve essere sempre considerato come componente influente nelle performance degli operatori di supporto al processo. L’idea è stata quindi quella di prevedere la possibilità di caricare in memoria un gruppo di istanze alla volta su cui applicare gli operatori specificati nelle query, e di poter liberare memoria dalle istanze caricate una volta utilizzate per ottenere i risultati necessari. Il metodo *load* :

```
public void load(int i)
    throws InternalTableException, DOMParserException {
    int num_inst = getCount();
    if (pointer > num_inst)
        // out of range
        throw new InternalTableException
            ("Exception in loadInstances: no more instance to import!");
    if (i == ALL) {
        // vengono importate tutte le istanze del file...
        String s = schema.getAttribute("file_name");
        KDDMLPPTableReader k =
            new KDDMLPPTableReader(kddml.Core.WorkingPaths.PPtable_path + s);
        k.importData();
        k.importMetaData();
        tuple = k.getInstances();
        metatuple = k.getMetaInstances();
        pointer = END_OF_INSTANCES;
    }
    else {
        // importazione di un blocco di istanze...
```

```

String s = schema.getAttribute("file_name");
KDDMLPPTableReader k =
    new KDDMLPPTableReader(
        kddml.Core.WorkingPaths.PPtable_path + s,pointer,
        pointer+i);
k.importData();
k.importMetaData();
tuple = k.getInstances();
metatuple = k.getMetaInstances();
pointer = pointer + i;
if (pointer > num_inst)
    pointer = END_OF_INSTANCES;
}
}

```

prevede l'importazione dei dati e dei metadati dei file di testo "file_name". data e "file_name".metadata, secondo l'attributo dello schema, nel path dedicato alle tabelle estese. L'importazione è fatta per un numero di istanze che varia in base al valore dell'intero passato come parametro al metodo. Sia il metodo *importData* che *importaMetaData* si preoccupano del parsing del file contenente la query con le DTD del linguaggio KDDML e la creazione di una tabella o di una tabella estesa rispettivamente. Solo il metodo *importMetaData*, invece, ha come effetto laterale il settaggio dell' *history*, per la tabella estesa appena creata, con il sottoalbero rappresentato dall'ultimo elemento del risultato del parsing. Il metodo *unload*:

```

public void unload() throws InternalTableException {
    // scrive le istanze in un file temporaneo
    String path_xml =
        kddml.Core.WorkingPaths.PPtable_path + TMP_NAME;
    String path_data =
        kddml.Core.WorkingPaths.PPtable_path + TMP_NAME + ".data";
    String path_metadata =
        kddml.Core.WorkingPaths.
            PPtable_path + TMP_NAME + ".metadata";
    File f_xml = new File(path_xml);
    File f_data = new File(path_data);
}

```

```
File f_metadata = new File(path_metadata);
if (first_iteration) {
    PrintWriter out_xml, out_data, out_metadata;
    try {
        f_xml.createNewFile();
        f_data.createNewFile();
        f_metadata.createNewFile();
        out_xml = new PrintWriter(new FileOutputStream(path_xml));
        out_data = new PrintWriter(new FileOutputStream(path_data));
        out_metadata = new PrintWriter(new FileOutputStream(path_metadata));
        out_xml.flush();
        out_data.flush();
        out_metadata.flush();
        out_xml.close();
        out_data.close();
        out_metadata.close();
    }
    catch(java.io.IOException e) {
        throw new InternalTableException("I/O error: "+e.toString());
    }
    first_iteration = false;
}
KDDMLPPTableWriter k = new KDDMLPPTableWriter
    (kddml.Core.WorkingPaths.PPtable_path + TMP_NAME, tuple, metatuple);
kToFile();
tuple.delete();
metatuple.delete();
// chiamata alla Garbage Collector
System.gc();
}
```

scrive all'interno di un file temporaneo l'insieme di istanze relative ai dati e ai metadati, liberando successivamente memoria.

5.2.2 Introduzione di nuovi operatori e modifica dell'interprete nel supporto a run-time

Il parsing della query XML porta alla costruzione di un albero DOM (*Document*) che è usato per creare e istanziare la classe che realizza l'interprete (*QueryExecutor.java*). Ogni figlio con tag "KDD_QUERY" presente nel *Document* rappresenta un oggetto di tipo *Element* (del package *org.w3c.dom* importato). Ciascuna query viene risolta in base al tipo di operatore, individuato dal tag della query, ricercando fra le classi del package *kddml.Operators* quella col nome corrispondente. Gli operatori sono suddivisi in pacchetti in base alla fase del processo di KDD che essi supportano, pertanto a:

- *kddml.Operators.DataMining*
- *kddml.Operators.PostProcessing*

è stato aggiunto il package *kddml.Operators.PreProcessing* per gli operatori che supportano i nuovi tipi introdotti nel linguaggio KDDML sulla base del DPL.

Interfaccia per tutti gli operatori di PreProcessing

L'interfaccia implementata da tutti gli operatori consiste di tre metodi:

readInput: in cui si implementa il caricamento dei dati di input. Questo metodo ha tre parametri costruiti e passati dalla classe *QueryExecutor.java* durante l' *unfolding* della query da eseguire, essi sono:

- **objects:** vettore di *KDDMLObject*, contenente solo i figli della query che nel linguaggio KDDML rappresentano un *KDDMLObject*;
- **elements:** vettore di *Element*, contenente i figli della query che nel linguaggio KDDML non rappresentano un *KDDMLObject*;

- **attributes:** tabella Hash contenente le associazioni nome-valore per ogni figlio della query.

execute: risoluzione dell'operatore e creazione del KDDMLObject da restituire in output

addHistory: aggiornamento dell'history per gli output (implementata solo quando l'output è una InternalPPTable)

Architettura del modulo di PreProcessing

La gerarchia delle classi del package kddml.Operators.PreProcessing illustrata in fig.5.2, prevede 3 superclassi che implementano l'interfaccia:

Instance_Level_Dipendent_Transformation superclasse per tutte le classi che implementano operatori che producono trasformazioni sulle istanze del dataset, con risultati che sono dipendenti dai valori assunti dalle altre istanze;

Instance_Level_Indipendent_Transformation superclasse per tutte le classi che implementano operatori che producono trasformazioni sulle istanze del dataset, con risultati che sono indipendenti dai valori assunti dalle altre istanze;

Schema_Level_Transformation superclasse per tutte le classi che implementano operatori che producono trasformazioni di schema;

Attenendosi alle descrizioni sulle tecniche di preprocessing illustrate in 2.1, tutte le operazioni implementate hanno lo scopo comune di garantire la qualità dei dati basandosi sulla conoscenza del dominio; quello che le differenzia è il campo d'azione (a livello di istanza e a livello di schema)

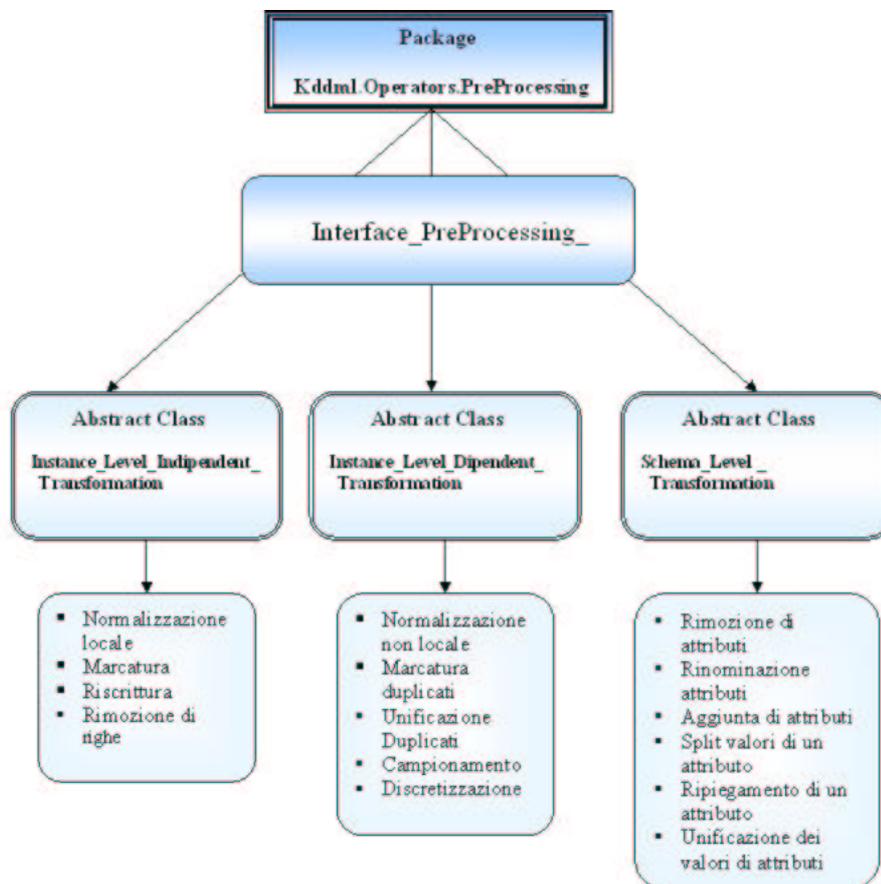


Figura 5.2: Gerarchia delle classi del modulo di PreProcessing

e la necessità o meno di considerare tutte le istanze per produrre i risultati (a livello di istanza dipendenti o indipendenti).

5.2.3 Operatori a livello di schema

Gli operatori di preprocessing che operano a livello di schema (vedi schema UML di fig. 5.3), sono implementati in classi che prendono un `KDDMLObject`

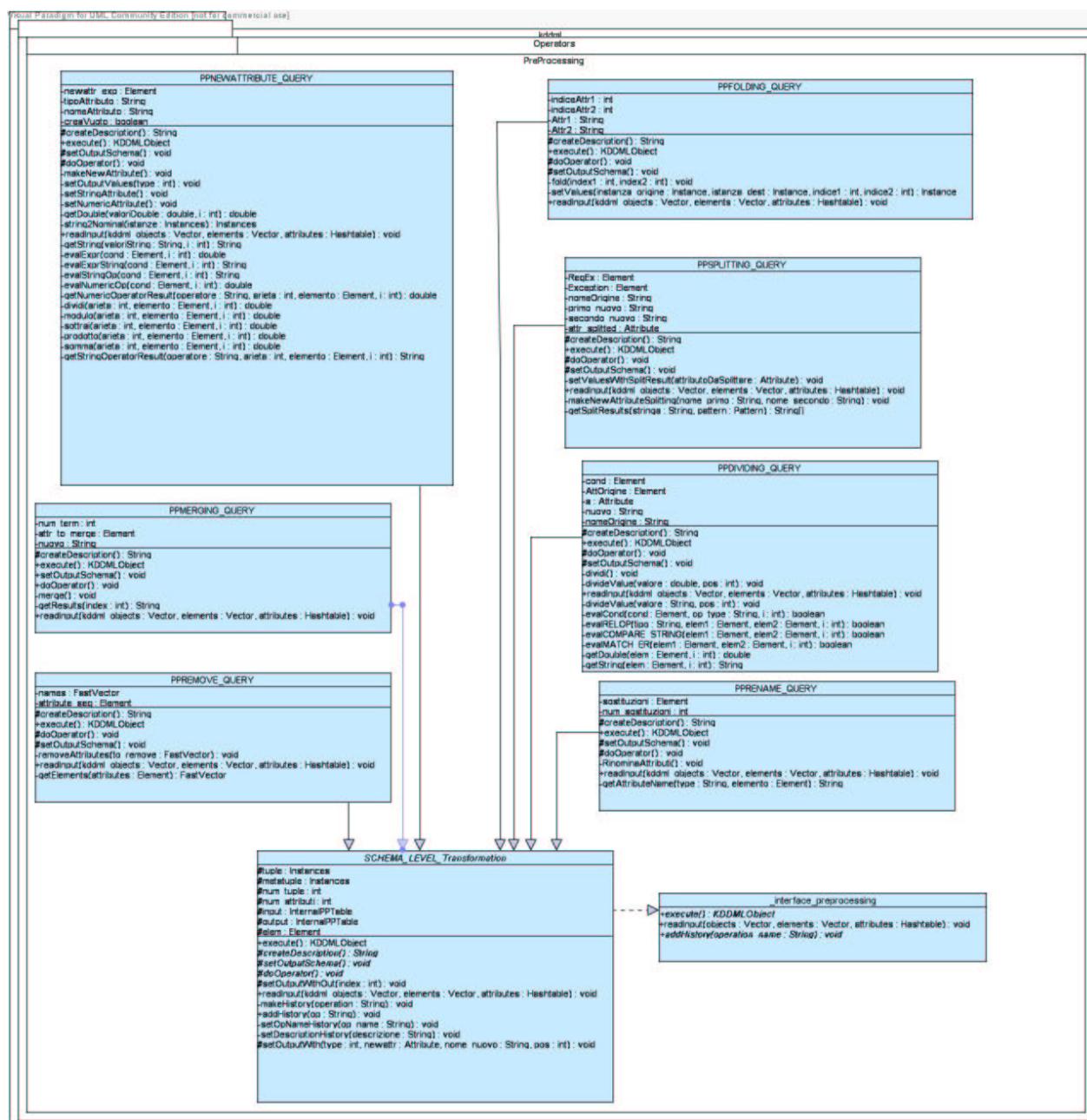


Figura 5.3: Gerarchia delle classi degli operatori a livello di schema

di tipo `InternalPPTable` (tabella estesa) e ne modificano lo schema relazionale. La superclasse che implementa l'interfaccia, prevede:

- il caricamento dello schema della tabella di input;
- la modifica dello schema secondo le specifiche proprie dell'operatore;
- il caricamento delle istanze di input per blocchi;
- l'esecuzione del corpo dell'operatore;
- la scrittura delle istanze nella tabella da restituire in output per il blocco di input appena caricato;
- al termine dell'analisi di tutte le istanze in input, la costruzione della componente *History* dell'InternalPPTable di output, come copia della History dell'input più un nuovo figlio relativo all'operazione appena applicata.

Classi PPREMOVE_QUERY e PPRENAME_QUERY

La rimozione e la rinominazione sono le operazioni base a livello di schema. Entrambe operano esclusivamente sugli attributi senza scendere mai a livello di istanza. La classe PPREMOVE_QUERY, infatti, implementa l'operazione di rimozione di una sequenza di attributi da una tabella estesa e usa i blocchi delle istanze di input, caricate di volta in volta, per selezionare i valori dei soli attributi da restituire nell'output. In modo ancora più stretto la classe PPRENAME_QUERY implementa la rinominazione di una sequenza di attributi realizzandola già al momento della modifica dello schema della tabella di output; nel resto dell'esecuzione carica le istanze di input a blocchi e le scrive nell'output così come sono.

Classe PPSPLITTING_QUERY

La classe PPSPLITTING_QUERY implementa l'operatore di SPLIT. Anche questo è un costrutto che modifica lo schema relazionale della tabella, ed appartiene quindi alla classe delle trasformazioni di schema. Vengono introdotti due nuovi attributi, e la sequenza di caratteri, che rappresenta il valore dell'attributo specificato in ingresso, è spezzata e distribuita sui due nuovi attributi. Il punto di "spezzamento" viene identificato come la prima occorrenza di una sottostringa specificata dall'utente attraverso una espressione regolare, e tale sottostringa apparterrà interamente alla seconda parte del taglio. Per il trattamento delle espressioni regolari e il matching con i valori dell'attributo è stato utilizzato il package *java.util.regex*. Se il punto di spezzamento non dovesse essere individuato in qualche tupla, è possibile richiedere la sua marcatura, eventualmente assegnando un codice di identificazione. L'operazione di SPLIT appartiene a quella classe di operazioni utili per risolvere conflitti strutturali dei dati in caso di sorgente multipla visti in 2.1.2.

Classe PPFOLDING_QUERY

La classe PPFOLDING_Query implementa l'operatore FOLD che, a livello di schema relazionale, comporta la riduzione del numero di colonne di una tabella, e contemporaneamente un raddoppio del numero di righe. In particolare, viene rimossa la colonna corrispondente al primo attributo specificato, ma sarebbe improprio dire che viene rimosso l'attributo: infatti, il valore del primo attributo viene inserito in corrispondenza del secondo. Ogni tupla viene in questo modo replicata, ed ogni coppia si distinguerà solamente per il valore associato al secondo dei due attributi specificati nel costrutto FOLD. Il risultato dell'operazione sulla parte dei dati della tabella estesa di

dimensione $n*m$ è una tabella di dimensione $2n*m-1$. Lo stesso vale per i metadati che, per la colonna relativa all'attributo origine, verranno spostati in corrispondenza dei valori dell'attributo destinazione sulla colonna dei metadati .

Classe PPNEWATTRIBUTE_QUERY

Classe che implementa una trasformazione a livello di schema, mediante la costruzione di un nuovo attributo dal nome e dal tipo specificato. I valori assunti dal nuovo attributo per la parte relativa ai dati sono, a discrezione dell'utente:

- missing,
- calcolati valutando un'espressione,
- copia di altri attributi già presenti.

Il corrispondente attributo dei metadati ha valori:

- missing (se il nuovo attributo dei dati ha valori missing oppure calcolati su un'espressione);
- gli stessi valori dei metadati dell'attributo originari (se si costruisce il nuovo attributo come copia).

Le operazioni a disposizione per la costruzione di un attributo mediante valutazione di un'espressione, sono numeriche o su stringhe. Le operazioni numeriche trattate sono :

- sommatoria,
- produttoria,

- sottrazione,
- divisione,
- modulo.

Quelle su stringhe sono :

- concatenazione,
- confronto.

Su ciascuna istanza il nuovo attributo valuta nell'espressione l'operatore usando come operandi:

- il valore assunto, su quell'istanza, da un attributo specificato,
- un valore costante.

Vista la possibilità di specificare il tipo dell'attributo da creare, si fornisce anche la funzionalità auspicata di conversione del tipo. L'utilità è evidente soprattutto quando si incorre in problemi dovuti all'incompatibilità tra operandi durante operazioni che coinvolgono attributi di tipo diverso.

Classe PPMERGING_QUERY

Classe che risolve l'operatore di merging dei valori di una sequenza di attributi. L'operazione di merging è un'operazione che opera a livello di schema sui dati prevedendo l'inserimento di un nuovo attributo di tipo string sia sui dati che sui metadati. Il valore, relativo alla parte dei dati, che il nuovo attributo assume è formato dalla fusione, intesa come concatenazione di stringhe, di attributi, i quali non vengono rimossi dalla tabella. Sarà eventualmente necessario rimuoverli successivamente in maniera esplicita. Si aggiunge in

modo simmetrico anche l'attributo dei metadati, corrispondenti a tale nuovo attributo, tutti con valore missing. Si è preferita questa alternativa, per mantenere le informazioni di marcatura dei due attributi di fusione. Infatti la trasformazione della tabella attraverso il MERGE non assegna alcuna informazione di marcatura al nuovo attributo risultato.

