

Indice

INTRODUZIONE.....	VII
CAPITOLO 1 DATA MINING & KNOWLEDGE DISCOVERY IN DATABASES.....	1
1.1 INTRODUZIONE.....	1
1.2 LA CONOSCENZA.....	2
1.3 KNOWLEDGE DISCOVERY IN DATABASES	6
1.3.1 FASI DEL KDD	8
1.3.2 CICLO VIRTUOSO	12
1.3.3 ARCHITETTURA TIPICA DI UN SISTEMA KDD.....	15
1.3.4 SETTORI APPLICATIVI DEL KDD	17
1.4 DATA MINING	19
1.4.1 REGOLE DI ASSOCIAZIONE.....	21
1.4.2 CLASSIFICAZIONE.....	26
1.4.3 CLUSTERING.....	40
1.4.4 ASPETTI TEMPORALI NEL DATA MINING	44
1.5 IL FUTURO DEL KDD	45
CAPITOLO 2 KDDML.....	49
2.1 INTRODUZIONE.....	49
2.2 XML.....	50
2.2.1 DOCUMENTI XML.....	51
2.2.2 ELABORAZIONE DI DOCUMENTI XML.....	53
2.3 PMML.....	54
2.4 KDDML.....	56
2.4.1 IL LINGUAGGIO	57
2.4.2 L'ARCHITETTURA	65
2.4.3 ESEMPIO DI APPLICAZIONE DI KDDML	71
2.4.4 LA GUI ATTUALE.....	75
CAPITOLO 3 SISTEMI DI KD E LORO ASPETTI VISUALI	79
3.1 INTRODUZIONE.....	79
3.1.1 PARAMETRI GENERALI DI VALUTAZIONE	81
3.2 WEKA.....	83
3.2.1 WEKA GUI	85
3.2.2 ESEMPIO DI UTILIZZO DI WEKA	92
3.3 YALE.....	98

3.3.1	YALE GUI.....	99
3.3.2	ESEMPIO DI UTILIZZO DI YALE	105
3.4	ORANGE.....	109
3.4.1	ORANGE CANVAS	111
3.4.2	ESEMPIO DI UTILIZZO DI ORANGE CANVAS	116
3.5	GHOSTMINER	120
3.5.1	GHOSTMINER DEVELOPER	123
3.5.2	ESEMPIO DI UTILIZZO DI GHOSTMINER DEVELOPER	127
3.6	TANAGRA.....	132
3.6.1	TANAGRA GUI.....	134
3.6.2	ESEMPIO DI UTILIZZO DI TANAGRA.....	139
CAPITOLO 4 COMPARAZIONE.....		145
4.1	INTRODUZIONE.....	145
4.2	MODALITÀ DI ACCESSO A DATI E MODELLI	146
4.3	COSTRUZIONE E RAPPRESENTAZIONE DI UNA QUERY.....	149
4.3.1	RAPPRESENTAZIONE DEL FLUSSO DI DATI: ALBERO vs GRAFO.....	150
4.3.2	PRESENZA DI UN WIZARD	152
4.3.3	IMMISSIONE DEI PARAMETRI	153
4.3.4	LIMITAZIONI E POTENZIALITÀ DEI VARI TOOL	154
4.3.5	RAPPRESENTAZIONE DELLA QUERY: ALCUNI ESEMPI.....	155
4.3.6	RAPPRESENTAZIONE DI COSTRUTTI COMPLESSI.....	158
4.3.7	ESPRESSIVITÀ ED IMMEDIATEZZA DELLE RAPPRESENTAZIONI	162
4.4	ESECUZIONE E VALIDAZIONE	163
4.5	VISUALIZZAZIONE DEI DATI E DEI RISULTATI.....	166
4.6	ESTENDIBILITÀ DEI SISTEMI	168
4.7	SUPPORTO ALL'UTENTE.....	169
4.8	REQUISITI	170
4.8.1	ACCESSO AI DATI.....	171
4.8.2	LA METAFORA GRAFICA.....	171
4.8.3	SEMPLICE E COMPLETA ESPANSIONE DEL SISTEMA	172
4.8.4	ESECUZIONE E META-ESECUZIONE	173
4.8.5	IMMISSIONE DEI PARAMETRI	176
4.8.6	VISUALIZZAZIONE INPUT/OUTPUT	177
4.8.7	MODELLI ESTRATTI E UTILIZZO DI PMML	178
4.8.8	POSSIBILITÀ DI SALVATAGGIO DEL KNOWLEDGE FLOW.....	178
4.8.9	DOCUMENTAZIONE FORNITA.....	179
4.9	CONSIDERAZIONI FINALI.....	179
CAPITOLO 5 STUDIO DEI REQUISITI DEL LINGUAGGIO VISUALE PER KDDML		181
5.1	INTRODUZIONE.....	181