Esempi di query con operatori a livello di schema

Vediamo di seguito un esempio di utilizzo di operatori a livello di schema in linguaggio KDDML:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE KDDML_OBJECT SYSTEM
"../../../../project/DTD/Queries/kddml_query.dtd"
>
<KDDML_OBJECT>
  <KDD_QUERY name="SchemaLevel">
    <PPNEWATTRIBUTE_QUERY xml_dest="newattribute.xml">
      <PPSPLITTING_QUERY xml_dest="splitting.xml">
        <PPTABLE_LOADER file_name="dddd.xml"/>
        <ATTR name="address"/>
        <NEW_ATTR_NAME>
          <ATTR name="street"/>
        </NEW_ATTR_NAME>
        <NEW_ATTR_NAME>
          <ATTR name="number"/>
        </NEW_ATTR_NAME>
        <REGULAR_EXPRESSION Pattern="n\."/>
        <EXCEPTION marca="no ER"/>
      </PPSPLITTING_QUERY>
      <NEWATTR attribute_name="login_name" attribute_type="string"/>
      <NEWATT_EXP>
        <TERMINE>
          <ATTR name="surname"/>
        </TERMINE>
      </NEWATT_EXP>
    </PPNEWATTRIBUTE_QUERY>
  </KDD_QUERY>
```

```
</KDDML_OBJECT>
```

La query KDDML precedente opera uno split dei valori assunti dall'attributo "address" sulla base dell'espressione regolare "n\.". Al termine dello split, pertanto, i valori verranno suddivisi su due attributi ("street" e "number") usando come punto di spezzamento la sottostringa che mappa l'espressione regolare. I valori che non contengono l'espressione sono marcati con l'etichetta "no ER" nei metadati corrispondenti. Alla tabella estesa, quindi, viene applicato un operatore di NEWATTRIBUTE con il quale viene aggiunto in fondo a quelli già esistenti, un attributo con etichetta "login_name" di tipo stringa. Il criterio di costruzione dei valori di tale nuovo attributo è quello di "copia di un attributo già presente" (in questo caso come copia di "surname"). Particolare interessante è il tipo del "login_name" che può essere scelto indipendentemente dall'attributo che si vuol usare per costruire i valori, nel caso specifico, nonostante l'attributo di riferimento avesse tipo enumerato, il nuovo attributo ha gli stessi valori ma la tipologia assegnata è stringa. Se, ad esempio, a partire dalla tabella estesa risultante si volesse ora costruire la "login" come composizione del "login_name" e dell'attributo numerico "età", quindi eliminare il vecchio "login_name", in linguaggio KDDML si può:

```
<?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE KDDML_OBJECT
SYSTEM "../../project/DTD/Queries/kddml_query.dtd"
>
<KDDML_OBJECT>
  <KDD_QUERY name="remove">
    <PPREMOVE_QUERY xml_dest="removed.xml">
      <PPMERGING_QUERY xml_dest="merged.xml">
        <PPTABLE_LOADER file_name="newattribute.xml"/>
        <ATTRIBUTE_SEQ>
          <ATTR name="login_name"/>
```

```
<ATTR name="age"/>
</ATTRIBUTE_SEQ>
<NEW_ATTR_NAME>
  <ATTR name="login"/>
</NEW_ATTR_NAME>
</PPMERGING_QUERY>
<ATTRIBUTE_SEQ>
  <ATTR name="login_name"/>
</ATTRIBUTE_SEQ>
</PPREMOVE_QUERY>
</KDD_QUERY>
</KDDML_OBJECT>
```

In linguaggio KDDML questo modella le richieste dell'utente che sono eseguite unificando i valori di "login_name" e "age", considerando entrambi come stringhe indipendentemente dal tipo dell'attributo che essi rappresentano, per consentire la composizione di stringhe. Costruito il nuovo attributo come unificazione, quindi, viene applicato un operatore di rimozione di attributi sul "login_name".

5.2.4 Operatori a livello di istanza indipendenti dai valori assunti da tutte le istanze

Gli operatori di preprocessing che operano a livello di istanza e indipendenti (vedi schema UML di fig. 5.4), sono implementati in classi che prendono un KDDMLObject di tipo InternalPPTable (tabella estesa) e modificano i valori assunti da alcune istanze senza valutare i valori assunti dall'intero insieme di istanze della tabella estesa. La superclasse che implementa l'interfaccia, prevede:

- il caricamento dello schema della tabella di input;
- il caricamento delle istanze di input per blocchi;

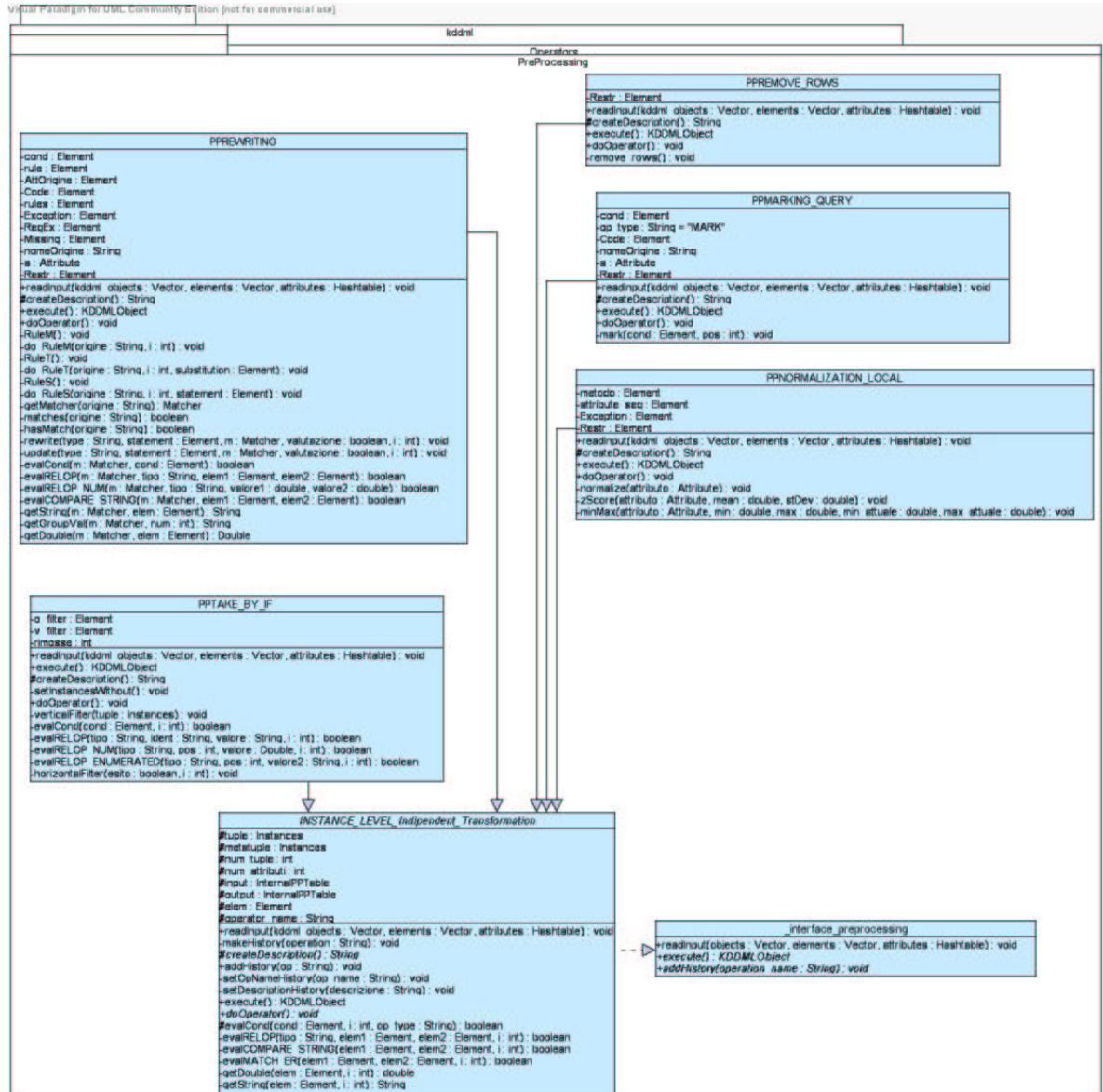


Figura 5.4: Gerarchia delle classi degli operatori a livello di istanza, con risultati indipendenti dai valori assunti da tutte le istanze

- l'esecuzione del corpo dell'operatore per la modifica dei valori;
- la scrittura delle istanze nella tabella da restituire in output per il

blocco di input appena caricato;

- al termine dell'analisi di tutte le istanze in input, la costruzione della componente *History* dell'InternalPPTable di output, come copia della History dell'input più un nuovo figlio relativo all'operazione appena applicata.

Per tutti gli operatori che estendono questa superclasse esiste un elemento opzionale "*Restr*", semanticamente equivalente alla clausola *Where* delle query SQL, con il quale specificare le condizioni da verificare per reputare un'istanza come candidata per la modifica dei suoi valori secondo le specifiche dell'operatore. Questo tipo di condizioni sono di tipo booleano e valutate sui dati o sui metadati della tabella estesa, in particolare esse possono essere combinazioni di operatori booleani AND,OR,NOT applicati a condizioni di :

- relazioni numeriche ($<$, $>$, $=$, \geq , \leq) sui valori dei dati,
- equivalenza di stringhe sui valori dei dati,
- equivalenza di stringhe su marche o eccezioni su valori dei metadati associati ad attributi.

Classe PPNORMALIZATION_LOCAL

Rispetto alla sintassi proposta nel DPL [9], l'operatore di normalizzazione è stato implementato in due classi distinte a seconda che la normalizzazione faccia uso di valori relativi alle istanze nel complesso (minimo, massimo, media e deviazione standard):

- valutati dall'utente e passati come parametro,
- calcolati in modo esatto sulla base di tutte le istanze.

La normalizzazione di tipo locale modella il primo dei due casi, per questa operazione pertanto è possibile caricare le istanze a blocchi per l'esecuzione dell'operatore nelle modalità previste (già illustrate in 2.1.3):

- min-max,
- z-score.

Classe PPREMOVE_ROWS

La classe PPREMOVE_ROWS implementa l'operatore di rimozione di righe dalla tabella estesa. In questo operatore, l'elemento "*Restr*" è obbligatorio per l'individuazione delle righe da eliminare come quelle istanze che soddisfano delle condizioni sui dati o sui metadati. Per la preparazione dei dati, tale operatore può essere utile in particolare per l'eliminazione di istanze con valori anomali o individuati come outlier.

Classe PPMARKING_QUERY

La classe PPMARKING_QUERY implementa l'operatore di marcatura, consente cioè l'inserimento di un valore nei metadati in corrispondenza di dati che rispettano una condizione specificata. La condizione di marcatura di una tupla può essere multiattributo, e quindi non dipendere solamente dal valore di una singola colonna. Inoltre l'operatore MARK permette di valutare un'espressione piuttosto che effettuare un confronto tra stringhe: la condizione di marcatura è perciò espressa attraverso una operazione di confronto tra due espressioni, oppure attraverso una combinazione, in AND, OR oppure NOT, di tali condizioni primitive. Le operazioni di confronto possono essere $<$, $>$, $=$, \geq , \leq , per valori numerici, e equivalenza di stringhe.

Classe PPREWRITING

Classe che risolve l'operatore di riscrittura. Tramite questo operatore è possibile ricercare, marcare e sostituire dei valori di un attributo. Le regole di riscrittura previste sono:

RULE_S: La regola di riscrittura principale, `rule_S`, permette di sostituire tutte le stringhe che rappresentano il valore di un attributo di una tabella estesa. L'applicazione della regola di riscrittura consiste nell'effettuare un pattern matching tra un'espressione regolare e i valori dell'attributo presi come stringhe. Se esso dovesse fallire per qualche istanza, la sostituzione non avviene, e nel caso in cui fosse specificato il tag `EXCEPTION`, l'istanza viene marcata in corrispondenza dell'attributo, ossia estesa con le informazioni sull'avvenuta eccezione. La stringa di sostituzione può essere, nel caso più semplice, una stringa costante, ma la vera potenza di questo costrutto consiste nella possibilità di definire una nuova stringa a partire da quella data in ingresso. Infatti il meccanismo di pattern matching non solo verifica che la stringa in ingresso soddisfi un certo schema di costruzione, ma allo stesso tempo lega, quando richiesto, delle sottostringhe a cosiddette variabili di raggruppamento. Allora la nuova stringa può essere composta a piacere, a partire da componenti della stringa in ingresso. È inoltre possibile esprimere una condizione, attraverso l'espressione "*if then else*", in modo che la sostituzione venga effettuata solamente in alcune istanze.

RULE_T: Ha un comportamento simile a quello di `rule_S`, tuttavia in questo caso il pattern non si riferisce all'intero contenuto dell'attributo. Il pattern matching viene effettuato ripetutamente e la stringa di sostituzione andrà a rimpiazzare tutte le occorrenze del pattern incontrate

nel valore attuale dell'attributo preso come stringa. Se è specificato il tag EXCEPTION e nessuna occorrenza del pattern dovesse essere trovata all'interno di un valore, allora nell'estensione della tabella l'istanza verrà "estesa" con le informazioni di marcatura/eccezione. Anche nella clausola rule_T la stringa di sostituzione è costituita da caratteri e da variabili di raggruppamento \$n racchiuse da parentesi tonde, come ad esempio "abc(\$1)%&\$£(\$2)123(\$3)!". Se si volesse che la sequenza di caratteri (\$1) non fosse trattata come meta-stringa rappresentante il valore del primo raggruppamento, ma come una normale parola di sostituzione, sarà sufficiente farla precedere il carattere "\".

RULE_M Essa non effettua alcuna modifica sui valori dell'attributo in questione, ma serve solamente per verificare che sia rispettato un dato schema di costruzione delle stringhe, chiamato pattern. Ogni qual volta il pattern matching dovesse fallire, l'istanza viene "marcata" e l'eccezione sollevata viene memorizzata nell'estensione della tabella.

La riscrittura è un'operazione utile in diversi momenti della pulizia dei dati, in particolare la regola Rule_M è utile per individuare e marcare i valori che si presentano :

- come possibili errori di digitazione e di sintassi ;
- con formato anomalo perché le fonti dei dati sono eterogenee (ad esempio nome attributo: "Sesso" valori: "M/F" VS "Maschile/Femminile", oppure nome attributo: "data di nascita" formato valori: gg/mm/aaaa VS mm/gg/aaaa VS gg.mm.aaaa)

Mediante le regole Rule_S e Rule_T, è possibile sostituire i valori che rispettano un'espressione regolare, o già marcati dopo la Rule_M, con dei nuovi

valori costruiti eventualmente usando sottostringhe dei valori attuali. Oltre a di problemi dei dati individuabili tramite pattern matching, in linguaggio KDDML la riscrittura è applicabile anche a dati mancanti consentendone altresì la loro marcatura e la riscrittura. Grazie a questo è possibile, quindi applicare tecniche note di trattamento dei dati mancanti già viste in 2.1.7.

Esempi di query con operatori a livello di istanza indipendenti

Vediamo di seguito un esempio di utilizzo di operatori a livello di istanza indipendenti in linguaggio KDDML:

```
<?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE KDDML_OBJECT
SYSTEM "../../../project/DTD/Queries/kddml_query.dtd"
>
<KDDML_OBJECT>
  <KDD_QUERY name="remove">
    <PPREMOVE_ROWS xml_dest="removerows.xml">
      <PPNORMALIZATION_LOCAL xml_dest="norm.xml">
        <PPTABLE_LOADER file_name="dddd.xml"/>
        <LOCAL_NORM_METHOD>
          <MIN_MAX_LOCAL max="10" max_Attuale="75" min="0" min_Attuale="40"/>
        </LOCAL_NORM_METHOD>
        <ATTRIBUTE_SEQ>
          <ATTR name="temperature"/>
        </ATTRIBUTE_SEQ>
        <EXCEPTION marca="out_of_range"/>
      </PPNORMALIZATION_LOCAL>
    <RESTR>
      <RESTR_COND>
        <HAS_EXCEPTION>
          <EXCEPTION marca="out_of_range"/>
          <ATTR name="temperature"/>
        </HAS_EXCEPTION>
      </RESTR_COND>
    </RESTR>
  </PPREMOVE_ROWS>
</KDD_QUERY>
```

```
</KDDML_OBJECT>
```

In linguaggio KDDML questo esempio modella le richieste dell'utente che sono eseguite normalizzando i valori dell'attributo "temperature" con una tecnica di min-max locale basata su un range di minimo e massimo stimato pari a [40,75]. A seguito della normalizzazione tutti i valori che cadono fuori dal range specificato sono marcati nei metadati corrispondenti con la marca "Ex. out_of_range". In un secondo momento, quindi, tutte le istanze che presentano nei metadati associati all'attributo "temperature" l'eccezione "Ex. out_of_range", vengono eliminate dalla tabella estesa finale. Passando a un esempio sulla rappresentazione in KDDML di query per la riscrittura, supponiamo di voler applicare sullo stesso attributo ("new_attribute"):

- regole di marcatura in corrispondenza di valori mancanti e in corrispondenza di valori che rispettano un pattern,
- regole di tipo rule_S per modificare valori dell'attributo che rispettano un'espressione regolare.

Abbiamo, quindi, ad esempio:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE KDDML_OBJECT SYSTEM
".../project/DTD/Queries/kddml_query.dtd"
>
<KDDML_OBJECT>
  <KDD_QUERY name="instanceIndipendent.xml">
    <PPREWRITING xml_dest="rewrited.xml">
      <PPTABLE_LOADER file_name="newattribute.xml"/>
      <ATTR name="new_attribute"/>
      <REWRITING_RULES>
        <RULE>
          <RULE_M>
            <IS_MISSING_RW/>
```

116 5. Estensione del linguaggio KDDML e del supporto a run-time

```
        <EXCEPTION marca="is_missing"/>
    </RULE_M>
</RULE>
<RULE>
    <RULE_M>
        <REGULAR_EXPRESSION Pattern="(.*)(n+)y"/>
        <EXCEPTION marca="ER(.*)(n+)y"/>
    </RULE_M>
</RULE>
<RULE>
    <RULE_S>
        <REGULAR_EXPRESSION Pattern="over(.*)" />
        <STATEMENT>
            <SUBSTITUTION substitution_string="rewrited"/>
        </STATEMENT>
        <EXCEPTION marca="no_ER"/>
    </RULE_S>
</RULE>
<RULE>
    <RULE_S>
        <REGULAR_EXPRESSION Pattern="(.*)(n+)y"/>
        <STATEMENT>
            <IF_THEN_ELSE>
                <RW_COND>
                    <COMPARE_STRING_RW>
                        <RW_TERM>
                            <RAGGR group_variable="2"/>
                        </RW_TERM>
                        <RW_TERM>
                            <CONSTANT valore="n"/>
                        </RW_TERM>
                    </COMPARE_STRING_RW>
                </RW_COND>
                <STATEMENT>
                    <SUBSTITUTION substitution_string="\$1"/>
                </STATEMENT>
            </IF_THEN_ELSE>
        </STATEMENT>
        <EXCEPTION marca="no_ER2"/>
    </RULE_S>
</RULE>
```

```
</REWRITING_RULES>  
</PPREWRITING>  
</KDD_QUERY>  
</KDDML_OBJECT>
```

Osserviamo in particolare nell'esempio l'ultima regola *rule_S* applicata a quei valori dell'attributo "new_attribute" che rispettano l'espressione regolare "(.*)_(n+)y". Al verificarsi della corrispondenza con il pattern, vengono legate delle sottostringhe a cosiddette variabili di raggruppamento, grazie a cui è possibile verificare se la sottostringa che mappa il gruppo numero due dell'espressione regolare coincide con la stringa "n", prima di decidere di sostituire il valore dell'attributo con la sottostringa che mappa il gruppo numero 1.

5.2.5 Operatori a livello di istanza dipendenti dai valori assunti da tutte le istanze

Gli operatori di preprocessing che operano a livello di istanza e dipendenti (vedi schema UML di fig. 5.5), sono implementati in classi che prendono un *KDDMLObject* di tipo *InternalPPTable* (tabella estesa) e modificano i valori assunti da alcune istanze sulla base dei valori assunti dall'intero insieme di istanze della tabella estesa. La superclasse che implementa l'interfaccia, prevede:

- il caricamento dello schema della tabella di input;
- il caricamento di tutte le istanze di input;
- l'esecuzione del corpo dell'operatore per la modifica dei valori;
- la scrittura delle istanze nella tabella da restituire in output;

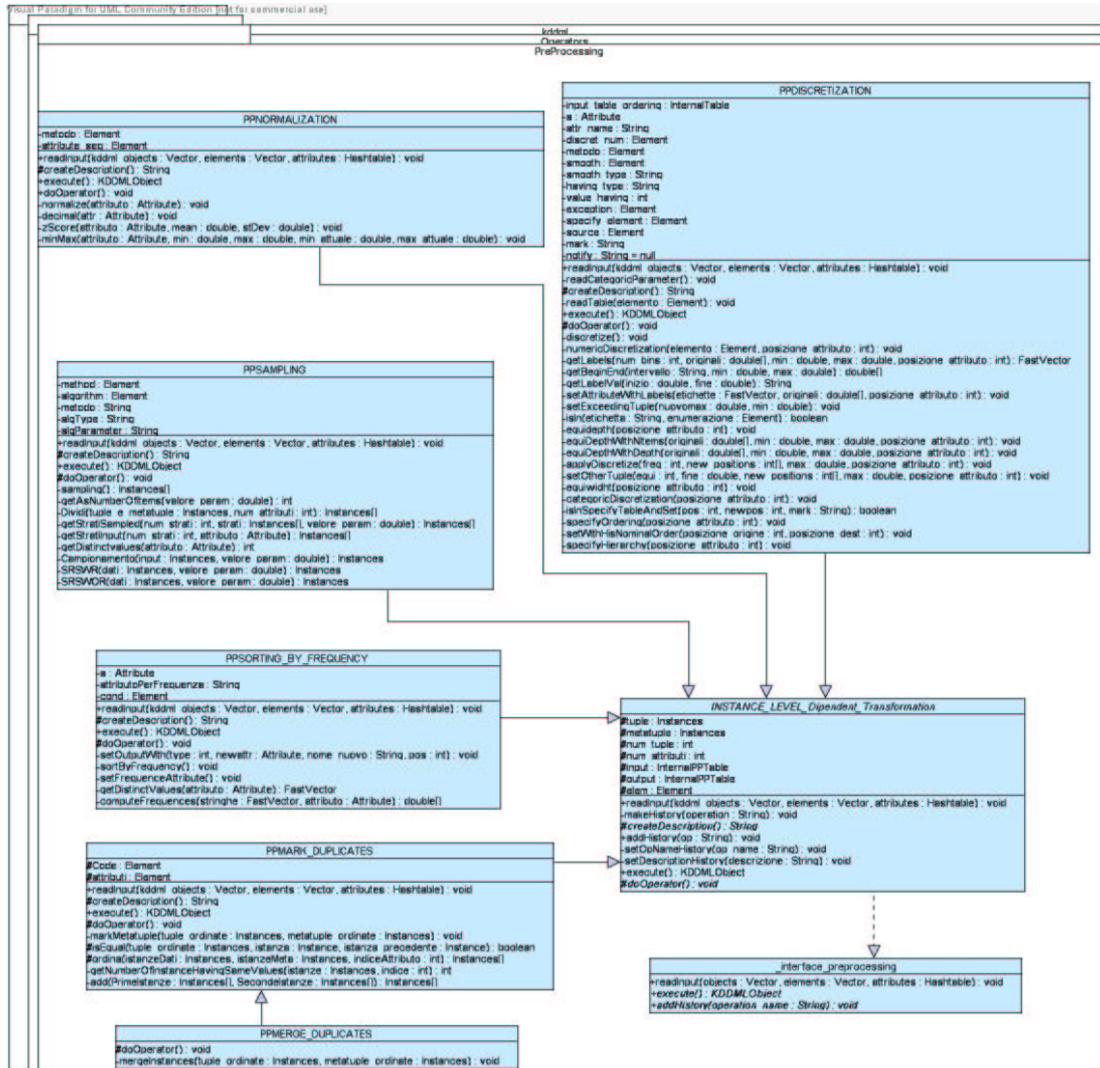


Figura 5.5: Gerarchia delle classi degli operatori a livello di istanza, con risultati dipendenti dai valori assunti da tutte le istanze

- al termine, la costruzione della componente *History* dell'*InternalPPTable* di output, come copia della *History* dell'input più un nuovo figlio relativo all'operazione appena applicata.

Classe PPNORMALIZATION

Questa classe implementa la normalizzazione non locale cioè per la quale non si prendono valori stimati per il minimo, il massimo, la media o la deviazione standard, questi infatti sono calcolati in modo esatto sulla base dei valori assunti dagli attributi da normalizzare, su tutte le istanze. Per la natura di questa normalizzazione, pertanto, è necessario portare tutte le istanze nelle struttura dati (Instances) di input per eseguire l'operatore nelle modalità previste (già illustrate in 2.1.3):

- min-max,
- z-score,
- decimale.

Classe PPDISCRETIZATION

Questa classe implementa la discretizzazione di attributi numerici e categorici illustrata in 2.1.4. La discretizzazione numerica consiste nel suddividere i valori assunti dall'attributo in un insieme di intervalli, quindi si sostituisce ciascun valore assunto con l'etichetta del suo intervallo di appartenenza. Le due varianti sono:

- *Natural Binning*,
- *Equal Frequency Binning*.

Nel *Natural Binning* la suddivisione mira a ottenere intervalli della stessa ampiezza. Nell' *Equal Frequency Binning* la suddivisione mira a ottenere intervalli nei quali ricade la stessa quantità dei valori originali. L'etichetta da associare a ciascun intervallo ottenuto può essere :

- la media dei valori compresi nell'intervallo,
- la mediana dei valori, calcolata sui quei valori originali che ricadono nell'intervallo,
- l'estremo superiore dell'intervallo,
- l'estremo inferiore dell'intervallo,
- scelta tra le etichette specificate dall'utente (ciascuna etichetta è associata all'intervallo seguendo la corrispondenza tra gli ordinamenti delle etichette fornite e degli intervalli ottenuti).

Per realizzare la discretizzazione *EquiWidht* (Natural Binning) è stata istanziata la classe della libreria Weka “weka.filters.unsupervised.attribute.Discretize” e applicato il filtro di discretizzazione alle istanze. Per il metodo *EquiDepth* (Equal Frequency Binning), invece, si è preferito risolvere l'operatore “a mano” per evitare i risultati indesiderati che si verificano con le classi messe a disposizione da Weka, nel caso di attributi con gli stessi valori duplicati per un numero superiore a quello della frequenza desiderata nell'intervallo. In questi casi infatti con la classe di Weka e l'uso del metodo “setUseEqual-Frequency(true)” accade che se c'è una quantità di valori identici superiore alla quantità della frequenza prefissata su ciascun intervallo, questi vengono comunque messi nello stesso. Il risultato desiderato secondo le specifiche del DPL è invece che gli intervalli contengano sempre una quantità di valori pari alla frequenza visto che l'etichetta può essere scelta anche mediante enumerazione che a quel punto renderà distinti anche due intervalli che contengono gli stessi valori. Nel caso di etichettamento mediante enumerazione, è possibile che le etichette fornite non coprano tutti gli intervalli risultanti dall'operazione di discretizzazione, in queste condizioni la scelta è stata quella di terminare comunque con successo l'operazione lasciando gli intervalli

per i quali non si ha etichetta così come sono durante il calcolo $[\text{inf}, \text{sup}]$. Su entrambe le modalità di discretizzazione numerica rimane comunque l'effetto naturale per cui un intervallo, nello specifico l'ultimo, ha un'ampiezza (o una frequenza) che può essere diversa da quella di tutti gli altri per ovvi motivi di scarto. La discretizzazione categorica consiste nel ridurre il numero di valori che l'attributo può assumere. Per i valori categorici non esiste un ordinamento pertanto è possibile:

- usare una tabella con cui ottenere un ordinamento sui possibili valori assunti, e successivamente applicare una discretizzazione numerica basata su ordinamento;
- usare una tabella in cui è specificata la gerarchia e sostituire i valori originali con il gruppo di appartenenza;
- usare l'ordine indotto dalla scala nominale, SOLO per gli attributi nominali, e successivamente applicare una discretizzazione numerica basata su ordinamento;.

In particolare quest'ultima modalità è stata aggiunta nel linguaggio KDDML pur non essendo prevista nel DPL, per dare la possibilità di trattare più semplicemente come numerici quegli attributi categorici ordinali per i quali esiste una sequenza significativa. E' possibile, infatti, decidere di sfruttare il fatto che al momento della definizione dell'attributo e dei valori nominali che esso poteva assumere, questi ultimi fossero già inseriti in sequenza crescente.

Classe PPSORTING_BY_FREQUENCY

Questo classe implementa l'operatore che permette di ordinare le tuple di una tabella in base alla frequenza con cui i valori di un attributo compaiono. L'utilità di questo riordinamento sta nell'ausilio che esso offre nella ricerca

degli outliers, i quali ricorrono con una frequenza bassa. Il problema degli outliers è in realtà una questione difficile da affrontare, e che non può prescindere da un'attenta analisi dei dati. Il costrutto in questione costituisce solamente un ulteriore strumento di analisi dei dati. Alla tabella riordinata viene anche aggiunto un nuovo attributo "Frequenza" + Attributo, dove Attributo è l'attributo su cui è stata ordinata la tabella. Esso riporta, per ogni tupla, la frequenza con cui è stato riscontrato, all'interno dell'intera tabella, il valore dell'attributo in questione. Assieme all'attributo viene aggiunto anche il relativo campo di estensione, il quale può essere utilizzato tipicamente per marcare le tuple che non raggiungono una certa soglia di frequenza minima. L'operatore realizzato porta alla costituzione di una tabella estesa le cui istanze sono ordinate in base alla frequenza in modo crescente, in fondo vengono lasciate tutte le istanze in cui l'attributo di riferimento ha valore mancante. La scelta di mettere i valori mancanti tutti in fondo è stata voluta per renderne più evidente la presenza, questo infatti porta alla scelta di tecniche per il loro trattamento che occupano una parte importante nella fase di preprocessing come già visto in 2.1.7.

Classe PPSAMPLING

Classe che risolve l'operatore di campionamento. Il campionamento è una trasformazione dipendente perché necessita di tutte le informazioni sui dati. Le modalità di campionamento previste sono quelle illustrate in 2.1.5:

- campionamento semplice;
- campionamento stratificato;
- campionamento su clusters.

Il campionamento semplice prevede di restituire un insieme di istanze (con o senza rimpiazzo), di cardinalità fissata, a partire da quelle iniziali. L'insieme campione con rimpiazzo è ottenuto usando la classe "Resample" del package fornito dalla libreria Weka "weka.filters.unsupervised.instance", per filtrare le istanze. Il campionamento stratificato costruisce un insieme composto da una quantità fissata di istanze, per ciascun valore assunto da un attributo nominale specificato. Il campionamento su clusters restituisce un insieme composto da una cardinalità fissata di sottoinsiemi, ciascuno contenente tutte le istanze che hanno lo stesso valore di un attributo fissato. Come caso particolare, se si fissa come attributo il cluster di appartenenza e si chiede di campionare in modo da avere items = "n", si ottiene un insieme contenente tutti gli elementi di "n" clusters. Per realizzare il campionamento su clusters che prevede di estrarre un insieme composto da una quantità fissata di elementi presi da ciascun cluster si può utilizzare la modalità "stratificata" specificando come attributo il "cluster di appartenenza".

Classi PPMARK_DUPLICATES e PPMERGE_DUPLICATES

La classe PPMARK_DUPLICATES risolve l'operatore di marcatura delle istanze che risultano duplicate. La verifica della duplicazione avviene sulla base di una "chiave" composta da una sequenza di attributi specificati così come illustrato in 2.1.6. La marcatura dei duplicati è una operazione che opera a livello di istanza. Essa prevede di aggiungere una marca con un codice specificato, o con un "NO MARK" (se nessun codice è specificato), al metadato relativo al valore di ciascun attributo chiave dell'istanza duplicata. La classe PPMERGE_DUPLICATES estende la PPMARK_DUPLICATES e risolve l'operatore di unificazione delle istanze individuate come duplicate. L'individuazione dei duplicati è la stessa della superclasse. L'unificazione

provoca una riduzione della dimensione della tabella prevedendo di mantenere solo un'istanza come “rappresentante” di più istanze duplicate. Questo rappresentante ha nei metadati una marca, per ogni attributo che compone la chiave, che può essere specificata oppure “NO MARK”.

Esempi di query con operatori a livello di istanza dipendenti

Illustriamo un esempio di query in linguaggio KDDML che modella la discretizzazione di un attributo nominale “outlook”:

```

<KDDML_OBJECT>
  <KDD_QUERY name="discretize">
    <PPDISCRETIZATION xml_dest="discretized.xml">
      <PPTABLE_LOADER file_name="dddd.xml"/>
      <ATTR name="outlook"/>
      <CATEGORIC_DISCRETIZATION>
        <SPECIFY>
          <ORDERING>
            <SOURCE_ORDER>
              <USE_NOMINAL_ORDER/>
            </SOURCE_ORDER>
            <NUMERIC_DISCRETIZATION>
              <NUMERIC_DISCRETIZATION_METHODS>
                <EQUIWIDHT value="2" having="width"/>
              </NUMERIC_DISCRETIZATION_METHODS>
              <SMOOTH_BY>
                <ENUMERATION>
                  <EXCEPTION marca="mark"/>
                  <ENUMERATION_ELEMENT name="a"/>
                  <ENUMERATION_ELEMENT name="b"/>
                  <ENUMERATION_ELEMENT name="c"/>
                  <ENUMERATION_ELEMENT name="d"/>
                  <ENUMERATION_ELEMENT name="e"/>
                </ENUMERATION>
              </SMOOTH_BY>
            </NUMERIC_DISCRETIZATION>
          </ORDERING>
        </SPECIFY>
      </CATEGORIC_DISCRETIZATION>
    </PPDISCRETIZATION>
  </KDD_QUERY>
</KDDML_OBJECT>

```

```
</PPDISCRETIZATION>
</KDD_QUERY>
</KDDML_OBJECT>
```

La discretizzazione desiderata è di tipo categorico su un attributo nominale, si fa uso del suo rango numerico direttamente preso dall'ordine di enumerazione dei valori che l'attributo può assumere specificati nella sua definizione dell'*Instances* dei dati. Supponiamo ad esempio che l'attributo sia definito come `attribute outlook {sunny,overcast,rainy,windy}`; i corrispondenti ranghi numerici indotti dall'ordine di definizione saranno $\{0,1,2,3\}$ rispettivamente. Quando la modalità di etichettamento specificata è l'enumerazione l'attributo risultante dall'operazione avrà tipo nominale, in tutti gli altri casi (media, mediana, inf, sup) è di tipo numerico.

5.2.6 Operatori di passaggio di fase nel processo KDD: TABLE2PPTABLE e PTABLE2TABLE

Tutti i tipi validi per il linguaggio KDDML che rientrano nelle *kdd_query_PPTable* prevedono sia in input che in output delle tabelle estese ovvero delle *PPTable* per il linguaggio. Il tipo *PPTable* come già visto in 5.1 è definito come estensione di una *Table* mediante la presenza di due elementi aggiuntivi oltre allo schema dei dati. Tali nuovi elementi sono quello che riferisce lo schema dei metadati (per la parte dell'estensione della tabella) e quello che mantiene traccia delle operazioni di preprocessing a cui la tabella in esame è stata sottoposta (*History*). La fase di PreProcessing, quindi, è ben distinta dalle altre fasi del processo KDD per il tipo primitivo di input e di output a cui tale fase fa riferimento. L'inizio delle applicazioni di operazioni di preprocessing mediante *kdd_query_PPTable* è segnato dal preventivo passaggio a tipo *PPTable*

per i dati a cui ci si vuol riferire. L'uso di un tipo TABLE2PPTABLE, a livello di implementazione, corrisponde all'istanziamento dell'elemento relativo al metaschema e di quello relativo alla History descritti nella DTD di una KDDML_TABLE valida per il linguaggio KDDML. Tutto questo si traduce nella costruzione di un oggetto di tipo InternalPPTable usando i dati della InternalTable di input, e nell'inserimento nell'albero DOM del documento da restituire in output di un nuovo figlio per la *History* inizializzato con la prima operazione di preprocessing effettuata (TABLE2PPTABLE) e la sua descrizione "Start PreProcessing". In modo analogo, nel processo di estrazione di conoscenza, per passare a fasi successive a quella del PreProcessing, la parte di estensione può essere tagliata per far riferimento ai soli dati. L'uso di PPTABLE2TABLE in KDDML comporta, infatti, la costruzione di un nuovo oggetto *InternalTable* considerando solo la parte relativa ai dati della tabella estesa in input. La presenza di questi operatori permette di mantenere i contenuti dei metadati e della history sempre consistenti rispetto alle operazioni effettuate. Si evita, quindi, la possibilità di modificare a livello di istanza o di schema i dati senza lasciar traccia delle trasformazioni nei metadati, soprattutto per quelle operazioni che lasciano marche relative ad eccezioni seppur non richieste in modo esplicito. Supponiamo ad esempio di voler normalizzare i valori di un attributo numerico, se si usa una modalità locale in cui si specificano il min e il max stimati, per valori fuori dal range viene inserita una marca nei corrispondenti metadati. A livello di informazioni fornite all'utente, sarà chiaro dalla History che i dati sono stati modificati da un'operazione di normalizzazione, guardando i soli valori, invece, non è sempre chiaro che alcuni di essi non sono stati normalizzati. Questa situazione potrebbe essere fuorviante se non ci fosse l'informazione nei metadati in cui, comunque, viene posta l'attenzione su un'avvenuta eccezione per ogni

valore fuori dal range durante la normalizzazione.

Capitolo 6

Estensione del compilatore

Nell’accezione più generica, il compilatore è quel programma che traduce automaticamente codice scritto in un linguaggio in un altro. Un compilatore è composto generalmente dalle seguenti componenti: analizzatore lessicale, analizzatore sintattico, analizzatore dei tipi, generatore di codice. Nel sistema KDDML il compilatore si preoccupa di tradurre le richieste dell’utente modellate nel query language MQL esteso al DPL (*linguaggio sorgente*), in linguaggio KDDML (*linguaggio destinazione*) per produrre risultati usando l’interprete del sistema (vedi fig. 6.1). Il formato delle query nel linguaggio KDDML è quello dell’XML vincolato dalle DTD specificate (vedi 5.1). Ogni linguaggio va analizzato nei due aspetti che lo caratterizzano: la sintassi e la semantica. La sintassi rappresenta l’aspetto formale del linguaggio ed è definita con modalità uniformi per ogni linguaggio attraverso formalismi per la loro specifica (espressioni regolari, automi a stati finiti, grammatiche non contestuali, DTD, XML schema e altri). La semantica, invece, rappresenta il “significato” associato ad ogni frase del linguaggio e la modalità di definizione varia da linguaggio a linguaggio. La traduzione tiene conto sia della sintassi che della semantica del linguaggio sorgente e di quello destinazione.

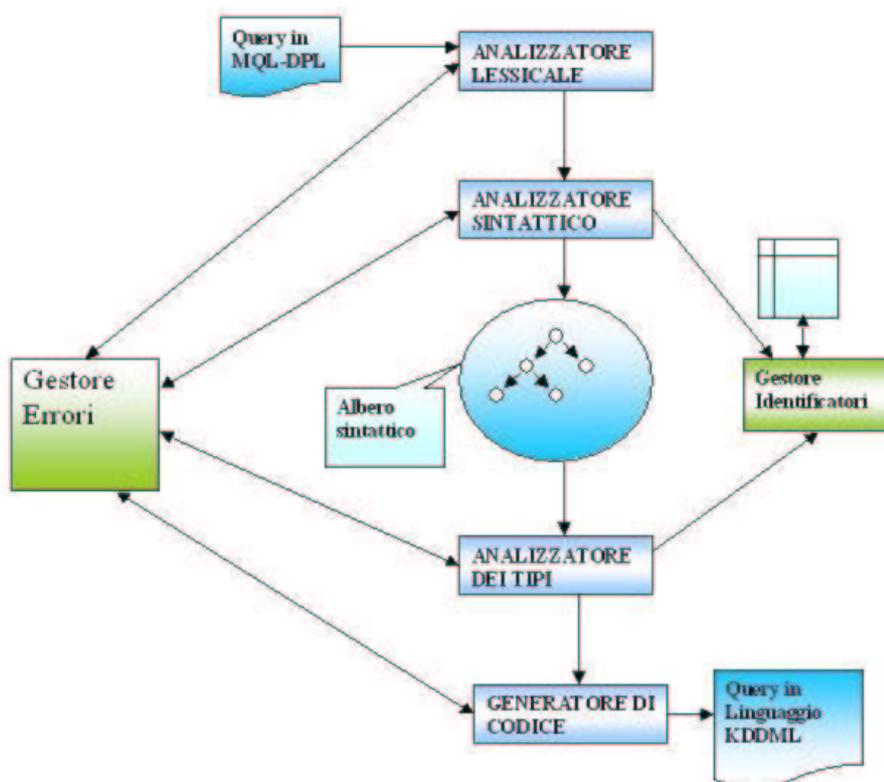


Figura 6.1: Compilatore: La struttura standard per la traduzione del linguaggio MQL-DPL in KDDML

Rispetto alle caratteristiche dei linguaggi di origine e di destinazione:

- la fase di analisi lessicale e sintattica dipendono soltanto dalla sintassi del linguaggio sorgente,

- la fase di analisi dei tipi dipende sia dalla sintassi che dalla semantica del linguaggio sorgente,
- la fase di generazione di codice dipende sia dalla sintassi che dalla semantica del linguaggio destinazione.

Data la complessità della definizione della semantica di un linguaggio, le fasi di analisi semantica dell'input e di sintesi dell'output non possono essere automatizzate, come invece avviene per le fasi di analisi lessicale e analisi sintattica. Per l'analisi lessicale, infatti, in genere si utilizzano dei programmi chiamati generatori di analizzatori lessicali che, a partire dalla specifica del lessico, generano automaticamente l'analizzatore lessicale corrispondente. Analogamente per l'analisi sintattica, si utilizzano dei programmi chiamati generatori di analizzatori sintattici che, a partire dalla specifica della sintassi, generano automaticamente l'analizzatore sintattico corrispondente. La realizzazione del compilatore per la traduzione delle interrogazioni nel formato del query language avviene sfruttando uno dei tanti tools esistenti per la compilazione: ANTLR (ANother Tool for Language Recognition). ANTLR è un linguaggio che fornisce un framework per la costruzione di riconoscitori, compilatori e traduttori generando codice Java, a partire dalla descrizione della grammatica del linguaggio sorgente definita secondo la notazione BNF (*Backus-Naur Form*). ANTLR è pienamente utilizzato nell'uso commerciale e nella comunità di ricerca. Le specifiche grammaticali messe a disposizione da ANTLR sono di 3 tipi: parser, lexer e tree parser; ogni specifica è stata utilizzata durante l'implementazione per realizzare una singola fase del processo di compilazione. ANTLR costruisce analizzatori di tipo top-down e richiede quindi una specifica LL(k) per le tre varianti grammaticali che, come vedremo, assumono di conseguenza una strutturazione molto simile. Il vantaggio principale di ANTLR è senza dubbio quello di essere uno strumento

completo che permette di combinare tutti gli strumenti utili nello sviluppo di un compilatore. Difatti, oltre a generare analizzatori lessicali e sintattici, basandosi rispettivamente su specifiche espresse con espressioni regolari e con grammatiche libere dal contesto, ANTLR permette anche di generare, durante la fase di analisi sintattica, l'albero sintattico del programma in ingresso (AST), e mette poi a disposizione una collezione di routines in grado di visitare e manipolare i nodi dell'albero. Queste librerie hanno permesso di effettuare type checking nonché generazione del codice. ANTLR offre al programmatore la possibilità di separare le specifiche lessicali e sintattiche in files distinti. Genericamente parlando, una descrizione ANTLR consiste di una collezione di regole lessicali e sintattiche che descrivono il linguaggio da riconoscere, e una collezione di azioni semantiche definite dall'utente che descrivono cosa fare quando la sequenza in input è stata riconosciuta. Ogni singola grammatica può essere definita in files multipli; la struttura grammaticale è identica per ogni tipo di specifica (lexer, parser, tree parser) ed è la seguente:

```
{definizione header e classi importate}

class <name> extends <Lexer | Parser | TreeParser>;

{inizializzazioni}

< sezione opzioni > < sezione tokens > < sezione regole >
```

Come si può osservare, ogni specifica grammaticale viene suddivisa in tre sezioni: sezione opzioni, sezione tokens e sezione regole. La prima contiene degli assegnamenti che specificano il comportamento dell'analizzatore (ad esempio definiscono il numero di simboli da scandire per individuare la regola

da applicare), mentre le altre due sezioni consentono di definire i simboli (terminali e non terminali) associati alla specifica grammaticale.

6.1 Analizzatore lessicale

L'analizzatore lessicale effettua una classificazione dei singoli elementi (parole chiave, operatori, identificatori, ecc) esaminando le frasi sorgente carattere per carattere. Per raggiungere questo obiettivo, l'analizzatore lessicale converte la sequenza di caratteri in input in una sequenza di tokens che diventeranno l'input per la fase di analisi sintattica. Per implementare un analizzatore lessicale è necessario specificare una classe separata che descrive come suddividere lo stream di caratteri in input in uno stream di tokens (vedi es.6.2).

Sfruttando il tool ANTLR è necessario implementare le tre sezioni differenti che specificano il comportamento dell'analizzatore. Tutte le regole lessicali che definiscono gli elementi del linguaggio DPL sono state inserite nel file `< lexer.g >` che raccoglie la definizione della grammatica MQLLexer. A partire da queste specifiche, ANTLR è in grado di generare il codice Java necessario per l'analisi lessicale di un documento DPL in ingresso (vedi fig.6.3).

6.1.1 Implementazione del file `lexer.g`

Sezione opzioni

La sezione opzioni contiene una serie di assegnamenti `< opzione = valore >` che determinano alcuni comportamenti di carattere generale associati al `lexer`. Nel nostro contesto, ad esempio, attraverso l'opzione `< Language = 'Java' >` è stato possibile scegliere il linguaggio associato all'analizzatore

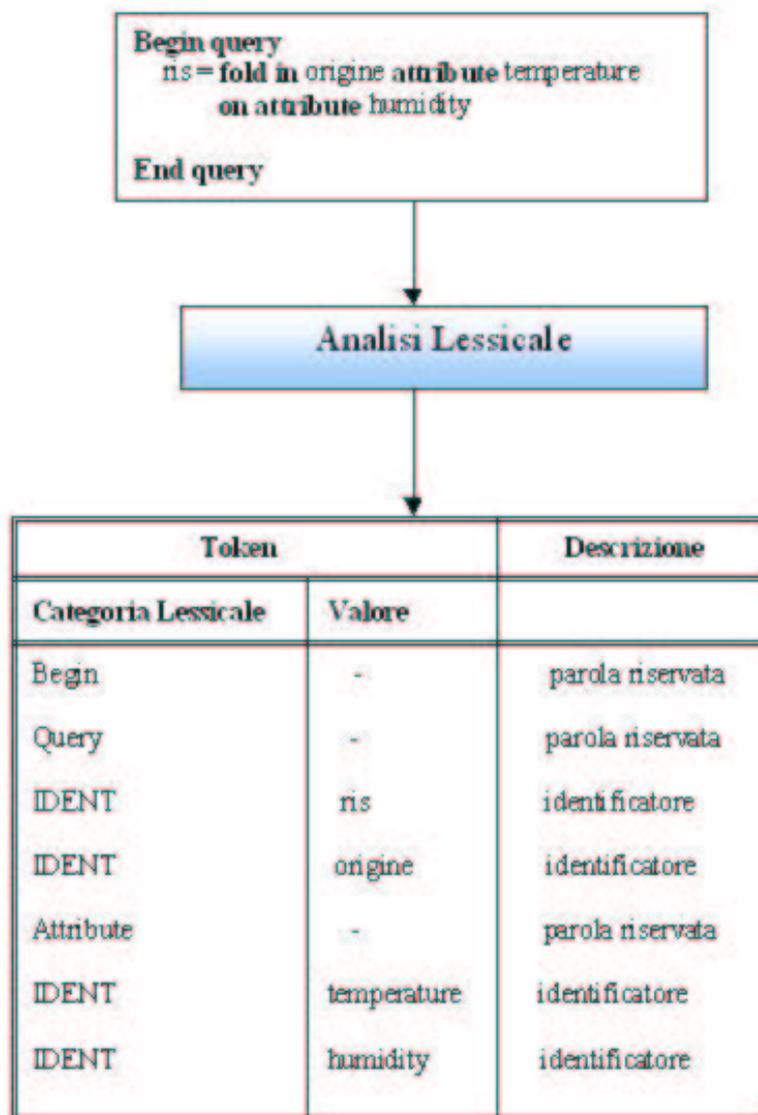


Figura 6.2: Esempio di analisi lessicale su una query in DPL.

che viene generato in output, mentre si utilizza l'opzione $\langle k = 3 \rangle$ per settare la profondità del lookahead, ossia il numero di caratteri che saranno esaminati per selezionare le regole grammaticali alternative nella generazione di un token a partire dal carattere corrente. Un'opzione utilizzata è stata

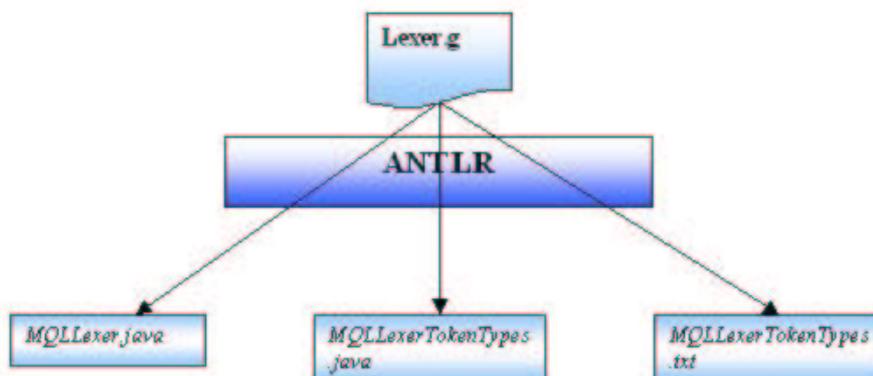


Figura 6.3: ANTLR per l'analisi lessicale.

inoltre quella che permette di disabilitare la distinzione fra lettere maiuscole e minuscole: $\langle caseSensitive = false \rangle$. Con questa soluzione, il linguaggio, ad esempio, riesce a considerare come sinonimi le parole chiave “begin”, “Begin” o “BEGIN”. Anche gli identificatori subiscono lo stesso trattamento per mezzo di particolari funzioni Java che convertono una stringa in lettere minuscole prima di memorizzarla all'interno della tabella degli identificatori.

Sezione tokens

La sezione tokens è stata utilizzata per definire le parole riservate del linguaggio. Si compone di un insieme di coppie del tipo `FROM = ‘from’` dove FROM è il token vero e proprio, mentre la stringa “from” costituisce il suo valore. Ogni token così definito viene inserito da ANTLR in una tabella,

detta tabella dei letterali, associata con l'analizzatore lessicale, che è generata e gestita con strategia hash.

Sezione regole

Le regole lessicali e sintattiche ANTLR seguono lo schema:

```
rulename [parametri formali] returns [type id1,...,type idn]
options {}
```

```
{inizializzazioni}:
    alternativa 1 |
    ...
    alternativa n;
exception
    catch[Exception e] {}
```

Le regole lessicali effettuano un matching dei caratteri dello stream in ingresso e restituiscono uno o più valori. Ogni regola ha un nome e opzionalmente ha :

- un insieme di argomenti,
- delle azioni di inizializzazione eseguite al momento di applicare la regola,
- una clausola “throws”;

Consideriamo ad esempio la regola definita nella grammatica del lexer MQL-DPL che permette il riconoscimento di un identificatore:

```
IDENT
options {
```

```

testLiterals = true;
paraphrase = ‘‘an identifier’’;
}:
((‘a’..‘z’|‘_’|‘$’)
(‘a’..‘z’|‘0’..‘9’|‘_’|‘$’)*
“.arff”          {$setType(FILE_ARFF);} |
“.xml”           {$setType(FILE_XML);} |
“.xqu”           {$setType(FILE_XQU);} |
“.sql”           {$setType(FILE_SQL);}?
|
“{all}”          {$setType(SET_STRING);});

```

Con ANTLR si possono associare opzioni all’intera grammatica oppure solo ad una particolare regola. Una sezione apposita è quindi dedicata a contenere le opzioni da utilizzare all’interno della regola. Con l’opzione `<paraphrase = ‘‘an identifier’’ >` viene impostato il trattamento degli errori: ogni volta che il token `IDENT` è oggetto di un errore (lessicale o sintattico), nel messaggio di errore restituito da ANTLR viene sostituita la parola “`IDENT`” con la frase specificata nell’opzione, “`an identifier`”. Questo rende più trasparente e comprensibile il messaggio all’utente. Con l’opzione `<testLiterals = true >` si riesce invece a distinguere le parole riservate del linguaggio, definite nella sezione `tokens` e memorizzate all’interno dell’apposita tabella, dagli identificatori. Infatti, come si può osservare, la definizione data per un identificatore va in conflitto con la definizione di una parola riservata che non è altro che una stringa di caratteri. Se l’opzione `testLiterals` è attivata, tuttavia, ANTLR provvede a generare codice Java per testare se il token appena riconosciuto è presente o meno all’interno della tabella. Se questa ricerca ha successo, il token individuato viene considerato come parola riservata, altrimenti come identificatore. Se si passa ad analizzare il corpo della regola, si osserva che contiene definizioni espresse con la notazione delle espressioni regolari. In questo caso, si specifica che un identificatore è una

sequenza di simboli che iniziano con una lettera dell'alfabeto e possono essere seguite da lettere o cifre. Opzionalmente può essere presente un'estensione “.arff”, ad esempio, che identifica un riferimento ad un file in formato “arff” contenuto nel repository dei dati. In quest'ultimo caso, alla regola viene associata un'azione semantica (racchiusa tra parentesi graffe) che contiene del codice Java. Se durante l'analisi del testo in ingresso l'analizzatore individua un identificatore contenente l'estensione “.arff”, viene richiamato il metodo specificato tra parentesi graffe (`setType(FILE_ARFF)`), che assegna al token individuato il tipo `FILE_ARFF` anziché il tipo `IDENT`. L'esempio appena visto mostra quindi come viene gestito in ANTLR un conflitto tra tokens. Il token `FILE_ARFF` può essere infatti considerato come un sottocaso del token più generico `IDENT` ed entrambi sono definiti con un'unica regola lessicale. Se si utilizzassero due regole differenti, sarebbe stato impossibile per l'analizzatore individuare quale regola applicare perché prima dell'estensione “.arff” possono essere presenti un numero arbitrario di caratteri. In modo del tutto analogo vengono risolti tutti gli altri conflitti presenti nella definizione degli elementi del linguaggio, ad esempio quello fra i tokens `FLOAT` e `INT` che rappresentano rispettivamente un numero in virgola mobile ed un numero intero (che è un caso particolare di `FLOAT`). Oltre a `setType(t)` appena visto, ANTLR mette a disposizione altri metodi per interrogare e manipolare i tokens del linguaggio. I più importanti sono `getType()` per leggere il tipo associato al token e `getText()` e `setText(v)` rispettivamente per leggere ed assegnare un valore ad un tokens. I commenti e gli spazi bianchi vengono trattati in modo simile associando al token individuato un tipo speciale: `<Token.SKIP >` che indica all'analizzatore di ignorare il token durante l'analisi e di non trasmetterlo al parser.

6.1.2 File generati da ANTLR per l'analisi lessicale

A partire dalle specifiche del file `lexer.g`, ANTLR è in grado di generare il codice Java necessario per l'analisi lessicale di un documento in linguaggio MQL-DPL in ingresso. Analizziamo i file generati da ANTLR e le funzionalità da essi supportate.

MQLLexer.java

Classe che contiene l'implementazione vera e propria dell'analizzatore lessicale. La classe raccoglie un metodo per ogni singola regola specificata all'interno della grammatica lessicale che esegue le azioni descritte nella regola e restituisce il token associato, più un metodo chiave “`public Token nextToken()`”. Questo metodo effettua uno *switch* sui simboli in ingresso per stabilire quale metodo (regola lessicale) richiamare.

MQLLexerTokenTypes.java

Contiene una interfaccia che elenca tutti i nomi dei tokens definiti all'interno del lexer, definiti nella sezione tokens e sezione regole, ognuno identificato da un valore intero. ·

MQLLexerTokenTypes.txt

Contiene la lista dei nomi dei tokens in formato testuale.

6.2 Analizzatore sintattico

Un analizzatore sintattico (o più brevemente parser) è dunque un programma che, prendendo in ingresso la definizione della grammatica per il

linguaggio da analizzare e l'insieme di tokens di un suo programma sorgente, è in grado di stabilire se tali tokens sono generabili dalla grammatica e quindi se il programma è sintatticamente valido, producendo così l'albero sintattico che lo rappresenta. Tale processo è comunemente noto come parsing. Per implementare un analizzatore sintattico è necessario specificare una classe separata che descriva le regole sintattiche da applicare per riconoscere lo stream di tokens in ingresso. Unitamente al riconoscimento dei tokens, l'analizzatore sintattico genera l'albero sintattico che rappresenta in forma intermedia la query in ingresso (vedi fig.6.4).

Una scelta di progetto effettuata è stata quella di delegare al parser il compito di inizializzare ogni identificatore individuato durante il processo di analisi all'interno dell'apposita tabella degli identificatori. Tale servizio si trasforma in una semplice richiesta diretta al gestore degli identificatori, il quale si preoccupa di controllare la presenza dell'identificatore all'interno della tabella, segnalando l'esito dell'operazione. Come per l'analizzatore lessicale, è stato definito un file `parser.g` a partire dal quale generare codice Java mediante ANTLR (vedi fig.6.5). La struttura di tale file prevede le stesse tre sezioni del `lexer.g`: opzioni, token e regole. La regola associata a ciascuna tipologia di query contiene, per la gestione degli identificatori, un controllo del tipo:

```
match(IDENT);

    if (SymbolTable.contains(i.getText()))
    {
        // controllo che non sia gia' presente
        num_of_errors++;
        String msg = "Indentifier <"+i.getText()+"> has been already used.\n";
        error_message = error_message + msg + "\n";
        //return num_of_errors;
    }
```

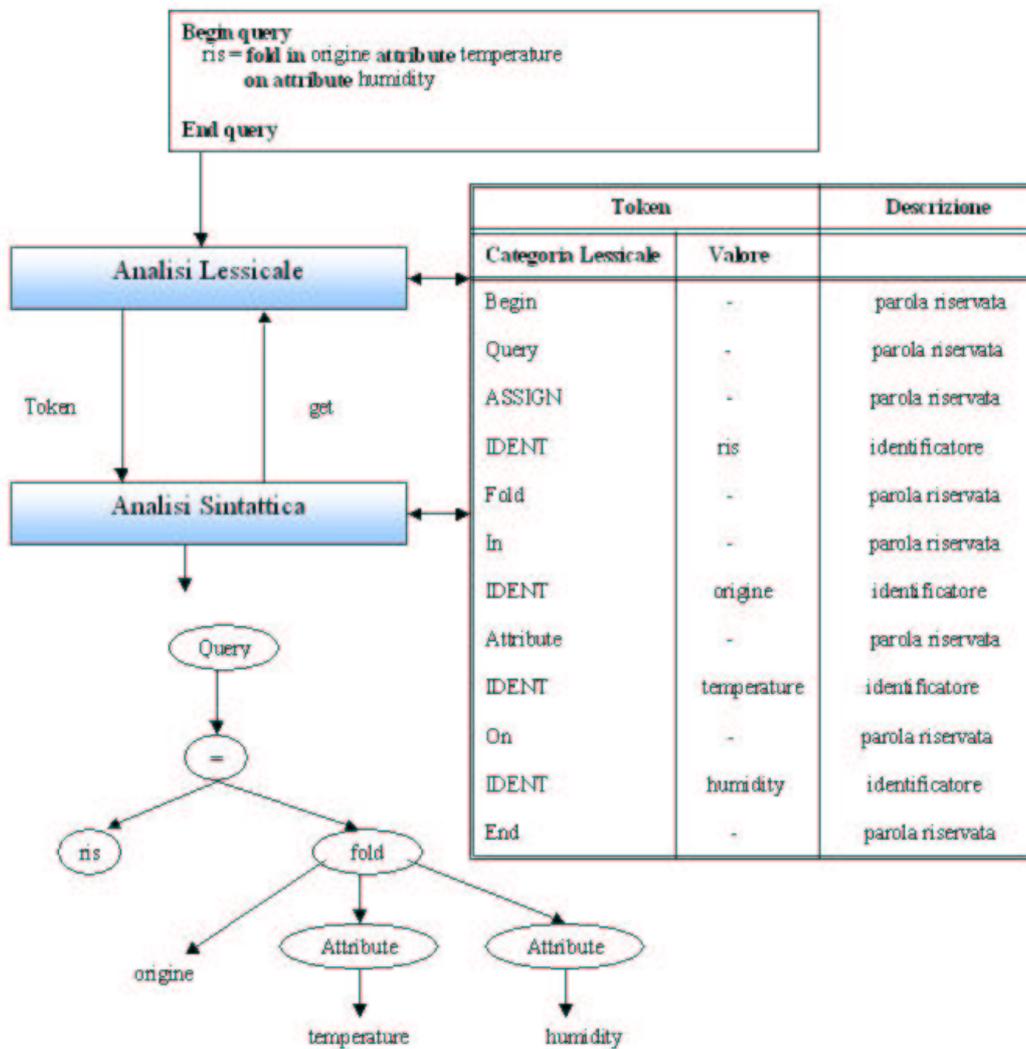


Figura 6.4: Esempio di analisi sintattica su una query in DPL.

```

else
{
    SymbolTable.insert(i.getText());
}

```

dove `SymbolTable` è la classe che realizza il gestore degli identificatori.

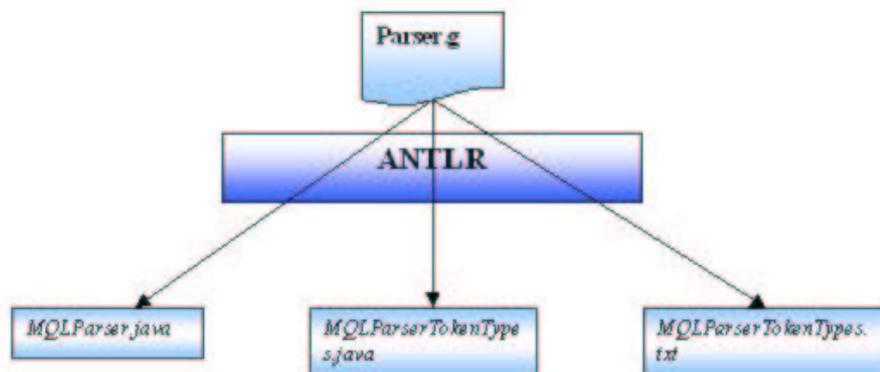


Figura 6.5: ANTLR per l'analisi sintattica.

6.2.1 Implementazione del file parser.g

Sezione opzioni

Così come per l'analizzatore lessicale, anche per un analizzatore sintattico è possibile specificare una serie di assegnamenti `< opzione = valore >` che determinano alcuni comportamenti di carattere generale associabili al parser. Ad esempio, la profondità del *lookahead*, che si riferisce al numero dei tokens in ingresso scanditi a partire da quello corrente, è controllata dall'opzione `< k = 6 >`. Per l'abilitazione della costruzione degli alberi AST (Abstract Syntax Tree) da parte dell'analizzatore è usata l'opzione `< buildAST = true >` che indica ad ANTLR di produrre il codice necessario alla generazione degli AST durante l'analisi sintattica.

Sezione regole

Le regole seguono lo stesso schema descritto in 6.1.1. In particolare nell'analisi sintattica sono tutte scaturite dalla regola principale per il programma in MQL-DPL. Tale regola ha nome “*program*” ed definita nel modo seguente:

```
program returns [int err] throws ParserErrorException
{
    err = 0;
    SymbolTable.removeAll();
}:
    BEGIN! QUERY^
    query
    END! QUERY!
    {
        err = num_of_errors;
        if (num_of_errors >0)
            throw new ParserErrorException(err, error_message);
    }

exception // for rule program
catch [RecognitionException ex] {
    // riporta l'errore sullo std output
    reportError(ex.toString());
    // incrementa il contatore degli errori sintattici
    num_of_errors++;
    error_message = error_message + ex.toString() + "\n";
    throw new ParserErrorException(num_of_errors, error_message);
}
catch [TokenStreamException ex] {
    // riporta l'errore sullo std output
    reportError(ex.toString());
    num_of_errors++;
    error_message = error_message + ex.toString() + "\n";
    throw new ParserErrorException(num_of_errors, error_message);
}
```

Nella parte dedicata all’inizializzazione si provvede a ripulire la tabella degli identificatori da eventuali residui di un precedente processo di parsing e ad azzerare il contatore degli errori. Nel corpo della regola, i simboli espressi con lettere maiuscole (BEGIN, QUERY, END) rappresentano tokens del linguaggio definiti durante l’analisi lessicale, mentre il simbolo “query” identifica una nuova regola introdotta all’interno della grammatica corrente. Una query è definita come

```
query:
    a:LET! q1
    q1;
q:
    q1 q2;
```

Per l’analisi sulle query DPL è stata aggiunta una nuova tipologia di query al tipo “q1” chiamata “q1_ppTableQuery”. Questa nuova regola consiste in un identificatore che si riferisce ad una tabella estesa oppure ad un operatore che restituisce una tabella estesa. Passando all’azione semantica associata alla regola, essa è racchiusa tra parentesi graffe e sarà eseguita all’uscita dell’analisi della regola stessa. In questo caso, si provvede ad assegnare alla variabile “err”, parametro di output della regola, il contenuto della variabile “num_of_errors”. Quest’ultima è una variabile globale definita ed istanziata nella sezione di inizializzazione dell’intera grammatica. Il suo valore viene incrementato in conseguenza ad un errore riscontrato nel processo di analisi, che viene rilevato dalla regola attraverso la sezione exception. Il meccanismo delle eccezioni fornito da ANTLR consente anche di gestire l’individuazione multipla degli errori cui abbiamo accennato nel capitolo precedente. Quando la regola individua un errore (ad esempio per un token inatteso), il costrutto “try-catch” di Java permette infatti di eseguire delle istruzioni per ripristinare uno stato corretto del programma. Nell’analizzatore sintattico,

all'interno del codice `try-catch` si può utilizzare il metodo, definito in ANTLR, `consumeUntil(SetOfToken)` che permette di scandire i tokens in ingresso finché non viene individuato un token che fa parte dell'insieme i simboli `end`, `in` e `;`. Appena l'eccezione è stata individuata, si segnala l'errore, si incrementa il contatore degli errori `num_of_errors` e infine si ripristina il normale funzionamento del parser a partire dal token immediatamente successivo ad uno dei tre simboli di arresto, proseguendo l'analisi a partire dalla regola opportuna. ANTLR riesce a generare l'albero sintattico astratto (AST) simultaneamente al processo di analisi. ANTLR infatti, consente di generare degli AST come rappresentazione intermedia dei programmi in input che ne rispecchiano le strutture sintattiche. In aggiunta, ANTLR offre la possibilità di associare automaticamente azioni semantiche ai nodi dell'AST; tali azioni corrispondono a codice Java che può essere eseguito durante il processo di visita dell'albero. ANTLR aiuta nella costruzione degli AST fornendo una notazione grammaticale che suggerisce all'analizzatore quali tokens devono essere trattati come nodi interni, quali come foglie e quali devono essere ignorati rispetto alla costruzione dell'albero. La distinzione è fatta mediante l'uso di due simboli diversi:

- Il simbolo `^` specifica ad ANTLR di considerare il token come radice dell'albero che si sta costruendo,
- Il simbolo `!` specifica ad ANTLR di ignorare il token durante la costruzione dell'albero.

Si ha, tornando all'esempio dell'operatore `fold`, la regola:

```
/**  
 * operatore fold:  
 * ripiega un attributo su un altro in una ptable  
 */
```

```

fold_stm:
    FOLD^ IN!
    (identifier | LPAREN! ppTableQuery RPAREN!)
    attr_stm ON! attr_stm;

attr_stm:
    ATTRIBUTE^ identifier ;

```

L'albero sintattico generato dall'analizzatore sintattico è quello illustrato dalla figura 6.4. Come si può osservare nell'esempio, con questa notazione, comunichiamo ad ANTLR di trattare il token `FOLD` come operatore e quindi come nodo interno dell'albero sintattico. I token `LPAREN` e `RPAREN` che rappresentano le parentesi e che esprimono l'annidamento rispetto al primo operando associato, sono invece ignorati durante il processo di generazione dell'albero. Passando al livello più basso si generano i nodi per `ATTRIBUTE`, anche essi trattati come nodi interni dell'AST da generare; infine ci sono le foglie relative agli identificatori. ANTLR permette inoltre di visualizzare l'albero prodotto durante il parsing utilizzando una notazione LISP del tipo $(operatore\ operando_1\ operando_n)$, mediante il metodo `“.toStringList()”` della classe fornita per la gestione degli AST. Ad esempio, il risultato del parsing e l'albero generato in notazione LISP nel caso della query di fig.6.4 è:

```

... MQL ---> KDDML Compiler. Version 2.0.

Insert < ris >in symbol table.
Parsing successfully!!!

Result of Parsing:
( query ( = ris ( fold origine ( attribute temperature ) ( attribute humidity ) ) ) )

```

6.2.2 File generati da ANTLR per l'analisi sintattica

A partire dalle specifiche del file `parser.g`, ANTLR è in grado di generare il codice Java necessario per l'analisi sintattica di un documento in linguaggio MQL-DPL in ingresso. Analizziamo i file generati da ANTLR e le funzionalità da essi supportate.

MQLParser.java

che contiene la classe Java che implementa l'analizzatore sintattico vero e proprio.

MQLParserTokenTypes.java

contiene una interfaccia che elenca tutti i nomi dei tokens definiti per il linguaggio. In questo caso, l'elenco coincide con i tokens prodotti durante l'analisi lessicale che sono importati direttamente dall'interfaccia *MQLLexerTokenTypes.java*.

MQLParserTokenTypes.txt

contiene la lista dei nomi dei tokens in formato testuale.

6.3 Analizzatore dei tipi

All'analizzatore dei tipi spetta il controllo statico, questo modulo, infatti, ha lo scopo di assicurare che certi errori di programmazione siano individuati e segnalati prima di eseguire il programma. Tra questi errori figurano sicuramente gli errori semantici, segnalati non appena un operatore viene applicato ad un operando incompatibile. Generato l'albero sintattico (AST) che rappresenta la query in ingresso, è necessario un metodo per etichettare

i nodi dell'albero con azioni semantiche che possono essere eseguite durante la visita dell'albero stesso. Per questo, ANTLR mette a disposizione una notazione speciale per associare del codice Java ai nodi dell'albero sintattico. E' necessario implementare un tipo di grammatica, chiamata **tree parser**, che visita la rappresentazione intermedia prodotta dal parser durante la fase precedente ed effettua il controllo dei tipi sulla query in ingresso. L'implementazione è fatta nel file `typeChecker.g` dove si effettua un controllo sulla correttezza dei tipi degli identificatori presenti nella query, provvedendo anche ad effettuare un controllo sulla presenza dei files nei repository relativi quando riferiti all'interno della query. Viene effettuata una analisi bottom-up dell'albero sintattico restituito dal Parser, memorizzando opportunamente il tipo di ogni identificatore trovato nella tabella dei simboli inizializzata dall'analizzatore sintattico. E' sufficiente richiamare un metodo, "`checkType(id, tipo)`", definito nella classe `SymbolTable` per verificare che il tipo dell'identificatore sia quello atteso dal contesto dell'operatore, se il tipo è quello atteso, non resta che assegnare all'identificatore il tipo, restituito in output da questa regola; in caso contrario, viene restituito un messaggio di errore di tipo. L'informazione di tipo via via calcolata può essere propagata ai livelli superiori dell'AST dove viene utilizzata per calcolare il tipo dei nodi antenati, come richiesto da una visita di tipo bottom-up. L'intero processo termina solo quando viene raggiunto un nodo che identifica un operatore di assegnamento. In questo caso viene effettuata la registrazione fisica del tipo calcolato nella tabella degli identificatori utilizzando il metodo "`addType(String chiave, int tipo)`" che prende come parametri una stringa contenente l'identificatore da aggiornare e il tipo da assegnare.

6.3.1 Implementazione del file typeChecker.g

Sezione regole

La struttura di una regola per questo tipo di grammatica è analoga a quella delle regole precedentemente viste, ma nel corpo una produzione alternativa è composta da una lista di elementi dove ogni elemento può essere uno degli items presenti in una regola ANTLR, come per gli altri analizzatori, ma con in più la possibilità di esprimere un elemento tree pattern della forma:

$\#(radice\ figlio_1, figlio_2, \dots, figlio_n)$ per controllare il tipo degli elementi figli prima di assegnare il tipo al nodo radice.

Per ogni token relativo agli operatori introdotti dal DPL c'è una regola associata per il controllo dei tipi. Ad esempio, sempre per il token "FOLD", la regola associata è :

```
//----- FOLD -----
/**
 * controllo dei tipi per FOLD; <BR>
 * se il figlio ritorna tipo ppTUPLE allora si restituisce tipo ppTUPLE,
 * altrimenti si restituisce errore.
 */

fold returns [short type] throws TypeErrorException,
NoFileExistWarning {
    type = MQLTypes.VOID;
    String i;
    short a;}:
    #(FOLD
(a=ppTableQuery
    {type = MQLTypes.PP_TUPLE;}
|
i=identifier
    {type = SymbolTable.checkType(i,MQLTypes.PP_TUPLE,MQLTypes.PP_TUPLE);}
) );
```

Nella sezione di inizializzazione, racchiusa tra parentesi graffe, vengono definite ed inizializzate tutte le variabili utilizzate all'interno del corpo della regola. Nel nucleo della regola, viene definito l'elemento tree pattern associato all'operatore in questione, che ha come radice il token FOLD definito nella sezione tokens durante l'analisi lessicale; il suo unico figlio contiene un oggetto che rappresenta un tabella estesa ottenuta in due modalità distinte:

- attraverso una generica `ppTableQuery`, ossia un annidamento tra operatori, ampio quanto si vuole, ma che restituisce un oggetto di tipo `PP_TUPLE` (tabella estesa). In questo caso, l'analizzatore effettua una analisi ricorsiva del sottoalbero che contiene la `ppTableQuery`. Il risultato di questa analisi viene memorizzato nell'identificatore "a". Appena l'analisi del sottoalbero è terminata e il tipo risultante è noto, l'analizzatore può eseguire il codice Java responsabile dei necessari controlli di tipo relativi a questo ramo dell'albero. In questo caso, è sufficiente assicurarsi che "a" abbia tipo `PP_TUPLE`. Se il tipo è quello atteso, non resta che assegnare all'identificatore *type*, contenente il valore restituito in output da questa regola, il tipo tabella. In caso contrario, viene restituito un errore di tipo.
- utilizzando un identificatore che riferisce un file XML contenente un `KDDML_OBJECT`. Le informazioni sul tipo associato all'identificatore sono contenute all'interno della tabella degli identificatori. E' sufficiente a questo punto richiamare un metodo, `checkType(id, tipo)`, definito nella classe `SymbolTable` per verificare che il tipo dell'identificatore sia quello atteso dal contesto dell'operatore (in questo caso `PP_TUPLE`). In base al risultato ottenuto si procede in modo analogo al caso precedente.

6.3.2 File generati da ANTLR per l'analisi dei tipi

A partire dalle specifiche del file `typeChecker.g`, ANTLR è in grado di generare il codice Java necessario per l'analisi dei tipi di un documento in linguaggio MQL-DPL in ingresso. Analizziamo i file generati da ANTLR e le funzionalità da essi supportate (vedi fig.6.6).

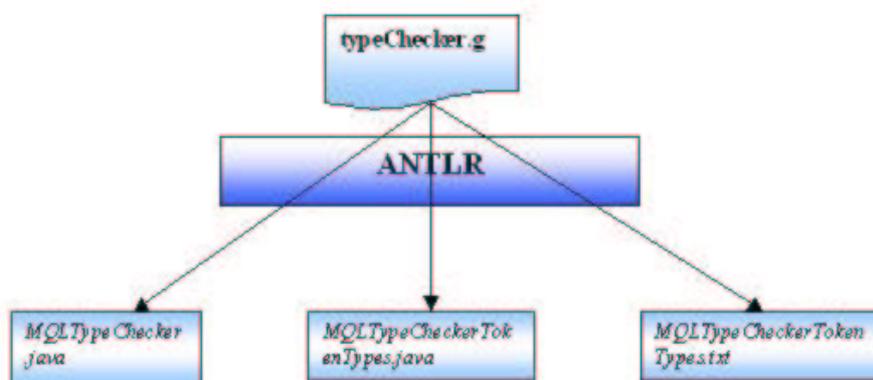


Figura 6.6: ANTLR per l'analisi dei tipi.

MQLTypeChecker.java

che contiene la classe Java che implementa l'analizzatore dei tipi vero e proprio.

MQLTypeCheckerTypes.java

contiene una interfaccia che elenca tutti i nomi dei tokens definiti per il linguaggio. In questo caso, l'elenco coincide con i tokens prodotti durante l'analisi lessicale che sono importati direttamente dall'interfaccia *MQLLexer-TokenTypes.java*.

MQLParserTokenTypes.txt

contiene la lista dei nomi dei tokens in formato testuale.

6.4 Generazione del Codice

La fase della generazione di codice dipende sia dalla sintassi che dalla semantica del linguaggio destinazione, prendendo come struttura di riferimento della query in linguaggio sorgente l'albero sintattico generato e tipato dalle fasi precedenti. Nel caso del sistema KDDML il linguaggio destinazione è il linguaggio KDDML nel formato XML vincolato dalla grammatica espressa nelle DTD specificate in 5.1.1. Per la parte di interesse del query language in questo lavoro relativamente al DPL, nel linguaggio KDDML i tipi coinvolti sono quelli che appartengono all'entità “% kdd_query_table ” della categoria “KDD_QUERY” inseriti come descritto in 5.1.1. La regola generale su cui è stata basata l'intera fase di traduzione è: “*Generare tante KDD_QUERY quante sono le istruzioni di assegnamento della query MQL in ingresso.*” La struttura fisica del programma di output del traduttore è quella del formato XML. Una delle caratteristiche principali di XML è la possibilità di assegnare una struttura ad un documento. Ogni documento XML comprende una *struttura logica* ed una *struttura fisica*. La prima è simile ad un modello sui dati e indica quale sia la forma degli oggetti da inserire nel documento.

Questa struttura può essere definita in modo formale attraverso un particolare formalismo di XML noto con il nome di Document Type Definition (DTD). La struttura fisica, invece, contiene i dati effettivamente utilizzati dal documento suddivisi in una serie di oggetti ciascuno composto da uno o più elementi logici che costituiscono l'aspetto più comune di markup di XML. In generale esiste un elemento principale che costituisce la radice del documento e che non figura nella portata di nessun altro elemento; tutti gli altri elementi devono regolarmente annidarsi l'uno dentro l'altro. Un elemento è a sua volta composto da un TAG di apertura (detto "start-TAG") della forma "<nome elemento>", un TAG di chiusura (detto "end-TAG") della forma "</nome elemento>" e un numero variabile di attributi che consentono di associare valori ad un elemento senza che siano considerati parte del contenuto dell'elemento stesso. Ogni attributo ha valore atomico e assume la forma `nome = 'valore'`. La cosa fondamentale è che gli elementi possono essere strutturati e contenere all'interno della loro portata altri elementi. Una caratteristica introdotta in XML è stata inoltre quella di poter definire elementi vuoti, cioè senza alcun contenuto o attributo, attraverso la forma speciale "<nome elemento/>". Tornando all'esempio della query DPL:

```
Begin query
  ris = fold in origine attribute temperature
        on attribute humidity
End query
```

il cui albero sintattico generato dal parser è quello illustrato in fig.6.4, la DTD da rispettare del file XML da restituire in output è:

```
<!ELEMENT PPFOLDING_QUERY ((%kdd_query_PPtable;),ATTR,ATTR)>
<!ATTLIST PPFOLDING_QUERY xml_dest CDATA #IMPLIED>
```

```
<!ELEMENT ATTR EMPTY>
<!ATTLIST ATTR name CDATA #REQUIRED>
```

In linguaggio KDDML, quindi, la struttura fisica della query che il traduttore deve essere in grado di generare sarà:

```
KDDML_OBJECT>
  <KDD_QUERY name="origine">
    <PPFOLDING_QUERY xml_dest="ris.xml">
      <PPTABLE_LOADER file_name="dddd.xml"/>
      <ATTR name="temperature"/>
      <ATTR name="humidity"/>
    </PPFOLDING_QUERY>
  </KDD_QUERY>
</KDDML_OBJECT>
```

La traduzione avviene eseguendo una visita *depth-first* dell'albero sintattico come in fig. 6.7.

Questo tipo di visita comincia dalla radice e visita ricorsivamente tutti i figli di ogni nodo procedendo con una valutazione da sinistra verso destra. Quando sono stati visitati tutti i figli del nodo corrente, la ricerca prosegue verso i fratelli del nodo stesso, se presenti, altrimenti continua al livello superiore. In questo modo ogni nodo diverso da una foglia è attraversato per due volte durante il processo: la prima nella fase discendente della valutazione durante lo spostamento verso le foglie dell'albero, mentre la seconda quando abbiamo concluso la visita di tutti i nodi associati a quel livello e ritorniamo verso il livello superiore. Di conseguenza, possiamo associare ad ogni nodo dell'albero che sia diverso da una foglia, due regole semantiche distinte le quali che provocano diversi effetti laterali:

- se un nodo interno viene attraversato durante l'analisi discendente, allora si procede alla valutazione della prima regola semantica che provvede ad aprire il TAG relativo al nodo.

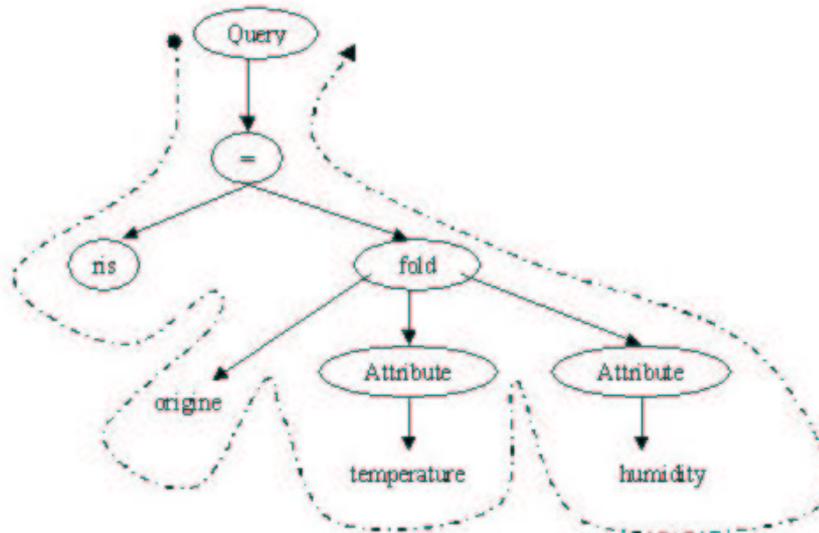


Figura 6.7: Esempio di visita Depth-first dell'albero sintattico nella generazione di codice.

- al contrario, se un nodo interno viene visitato durante la fase ascendente allora si procede alla valutazione della regola semantica il cui scopo è chiudere il TAG relativo al nodo.

In entrambe le fasi se il nodo in analisi è il padre di un nodo foglia, si provvede a settare un particolare attributo, oppure a generare una riga XML contenente un TAG vuoto se il nodo foglia riferisce un elemento vuoto. ANTLR prende in considerazione l'AST come uno stream di tokens bidimensionali e mette a disposizione una interfaccia (nota come interfaccia AST) che fornisce i metodi per visitare e manipolare i nodi dell'albero sintattico. ANTLR considera allo stesso modo nodi ed alberi, in quanto un albero viene visto semplicemente come una coppia composta dal nodo radice e dal puntatore alla lista dei suoi figli. Per l'implementazione della visita (nella procedura ri-

corsiva “`serialize`” della classe `translator.java`) sono stati pesantemente utilizzati i metodi `getFirstChild()` e `getNextSibling()`, definiti nell’interfaccia `AST`, che restituiscono rispettivamente il primo figlio associato ad un nodo ed il puntatore alla lista dei fratelli del nodo stesso.

Capitolo 7

Esempio completo di preprocessing nel sistema KDDML

Illustriamo di seguito un esempio di dataset che risulta inadatto ad essere sottoposto alla fase di data mining applicazione se non attentamente preelaborato. Il file preso come riferimento ha dati su 1500 clienti di una compagnia e contiene una serie di problemi da individuare e correggere. Una porzione di dati disponibili è illustrata in fig. 7.1. Iniziamo con i valori dell'attributo Social Security Number (SSN). E' giusto supporre che tutti i 1500 clienti siano clienti diversi, pertanto tutti i valori di SSN devono essere diversi. Per assicurarsi che queste condizioni siano state rispettate si può sottoporre l'intero dataset a un'operazione di marcatura dei duplicati usando come chiave unica l'attributo SSN. Usando il query language MQL-DPL, si formalizza questo tipo di richiesta tramite la query:

```
begin query
let
    dataset = loadARFF cleansing.arff;
```

Output Data								
Customer	SSN	Birthdate	Age	Region	CredCardUser	Income	Purchase	AmtSpent
1	539-84-9599	10/26/44	59	East	0	62900	4	2080
2	444-05-4079	01/01/32	67	West	1	23300	0	0
3	418-18-5649	08/17/73	25	East	1	48700	8	3990
4	065-63-3311	08/02/47	51	West	1	137600	2	920
5	059-58-9566	10/03/48	50	East	0	101400	2	1000
6	443-13-8685	03/24/60	39	East	0	139700	1	550
7	638-89-7231	12/02/43	55	South	1	50900	3	1400
8	202-94-6453	11/08/74	24	South	1	50500	0	0
9	266-29-0308	09/28/67	31	North	0	151400	2	910
10	943-85-8301	07/05/65	33	W	0	88300	2	1080
11	047-07-5332	11/13/64	34	North	0	120300	3	1390
12	503-14-3400	03/21/34	65	East	0	25800	1	560
13	690-97-5817	12/29/60	38	West	0	65500	1	410
14	614-59-6703	08/01/72	26	South	1	54400	9999	9999
15	704-98-5642	05/01/52	46	North	0	54500	7	3570
16	588-92-9590	08/29/39	59	South	0	143500	2	1070
17	267-58-0440	04/05/57	42	East	1	40500	2	950
18	833-81-4506	11/01/67	31	West	1	61900	2	930
19	645-20-8840	03/02/58	41	West	1	106300	2	1010
20	700-29-4704	01/10/33	66	South	0	133700	3	1570
21	443-16-6366	11/28/38	60	S	1	101900	3	1560

Figura 7.1: Dataset da sottoporre a preprocessing.

```

dataset_pp = table2pptable dataset
in
mark_duplicates_SSN = markduplicates in
    dataset_pp
    on attribute SSN withcode "not_unique"
end query

```

Lo scopo di questa query è quello di:

- prelevare il dataset iniziale a disposizione, convertirlo nel tipo “tabella estesa” del linguaggio KDDML (“table2pptable”)
- applicare l’operazione di marcatura dei duplicati usando come chiave di identificazione il valore dell’attributo “SSN”. Ogni duplicato identificato sarà marcato nella parte dei metadati relativi all’attributo specificato con il codice “not_unique”

Per l'esecuzione di tale query mediante l'interprete del sistema KDDML, essa viene tradotta mediante il compilatore che produce un file .XML che rispetta le DTD con le quali è stata definita la grammatica del linguaggio KDDML. Durante il parsing per verificare la consistenza del file .XML da sottoporre al sistema, si riconosce, in particolare, una *'kdd_query_table'* per il caricamento del file .arff originario, due *'kdd_query_PPtable'* per l'inizio della fase di preprocessing e per la marcatura duplicati:

```
<KDDML_OBJECT>
  <KDD_QUERY name="dataset">
    <ARFF_LOADER xml_dest="dataset.xml" file_name="cleansing.arff"/>
  </KDD_QUERY>
  <KDD_QUERY name="dataset_pp">
    <TABLE2PPTABLE xml_dest="dataset_pp.xml">
      <TABLE_LOADER file_name="dataset.xml"/>
    </TABLE2PPTABLE>
  </KDD_QUERY>
  <KDD_QUERY name="mark_duplicates_SSN">
    <PPMARK_DUPLICATES xml_dest="mark_duplicates_SSN.xml">
      <PPTABLE_LOADER file_name="dataset_pp.xml"/>
      <ATTRIBUTE_SEQ>
        <ATTR name="SSN"/>
      </ATTRIBUTE_SEQ>
      <WITH_CODE marca="not_unique"/>
    </PPMARK_DUPLICATES>
  </KDD_QUERY>
</KDDML_OBJECT>
```

L'esecuzione di questa query nel sistema KDDML produce i risultati HTML dei quali è illustrata in fig. 7.2 una porzione della pagina principale. Osservando le descrizioni della parte di estensione della tabella del dataset prodotto dalla query, appare evidente che i valori dell'attributo "SSN" sono "missing" al 99% (a meno di 4 istanze), pertanto sulle istanze con valore duplicato è stata apposta la marca di "not_unique". A fronte di tale risultato questi valori possono essere valutati come errori durante l'immissione dati e

The screenshot shows a web browser window displaying the results of a preprocessing operation. The browser title is "Output Table - Microsoft Internet Explorer". The address bar shows "mywebsearch". The main content is a table with two sections:

Attribute Info

Field Name	Type	Values: Cardinality (%)	Mean	Variance	Missed Values (%)
METADATI Customer	string	n/a	n/a	n/a	57000 (100%)
METADATI SSN	string	n/a	n/a	n/a	56996 (100%)
METADATI Birthdate	string	n/a	n/a	n/a	57000 (100%)
METADATI Age	string	n/a	n/a	n/a	57000 (100%)
METADATI Region	string	n/a	n/a	n/a	57000 (100%)
METADATI CredCardUser	string	n/a	n/a	n/a	57000 (100%)
METADATI Income	string	n/a	n/a	n/a	57000 (100%)
METADATI Purchase	string	n/a	n/a	n/a	57000 (100%)
METADATI AmtSpent	string	n/a	n/a	n/a	57000 (100%)

Preprocessing operations

Operation	Description
TABLE?PPTABLE	Start PreProcessing
PPMARK_DUPLICATES	key : ssn

XML Link: [./Data/KDDMLPPTable/mark_duplicates_SSN.xml](#)

Output Instances: [mark_duplicates_SSN.xml_data.html](#)
[metadata : mark_duplicates_SSN.xml_metadata.html](#)

Figura 7.2: Risultati di marcatura dei duplicati sull'attributo SSN.

corretti oppure, vista la poca numerosità dei valori duplicati, si può decidere di eliminare una delle istanze ripetute senza perdere troppi dati informativi (perché ad esempio non si conosce il vero valore da assegnare a "SSN"). In quest'ultimo caso la query MQL-DPL sottoposta al sistema è :

```
begin query

merge_duplicates_SSN = markduplicates in
mark_duplicates_SSN
on attribute SSN

end query
```

tradotta in linguaggio KDDML come:

```
<KDDML_OBJECT>
<KDD_QUERY name="merge_duplicates_SSN">
```


- quella inserita al momento della marcatura dei duplicati, secondo le specifiche della query;
- quella inserita a seguito dell'unificazione per notificare che quell'istanza è l'unica rappresentante di un insieme di istanze che sono state eliminate perché presentavano lo stesso valore su quell'attributo. Non essendo stato specificato il codice di marcatura (elemento opzionale del tipo "PPMERGE_DUPLICATES" di KDDML), viene comunque apposta la marca di default "NO MARK".

I risultati HTML prodotti sono parzialmente illustrati in fig.7.4 dove la history mostra quali sono le operazioni effettuate e le modalità specifiche per ciascuna operazione. In particolare l'ordine delle operazioni di preprocessing applicate è indicativo anche dei criteri di applicazione.

Per completezza, notiamo che in modo del tutto analogo poteva essere usata una query del tipo:

```
begin query
merge_duplicates_SSN = removerows
in mark_duplicates_SSN
where hasmark "not_unique with previous"
end query
```

per decidere di mantenere solo la prima occorrenza delle duplicate. Gli attributi "Birthdate" e "Age" presentano due aspetti indesiderati. Il formato della data di nascita esatto è mm/gg/aaaa mentre osservando i dati si ha che essi hanno formato mm/gg/aa. Si decide pertanto di riscrivere i valori della data nel formato corretto usando l'ipotesi che se le due cifre relative all'anno sono inferiori al comprese nell'intervallo "[00,04]" il millennio di riferimento è il 2000, mentre se le cifre dell'anno sono superiori a "04" allora il millennio è 1900. A livello di query language, questo corrisponde alla query seguente:

The screenshot shows a web browser window titled 'Output Table - Microsoft Internet Explorer'. The main content is a table with two sections. The first section, 'Attribute Info', has columns for Field Name, Type, Values: Cardinality (%), Mean, Variance, and Missed Values (%). The second section, 'Preprocessing operations', has columns for Operation and Description.

Attribute Info					
Field Name	Type	Values: Cardinality (%)	Mean	Variance	Missed Values (%)
METADATI Customer	string	n/a	n/a	n/a	56998 (100%)
METADATI SSN	string	n/a	n/a	n/a	56996 (100%)
METADATI Birthdate	string	n/a	n/a	n/a	56998 (100%)
METADATI Age	string	n/a	n/a	n/a	56998 (100%)
METADATI Region	string	n/a	n/a	n/a	56998 (100%)
METADATI CredCardUser	string	n/a	n/a	n/a	56998 (100%)
METADATI Income	string	n/a	n/a	n/a	56998 (100%)
METADATI Purchase	string	n/a	n/a	n/a	56998 (100%)
METADATI AmtSpent	string	n/a	n/a	n/a	56998 (100%)

Preprocessing operations	
Operation	Description
TABLE2PPTABLE	Start PreProcessing
PPMARK_DUPLICATES	key : ssn
PPMERGE_DUPLICATES	key : ssn

XML Link: [./Data/KDDMLPPTable/merge_duplicates_SSN.xml](#)
 Output Instances: [merge_duplicates_SSN.xml_data.html](#)
 metadata: [merge_duplicates_SSN.xml_metadata.html](#)

Figura 7.4: Prima pagina dei risultati prodotti dall'unificazione delle istanze con valori duplicati per l'attributo SSN.

```
begin query
  rewritten = rewrite in merge_duplicates_SSN
  attribute Birthdate
    using
      ( on er "(\\d\\d)/(\\d\\d)/(\\d\\d)"
        if (((("00" equal group 3) | ("01" equal group 3) ) |
            (("02" equal group 3) | ("04" equal group 3)))
          then (into "$1/$2/20$3")
            else (into "$1/$2/19$3"))
end query
```

Con questa query quello che succede è che i valori dell'attributo "Birthdate" viene sottoposto a riscrittura usando l'espressione regolare :

"(\\d\\d)/(\\d\\d)/(\\d\\d)"

in modo da riferirsi solo alle date nel formato mm/gg/aa. Al verificarsi del pattern matching la sostituzione è guidata mediante la verifica dell'uguaglianza tra il gruppo numero 3 ottenuto dal matching (che nello specifico è la parte della stringa che mappa le due cifre dell'anno) e le stringhe 00, 01, 02, 03, 04. Se tale uguaglianza è verificata il millennio di riferimento è quello del 2000 pertanto la nuova data sarà ottenuta dal valore originario come la stringa originaria che mappa il gruppo numero 1 del pattern (per le due cifre del mese), la stringa originaria che mappa il gruppo numero 2 del pattern (per le due cifre del giorno), una stringa composta da 20 concatenato alla stringa originaria che mappa il gruppo numero 3 del pattern. In modo del tutto analogo viene riscritta la data per quei valori che hanno le cifre relative all'anno che non verificano l'uguaglianza con nessuna stringa tra 00, 01, 02, 03, 04. In quest'ultimo caso, secondo le specifiche, le nuove sottostringhe che riferiscono l'anno saranno "19\$3" (la concatenazione di 19 e della stringa originaria che mappa il gruppo numero 3 del pattern). Al termine della traduzione in linguaggio KDDML di questa query per il trattamento dell'attributo "Birthdate" si ottiene il file .XML seguente:

```
<KDDML_OBJECT>
  <KDD_QUERY name="rewritten">
    <PPREWRITING xml_dest="rewritten.xml">
      <PPTABLE_LOADER file_name="merge_duplicates_SSN.xml"/>
      <ATTR name="Birthdate"/>
      <REWRITING_RULES>
        <RULE>
          <RULE_S>
            <REGULAR_EXPRESSION Pattern="(\d\d)/(\d\d)/(\d\d)"/>
            <STATEMENT>
              <IF_THEN_ELSE>
                <RW_COND>
                  <OR_RW>
                    <RW_COND>
                      <OR_RW>
```

```
<RW_COND>
  <COMPARE_STRING_RW>
    <RW_TERM>
      <CONSTANT valore="00"/>
    </RW_TERM>
    <RW_TERM>
      <RAGGR group_variable="3"/>
    </RW_TERM>
  </COMPARE_STRING_RW>
</RW_COND>
<RW_COND>
  <COMPARE_STRING_RW>
    <RW_TERM>
      <CONSTANT valore="01"/>
    </RW_TERM>
    <RW_TERM>
      <RAGGR group_variable="3"/>
    </RW_TERM>
  </COMPARE_STRING_RW>
</RW_COND>
</OR_RW>
</RW_COND>
<RW_COND>
  <OR_RW>
    <RW_COND>
      <COMPARE_STRING_RW>
        <RW_TERM>
          <CONSTANT valore="02"/>
        </RW_TERM>
        <RW_TERM>
          <RAGGR group_variable="3"/>
        </RW_TERM>
      </COMPARE_STRING_RW>
    </RW_COND>
    <RW_COND>
      <COMPARE_STRING_RW>
        <RW_TERM>
          <CONSTANT valore="04"/>
        </RW_TERM>
        <RW_TERM>
          <RAGGR group_variable="3"/>
        </RW_TERM>
      </COMPARE_STRING_RW>
    </RW_COND>
  </OR_RW>
</RW_COND>
```

```

        </RW_TERM>
    </COMPARE_STRING_RW>
</RW_COND>
    </OR_RW>
</RW_COND>
</OR_RW>
</RW_COND>
<STATEMENT>
    <SUBSTITUTION substitution_string="$1/$2/20$3"/>
</STATEMENT>
<STATEMENT>
    <SUBSTITUTION substitution_string="$1/$2/19$3"/>
</STATEMENT>
</IF_THEN_ELSE>
</STATEMENT>
</RULE_S>
</RULE>
</REWRITING_RULES>
</PPREWRITING>
</KDD_QUERY>
</KDDML_OBJECT>

```

Il file .XML prodotto a seguito della compilazione è eseguibile dal sistema KDDML che riconosce un tipo “*kdd_query_PPtable*” in particolare un “PPREWRITING” (definito in 5.1.1) e lo esegue tramite la chiamata della classe java con lo stesso nome. Al termine dell’elaborazione della query il sistema restituisce i risultati in formato HTML con i dati modificati rispettando il formato corretto mm/gg/aaaa come in fig.7.5.

Per l’attributo “Age” notiamo che ci sono dei valori che hanno valori palesemente scorretti perché negativi. Tali valori sono probabilmente il frutto di un errore dovuto al cambio di millennio che ha causato esiti errati per l’applicazione di una formula di calcolo del tipo “(anno corrente - anno di nascita)” condizionata da vincoli del tipo se le due cifre relative all’anno sono comprese tra 30 e 99 allora la persona in esame è nata nel ventesimo secolo altrimenti è nata nel ventunesimo. Quello che è successo è che se ad esempio

Output Data								
Customer	SSN	Birthdate	Age	Region	CredCardUser	Income	Purchase	AmtSpent
1	539-84-9599	10/26/1944	59	East	0	62900	4	2080
2	444-05-4079	01/01/1932	67	West	1	23900	0	0
3	418-18-5649	08/17/1973	25	East	1	48700	8	3990
4	065-63-3311	08/02/1947	51	West	1	137600	2	920
5	059-58-9566	10/03/1948	50	East	0	101400	2	1000
6	443-13-8685	03/24/1960	39	East	0	139700	1	330
7	638-89-7231	12/02/1943	55	South	1	50900	3	1400
8	202-94-6453	11/08/1974	24	South	1	50500	0	0
9	266-29-0308	09/28/1967	31	North	0	151400	2	910
10	943-85-8301	07/05/1965	33	W	0	88300	2	1080
11	047-07-5332	11/13/1964	34	North	0	120300	3	1390
12	503-14-3400	03/21/1934	65	East	0	25800	1	560
13	696-97-5817	12/29/1960	38	West	0	65500	1	410
14	614-59-6703	08/01/1972	26	South	1	54400	9999	9999
15	704-98-5642	05/01/1952	46	North	0	54500	7	3570
16	588-92-9590	08/29/1939	59	South	0	142500	2	1070
17	267-58-0440	04/05/1957	42	East	1	40500	2	950
18	833-81-4506	11/01/1967	31	West	1	61900	2	930
19	645-20-8840	03/02/1958	41	West	1	106300	2	1030

Figura 7.5: Dati modificati della riscrittura della data di nascita nel formato corretto.

si ha data di nascita 02/03/29 l'età risultante dal calcolo è $04 - 29 = -25$. Come primo passo di analisi si applica una marca su tutte le istanze che hanno valori negativi per l'età sottoponendo al sistema la query:

```
begin query

marked = mark in
  rewritten
  attribute Age
  if (Age<0)
  withcode "negative"

end query
```

tradotta come tipo “PPMARKING_QUERY” dei KDDML_OBJECT validi del linguaggio KDDML come segue:

```

<KDDML_OBJECT>
  <KDD_QUERY name="marked">
    <PPMARKING_QUERY xml_dest="marked.xml">
      <PPTABLE_LOADER file_name="rewrited.xml"/>
      <ATTR name="Age"/>
      <MARK_COND>
        <REL_MARK type="less">
          <MARK_TERM>
            <ATTR name="Age"/>
          </MARK_TERM>
          <MARK_TERM>
            <CONSTANT valore="0"/>
          </MARK_TERM>
        </REL_MARK>
      </MARK_COND>
      <WITH_CODE marca="negative"/>
    </PPMARKING_QUERY>
  </KDD_QUERY>
</KDDML_OBJECT>

```

L'esecuzione di questa query prende la tabella estesa modellata dal "marked.xml" e produce una tabella estesa che è identica a meno dei metadati relativi all'attributo "Age" in corrispondenza dei valori negativi. Per questi, infatti, il valore contenuto è "negative" così come specificato. Dalla pagina principale dei risultati HTML prodotti illustrata in fig. 7.6

è evidente che i valori marcati sono 12 visto che i restanti sono sempre settati a "missing". Osservando le date di nascita si osserva che gli anni hanno tutti valori stranamente bassi. Questa considerazione porta a pensare che la formula di calcolo usata per l'attributo "Age" fa riferimento al secolo di appartenenza settato con riferimenti al ventesimo secolo. La correzione radicale da applicare al dataset è sicuramente l'inserimento di un nuovo attributo costruito in base all'anno in corso e alla data di nascita. Si può decidere di sottoporre al sistema una query del tipo seguente:

```
begin query
```

The screenshot shows a web browser window titled 'Output Table - Microsoft Internet Explorer'. The browser's address bar contains 'mywebsearch'. The main content area displays two tables. The first table, 'Attribute Info', has six columns: 'Field Name', 'Type', 'Values: Cardinality (%)', 'Mean', 'Variance', and 'Missed Values (%)'. It lists ten metadata fields, all of which are strings with 'n/a' for mean and variance, and 100% missed values. The second table, 'Preprocessing operations', has two columns: 'Operation' and 'Description'. It lists five operations: 'TABLE:PPTABLE', 'PPMARK_DUPPLICATES', 'PPMERGE_DUPPLICATES', 'PPREWRITING', and 'PPMARKING_QUERY', each with a brief description of the operation performed.

Attribute Info					
Field Name	Type	Values: Cardinality (%)	Mean	Variance	Missed Values (%)
METADATI Customer	string	n/a	n/a	n/a	1500 (100%)
METADATI SSN	string	n/a	n/a	n/a	1498 (100%)
METADATI Birthdate	string	n/a	n/a	n/a	1500 (100%)
METADATI Age	string	n/a	n/a	n/a	1488 (99%)
METADATI Region	string	n/a	n/a	n/a	1500 (100%)
METADATI CredCardUser	string	n/a	n/a	n/a	1500 (100%)
METADATI Income	string	n/a	n/a	n/a	1500 (100%)
METADATI Purchase	string	n/a	n/a	n/a	1500 (100%)
METADATI AmtSpent	string	n/a	n/a	n/a	1500 (100%)

Preprocessing operations	
Operation	Description
TABLE:PPTABLE	Start PreProcessing
PPMARK_DUPPLICATES	key : ssn
PPMERGE_DUPPLICATES	key : ssn
PPREWRITING	RULE_S, on attribute : Birthdate
PPMARKING_QUERY	Attribute : Age

Figura 7.6: Porzione della pagina principale dei risultati prodotti dall'operazione di marcatura dei valori negativi dell'attributo "Age".

```

newAge = newattribute in
  (newattribute in (rewrite in (newattribute in marked
    withdescription
      name Born_year_string
      type string
      createdas copyof Birthdate)
    attribute Born_year_string using ( on er "(\\d\\d)/(\\d\\d)/(\\d+)" into "$3"
      )
    withdescription
      name Born_year
      type real
      createdas copyof Born_year_string
    )
  withdescription
    name Age_on_2004

```

```
        type real
        createdas -(2004,Born_year)
end query
```

Con questa query annidata gli effetti sono in sequenza i seguenti:

1. creo un nuovo attributo “Born_year_string” di tipo stringa che ha tutti i valori come copia dell’attributo “Birthdate”;
2. quindi si procede alla riscrittura di questo nuovo attributo temporaneo sostituendo i valori con la sola sottostringa relativa all’anno (gruppo numero 3 del pattern usato);
3. a questo punto si crea un nuovo attributo “Age_on_2004” come copia di “Born_year_string”, già adeguatamente riscritto per riferire solo la stringa che rappresenta l’anno di nascita, decidendo però che il tipo di “Age_on_2004” sia “real”. Quello che si vuole è che ogni valore sia il risultato dell’interpretazione della stringa come valore numerico;
4. finalmente ora abbiamo la possibilità di creare il nuovo attributo per riferire le età con riferimento all’anno 2004, usando ancora il costrutto “newattribute” per creare “Age_on_2004” con valori calcolati come il risultato dell’espressione numerica “2004-Born_year” su ciascuna istanza.

Anche in questo caso esiste più di un approccio allo stesso problema della creazione di un nuovo attributo stringa contenente solo la sottostringa relativa all’anno di nascita. Si poteva, infatti, decidere di usare il costrutto “SPLIT” sull’attributo “Birthdate” sulla base dell’espressione regolare “(\d\d\d\d)”. Con questa soluzione si sarebbe ottenuto un nuovo valore per “Birthdate” contenente solo le cifre relative a mese e giorno di nascita

nel formato “mm/gg/”, e la costruzione di un nuovo attributo contenete la sottostringa dei valori di “Birthdate” a partire dalla parte che mappa il pattern fino al termine (in questo caso ogni valore del nuovo attributo sarebbe stato costituito dalla sola sottostringa che mappa il pattern). In linguaggio KDDML la query corrispondente che deve essere interpretata è:

```

<KDDML_OBJECT>
  <KDD_QUERY name="newAge">
    <PPNEWATTRIBUTE_QUERY xml_dest="newAge.xml">
      <PPNEWATTRIBUTE_QUERY>
        <PPREWRITING>
          <PPNEWATTRIBUTE_QUERY>
            <PPTABLE_LOADER file_name="marked.xml"/>
            <NEWATTR attribute_name="Born_year_string" attribute_type="string"/>
            <NEWATT_EXP>
              <TERMINE>
                <ATTR name="Birthdate"/>
              </TERMINE>
            </NEWATT_EXP>
          </PPNEWATTRIBUTE_QUERY>
          <ATTR name="Born_year_string"/>
          <REWRITING_RULES>
            <RULE>
              <RULE_S>
                <REGULAR_EXPRESSION Pattern="(\\d\\d)/(\\d\\d)/(\\d+)" />
                <STATEMENT>
                  <SUBSTITUTION substitution_string="$3" />
                </STATEMENT>
              </RULE_S>
            </RULE>
          </REWRITING_RULES>
        </PPREWRITING>
        <NEWATTR attribute_name="Born_year" attribute_type="real"/>
        <NEWATT_EXP>
          <TERMINE>
            <ATTR name="Born_year_string"/>
          </TERMINE>
        </NEWATT_EXP>
      </PPNEWATTRIBUTE_QUERY>
      <NEWATTR attribute_name="Age_on_2004" attribute_type="real"/>
    </PPNEWATTRIBUTE_QUERY>
  </KDD_QUERY>
</KDDML_OBJECT>

```

```

<NEWATT_EXP>
  <TERMINE>
    <EXPR>
      <TERM_OPERATION>
        <NUMERIC_OPERATION op_type="subtract"/>
      </TERM_OPERATION>
      <TERM_SEQ>
        <TERMINE>
          <CONSTANT valore="2004"/>
        </TERMINE>
        <TERMINE>
          <ATTR name="Born_year"/>
        </TERMINE>
      </TERM_SEQ>
    </EXPR>
  </TERMINE>
</NEWATT_EXP>
</PPNEWATTRIBUTE_QUERY>
</KDD_QUERY>
</KDDML_OBJECT>

```

L'esecuzione di tale programma nel sistema KDDML produce una tabella estesa con 3 nuovi attributi come illustrato in fig.7.7. Di tali attributi quello di interesse è solo "Age_on_2004", pertanto si può decidere di buttare via gli altri 2 ("Born_year" e "Born_year_string") creati solo per utilità nel calcolo delle età, oltre al vecchio attributo "Age" reputato inconsistente.

Per attuare la trasformazione di schema eliminando gli attributi inutili viene inserita la query:

```

begin query

  removed_attribute = remove from newAge
                    attribute Born_year,Born_year_string,Age

end query

```

passata al sistema dal compilatore come un tipo "PPREMOVE_QUERY" del linguaggio KDDML:

Total Number of Attributes
12

Total Number of Instances
1498

Attribute Info

Field Name	Type	Values: Cardinality (%)	Mean	Variance	Missed Values (%)
Customer	numeric	n/a	750.7797062750334	187809.030930575	0 (0%)
SSN	string	n/a	n/a	n/a	0 (0%)
Birthdate	string	n/a	n/a	n/a	0 (0%)
Age	numeric	n/a	45.77970627503338	231.42906016750888	0 (0%)
Region	string	n/a	n/a	n/a	0 (0%)
CredCardUser	nominal	0.900 (60%) 1.598 (40%)	n/a	n/a	0 (0%)
Income	numeric	n/a	89706.3531375167	1.7068513359640558E9	0 (0%)
Purchase	numeric	n/a	123.06008010680908	1187021.7024667938	0 (0%)
AmntSpent	numeric	n/a	1560.2349799732976	1550933.8645872073	0 (0%)
Born_year_string	string	n/a	n/a	n/a	0 (0%)
Born_year	numeric	n/a	1951.704272363151	190.27387619018555	0 (0%)
Age_on_2004	numeric	n/a	52.29572763684913	190.27387619029784	0 (0%)

Figura 7.7: Porzione della pagina principale dei risultati prodotti dalla query per il trattamento dell' attributo "Age".

```

<KDDML_OBJECT>
  <KDD_QUERY name="removed_attribute">
    <PPREMOVE_QUERY xml_dest="removed_attribute.xml">
      <PPTABLE_LOADER file_name="newAge.xml"/>
      <ATTRIBUTE_SEQ>
        <ATTR name="Born_year"/>
        <ATTR name="Born_year_string"/>
        <ATTR name="Age"/>
      </ATTRIBUTE_SEQ>
    </PPREMOVE_QUERY>
  </KDD_QUERY>
</KDDML_OBJECT>

```

L'ultimo trattamento da effettuare è dedicato all'attributo "Region" i cui valori devono intuitivamente essere tali da rientrare in un insieme finito di possibilità ma che compaiono con abbreviazioni tali da renderli apparente-

mente più variegati della realtà. Il dataset viene sottoposto a riscrittura di tutti i valori di "Region" che mappano un pattern (sufficientemente generico ma identificativo) con una regione fra: North, South, West, East. I pattern scelti per applicare le riscritture sono rispettivamente: "(N|n).*", "(S|s).*", "(W|w).*", "(E|e).*". Resi uniformi i valori si può definitivamente convertire il tipo dell'attributo da stringa a enumerato per vincolare anche le successive immissioni di dati. La query sottomessa pertanto è la seguente:

```
begin query

rewritten_region =
remove from (newattribute in (
  rewrite in
    removed_attribute
  attribute Region using
    ( on er "(N|n).*" into "north")
    ( on er "(S|s).*" into "south")
    ( on er "(W|w).*" into "west")
    ( on er "(E|e).*" into "east"))

withdescription
name Region_enumerated
  type enumerated
  createdas copyof Region)
  attribute Region
end query
```

A seguito della compilazione il sistema procede con l'esecuzione di:

```
<KDDML_OBJECT>
<KDD_QUERY name="rewritten_region">
  <PPREMOVE_QUERY xml_dest="rewritten_region.xml">
    <PPNEWATTRIBUTE_QUERY>
      <PPREWRITING>
        <PPTABLE_LOADER file_name="removed_attribute.xml"/>
        <ATTR name="Region"/>
      <REWRITING_RULES>
```

```
<RULE>
  <RULE_S>
    <REGULAR_EXPRESSION Pattern="(N|n).*" />
    <STATEMENT>
      <SUBSTITUTION substitution_string="north" />
    </STATEMENT>
  </RULE_S>
</RULE>
<RULE>
  <RULE_S>
    <REGULAR_EXPRESSION Pattern="(S|s).*" />
    <STATEMENT>
      <SUBSTITUTION substitution_string="south" />
    </STATEMENT>
  </RULE_S>
</RULE>
<RULE>
  <RULE_S>
    <REGULAR_EXPRESSION Pattern="(W|w).*" />
    <STATEMENT>
      <SUBSTITUTION substitution_string="west" />
    </STATEMENT>
  </RULE_S>
</RULE>
<RULE>
  <RULE_S>
    <REGULAR_EXPRESSION Pattern="(E|e).*" />
    <STATEMENT>
      <SUBSTITUTION substitution_string="east" />
    </STATEMENT>
  </RULE_S>
</RULE>
</REWRITING_RULES>
</PPREWRITING>
<NEWATTR attribute_name="Region_enumerated" attribute_type="enumerated" />
<NEWATT_EXP>
  <TERMINE>
    <ATTR name="Region" />
  </TERMINE>
</NEWATT_EXP>
</PPNEWATTRIBUTE_QUERY>
```

```
<ATTRIBUTE_SEQ>
  <ATTR name="Region"/>
</ATTRIBUTE_SEQ>
</PPREMOVE_QUERY>
</KDD_QUERY>
</KDDML_OBJECT>
```

L'interprete procede con la chiamata alla classe "PPNEWATTRIBUTE_QUERY" sulla tabella estesa risultante dall'esecuzione del costrutto "PPREWRITING". Durante la riscrittura tutti i valori assunti dall'attributo "Region" sono uniformati a {south,north,west,east} verificando l'applicabilità di una delle regole su ciascuna istanza. Al termine viene costruito un nuovo attributo di tipo enumerato come copia dell'attributo "Region" adeguatamente riscritto. La conversione da stringa a nominale è risolta nella classe "PPNEWATTRIBUTE.java" sfruttando la libreria WEKA e in particolare istanziando la "*weka.filters.unsupervised.attribute.StringToNominal*". A questo punto non rimane altro che eliminare il vecchio attributo "Region" per portare a termine l'operazione e ottenere esattamente il risultato desiderato (vedi fig.7.8).

Il dataset così ottenuto rispetta le specifiche desiderate per ottenere un insieme di dati consolidato ottimale per proseguire con le successive fasi di Knowledge Discovery senza incorrere in inconvenienti dovuti a false interpretazioni di dati errati. Come appare evidente dalle modalità utilizzate nell'esempio, il preprocessing per sua natura è "*user guided*" nel senso che le decisioni su quale tipo di operazioni applicare e con quali parametri vengono prese tenendo conto del tipo di conoscenza che si vuole arrivare ad estrarre e ponderando le scelte sulla base della conoscenza del dominio.

KDDML Table Visualization

Total Number of Attributes: 9

Total Number of Instances: 1498

Attribute Info					
Field Name	Type	Values: Cardinality (%)	Mean	Variance	Missed Values (%)
Customer	numeric	n/a	750.7797062750334	187809.030930575	0 (0%)
SSN	string	n/a	n/a	n/a	0 (0%)
Birthdate	string	n/a	n/a	n/a	0 (0%)
CredCardUser	nominal	0: 900 (60%) 1: 598 (40%)	n/a	n/a	0 (0%)
Income	numeric	n/a	89706.3531375167	1.7068513359640558E9	0 (0%)
Purchase	numeric	n/a	123.06008010680908	1187021.7024667938	0 (0%)
AmtSpent	numeric	n/a	1560.2349799732976	1550933.8645872073	0 (0%)
Age_on_2004	numeric	n/a	52.29572763684913	190.27387619029784	0 (0%)
Region_enumerated	nominal	east: 384 (26%) west: 362 (24%) south: 407 (27%) north: 345 (23%)	n/a	n/a	0 (0%)

Figura 7.8: Porzione della pagina principale dei risultati prodotti dalla query per il trattamento dell' attributo "Region".

Capitolo 8

Conclusioni

Il contributo principale della tesi è la realizzazione dell'estensione del sistema KDDML per la risoluzione di problematiche di “qualità dei dati”. Con questo lavoro l'ambiente KDDML [23] è stato integrato in modo da consentire applicazioni per affrontare la fase di preprocessing rendendo l'ambiente completo rispetto all'intero processo KDD. Il lavoro più oneroso è stato sicuramente quello dedicato all'estensione della macchina astratta del sistema sia per l'introduzione del nuovo tipo di struttura dati finalizzato alla manipolazione delle tabelle estese, sia per l'implementazione degli operatori per la risoluzione delle trasformazioni di dati dettate dalle necessità in fase di preprocessing. Tutto il lavoro è stato eseguito cercando il più possibile di rispettare la semantica specificata sul linguaggio DPL [9] e modificando la sintassi solo allo scopo di ottenere un linguaggio per l'introduzione delle query che fosse il più uniforme possibile. Quando necessario, sono stati introdotti operatori aggiuntivi anche a livello più alto modificando la sintassi del query language per fornire funzionalità ulteriori e non previste, allo scopo di affrontare problemi, quale ad esempio il trattamento dei dati mancanti, che spesso si presentano su dati reali. Le difficoltà maggiori incontrate

riguardano soprattutto la necessità di apprendere nozioni su diverse tecnologie software utilizzate nel sistema. Innanzitutto per la rappresentazione dati semi-strutturati l'uso del formato XML e delle DTD oltre che delle interfacce fornite dal gruppo W3C sia per modificare e rappresentare i documenti XML come DOM (Document Object Model), sia per la visualizzazione sul browser in formato HTML mediante trasformazioni dei dati associate a fogli di stile (XSL). Per la compilazione, necessario è stato l'apprendimento delle specifiche del framework ANTLR già usato nel sistema per risolvere ogni fase di analisi lessicale, sintattica, dei tipi integrando le definizioni sintattiche, mantenendo valide le categorie già definite e cercando di rendere non fuorviante il lessico attenendosi all'impronta data al query language disponibile per l'utente. Le strutture dati utilizzate per la rappresentazione dei dati in forma tabellare sfruttano classi della libreria fornita da WEKA. Pertanto per fare un uso accurato degli oggetti disponibili si è resa necessaria un'attenta analisi dei sorgenti e degli effetti laterali di ciascun metodo usato. Il sistema KDDML è da qualche mese in fase di riprogettazione nella sua interezza (da qui la rinominazione del sistema KDDML-MQL [23] semplicemente in KDDML), pertanto la progettazione delle funzionalità di preprocessing per questo lavoro di tesi è avvenuta in modo parallelo alla ristrutturazione della gerarchia delle classi per la nuova versione alla quale essa si attiene. Questo interagire tra i lavori ha portato in un primo momento allo svolgimento di lavoro in riferimento a un sistema che da lì a poco sarebbe stato soppiantato da una nuova versione, con conseguente lavoro non produttivo se non per familiarizzare con gli strumenti. D'altra parte è stato possibile collaborare alla progettazione della nuova versione per permetterne la realizzazione con una visione completa delle necessità da soddisfare. Partendo dal lavoro concluso con questa tesi rimane non risolta l'implementazione di un'interfaccia

grafica completa che permetta la sottomissione di richieste mediante query language al sistema che si occuperà della risoluzione e della visualizzazione dei risultati. Miglioramenti futuri al sistema sono sicuramente inerenti all'ottimizzazione della traduzione delle query da linguaggio MQL-DPL a linguaggio KDDML, e alla realizzazione di strumenti di visualizzazione dei risultati sulla conoscenza estratta.

Ringraziamenti

Questo spazio è dedicato a tutte le persone che sono state importanti per me fino ad ora ciascuna a suo modo e ciascuna relativamente al contesto nel quale ho avuto l'opportunità di incontrarla. Il primo pensiero va sicuramente ai miei genitori che hanno sempre creduto in me e che, con i loro insegnamenti e il loro affetto, mi hanno reso la persona che oggi sono. Ringrazio mio fratello Orlando "rispettosa presenza" della mia vita, per essere sempre vicino a me come la persona su cui posso contare in qualunque momento e per qualunque tipo di necessità senza mai sentirsi nella posizione di invadere i miei spazi più di quanto opportuno. Un bacio a mia nonna, una persona speciale che mi ha aiutato a crescere accompagnandomi sin dalla mia infanzia con quella strana mescolanza di dolcezza e fermezza che la contraddistinguono. Un dolce pensiero al mio amore Alessio con cui ho avuto la fortuna di condividere anche gran parte della mia vita universitaria. L'angoscia e lo stress dello studio nei momenti più duri sono stati meno pesanti con la certezza di avere vicino una persona consapevole delle difficoltà, al punto da essere sempre anche in grado di amplificare la felicità dei traguardi raggiunti insieme. Un abbraccio va ai genitori e ai nonni di Alessio che mi sono stati vicino considerandomi davvero come una loro figlia e che hanno reso meno difficile affrontare la distanza dalla mia famiglia. Ringrazio il Prof. Franco Turini che sempre si è dimostrato una persona professionale e intelligente non

solo dal punto di vista della sua evidente preparazione ma anche dal punto di vista umano, capace di comportarsi sempre in modo assolutamente corretto accompagnando il tutto con una rara gentilezza. Un grazie va sicuramente a Andrea Romei che durante tutto il lavoro di tesi si è mostrato sempre disponibile ad ascoltarmi e che spesso mi ha aiutato ad avere una visione più chiara di tanti aspetti, e anche al prof. Salvatore Ruggieri per i suoi consigli. Abbraccio la mia amica del cuore Roberta, ormai come una sorella per me, che sin dal mio primo giorno a Pisa ha percorso con me questo tragitto. Ringrazio tutti i miei amici conosciuti all'Università, Andrea per essere sempre stato un piacevole compagno di avventure rendendo sempre meno tristi i giorni con la sua unica ironia, e ancora Omar, Simone, Francesco, Massimo, oltre ai miei compagni di viaggio nel lavoro di tesi che hanno resistito ai miei repentini cambi d'umore Barbara, Francesco, Pasquale, Marlis, Luana. Ringrazio anche tutte le persone conosciute in questo periodo universitario che mi hanno fatto compagnia e che mi hanno lasciato qualcosa nel cuore. Grazie a tutte loro ho raggiunto questo traguardo e sono felice perché oltre alle conoscenze acquisite con gli studi, oggi mi sento una persona migliore.

Bibliografia

- [1] J.Han, M.Kamber. *Data Mining, Concepts and Techniques*, Morgan Kaufmann Publishers Agosto 2000.
- [2] Fayyad, Piatetsky-Shapiro e Smyth. *Advances in Knowledge Discovery and Data Mining*, 1996.
- [3] Silberschatz, Stonebraker, Ullmann. *Database research: achievements and opportunities into the 21st century*, 1995.
- [4] H. B. Newcombe. *Handbook of record linkage: methods for health and statistical studies, administration and business*, 1998.
- [5] Quinlan. *Induction of decision tree machine learning*, 1986.
- [6] Quinlan. *C 4.5, Programs for machine learning*, 1988.
- [7] Baranauskas, Monard, Batista. *A computational extracting rules from data bases*.
- [8] E. Rahm and H. Hai Do. *Data Cleaning: Problems and Current Approaches*. Bulletin of the Technical Committee on Data Engineering December 2000 Vol. 23 No. 4 IEEE Computer Society.

-
- [9] Valentini Marlis *DPL: Un formalismo algebrico per la specifica della preparazione dei dati all'estrazione di conoscenza*. (Tesi di Laurea). Relatore prof. Franco Turini, 2003.
- [10] Sudipto Guha, Rajeev Rastogi, Kyuseok Shim. *ROCK: a robust clustering algorithm for categorical attributes*, 2000.
- [11] Alvaro E. Monge. *Matching Algorithms within a Duplicate Detection System*. Bulletin of the Technical Committee on Data Engineering December 2000 Vol. 23 No. 4 IEEE Computer Society.
- [12] Rohit Ananthakrishna, Surajit Chaudhuri, Venkatesh Ganti. *Eliminating Fuzzy Duplicates in Data Warehouses*, Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002.
- [13] J. MacQueen. *Some methods for classification, and analysis of multivariate observations*, 1967.
- [14] P. Alcamo, F. Domenichini *Un ambiente basato su XML per l'estrazione di conoscenza* (Tesi di Laurea). Relatore prof. Franco Turini, 2000.
- [15] M.A. Hernandez, S.J. Stolfo. *Real-World Data is dirty: Data Cleansing and the Merge/Purge problem*, 2000.
- [16] Surajit Chaudhuri, Kris Ganjam, Venkatesh Ganti, Rajeev Motwani. *Robust and Efficient Fuzzy Match for Online Data Cleaning*, SIGMOD 2003.
- [17] A. K. Jain, M.N. Murty, P. J. Flynn, *Data clustering: A survey*, ACM 1999.
- [18] M.A. Hernandez, S.J. Stolfo. *The Merge/Purge Problem for Large Databases*, 1995

- [19] Paul D. Allison *Missing Data*, 2002
- [20] Efron. *Estimating the error rate of a prediction rule: some improvements on cross-validation*. Journal of the American Statistical Association, 78, 316-331. 1983.
- [21] Ian Witten Witten , Eibe Frank. *Data Mining*, Book 2000.
- [22] J. Souza, S. Matwin, N.Japkowicz .*Evaluating Data Mining Models: A Pattern Language*,2001.
- [23] Romei Andrea *Implementazione di un query language per Knowledge Discovery* (Tesi di Laurea). Relatore prof. Franco Turini, 2002.
- [24] Miriam Baglioni *MQL: Una proposta di linguaggio per DM*(Tesi di Laurea). Relatore prof. Franco Turini, 2001.
- [25] eXtensible Markup Language specification: <http://www.w3c.org/XML>
- [26] Document Object Model specification: <http://www.w3c.org/DOM>
- [27] eXtensible StyleSheet Language specification:
<http://www.w3c.org/Style/XSL>
- [28] IBM's Alphaworks XML4J: <http://www.alphaworks.ibm.com/formula/xml>
- [29] Imielinsky T. and Mannila H. *A database Perspective of Knowledge Discovery* , 1996. In Communication of the ACM, 39(11): 58-64.
- [30] Predictive Model Markup Language version 2.0 <http://www.dmg.org>
- [31] J. Hand e Y.Fu. *Discovery of multi-level association rules from large databases* ,1995.

- [32] Waikato Environment for Knowledge Analysis:
<http://www.cs.waikato.ac.nz/ml/weka>.