5.2	KDDML & ACCESSO AI DATI.....	182
5.3	KDDML & METAFORA GRAFICA	185
5.4	KDDML & ESPANDIBILITÀ	193
5.5	KDDML & META-ESECUZIONE	199
5.6	KDDML & IMMISSIONE PARAMETRI.....	203
5.7	KDDML & VISUALIZZAZIONE	205
5.8	KDDML & MODELLI	207
5.9	KDDML & SALVATAGGI.....	208
5.10	KDDML & DOCUMENTAZIONE FORNITA	210
	CONCLUSIONI.....	211
	BIBLIOGRAFIA	217

Indice delle figure

Figura 1.1 Conoscenza e Metaconoscenza.....	3
Figura 1.2 Fasi del processo KDD.	8
Figura 1.3 Ciclo virtuoso.....	13
Figura 1.4 Architettura di un sistema per KDD.	16
Figura 1.5 Fasi tipiche della classificazione.....	28
Figura 1.6 Albero di decisione estratto dal training set di Tabella 1.3.	30
Figura 1.7 Algoritmo <i>greedy</i> per la costruzione di un albero di decisione.	31
Figura 1.8 Esempio di matrice di confusione.....	34
Figura 2.1 Possibile rappresentazione in XML delle RdA di Tabella 2.1.	52
Figura 2.2 DTD relative ai dati di Figura 2.1.....	52
Figura 2.3 Albero DOM del file XML di Figura 2.1.	53
Figura 2.4 Un esempio di modello PMML.	56
Figura 2.5 Architettura del sistema KDDML.....	67
Figura 2.6 Codice dell'interprete KDDML.	70
Figura 2.7 Esempio di meta-dati utilizzati da KDDML.....	71
Figura 2.8 File XML rappresentante una query KDDML.	72
Figura 2.9 Albero DOM per la query di Figura 2.8.	74
Figura 2.10 Gerarchia degli oggetti del sistema KDDML.....	75
Figura 2.11 Risultato della query di Figura 2.8.....	75
Figura 2.12 La GUI attuale del sistema KDDML.....	76
Figura 3.1 WEKA GUI Chooser.....	87
Figura 3.2 Ambiente Knowledge Flow di WEKA.....	89
Figura 3.3 Scelta del data set da utilizzare come training set.	93
Figura 3.4 Inserimento e collegamento degli operatori per caricare e visualizzare il data set.....	94
Figura 3.5 Visualizzazione dei dati scelti come training set.	95
Figura 3.6 Inserimento degli operatori per la costruzione dell'albero di classificazione.	96
Figura 3.7 Flusso finale dei dati.	97
Figura 3.8 Albero di classificazione estratto.....	97
Figura 3.9 YALE GUI.....	102
Figura 3.10 Welcome screen di YALE.	105
Figura 3.11 Caricamento e visualizzazione del training set.....	106
Figura 3.12 Inserimento di operatori per la costruzione dell'albero di classificazione.....	107
Figura 3.13 Vista finale del nostro <i>esperimento</i>	108
Figura 3.14 Versione XML del nostro <i>esperimento</i> e del diagramma di <i>nesting</i> degli operatori presenti.....	108
Figura 3.15 Statistiche sul test set.	109
Figura 3.16 Ambiente ORANGE Canvas.	112
Figura 3.17 Finestra di visualizzazione del collegamento attivo tra due widget. ...	113
Figura 3.18 Dialog Box per la scelta del file contenente il training set.	116
Figura 3.19 Inserimento del nodo per la costruzione dell'albero di classificazione.....	117
Figura 3.20 Schema finale per la costruzione di un albero di classificazione.	118
Figura 3.21 Visualizzazione dell'albero estratto.	119

Figura 3.22 Visualizzazione dell'applicazione prodotta dallo schema di Figura 3.20.	120
Figura 3.23 GhostMiner Developer.	125
Figura 3.24 Import <i>wizard</i> di GhostMiner Developer.	128
Figura 3.25 Grafico rappresentante la distribuzione dei dati in input.	128
Figura 3.26 Inserimento e configurazione dell'operatore <i>SSV Tree</i>	130
Figura 3.27 <i>Treeview</i> del sistema per la generazione dell'albero di classificazione.	130
Figura 3.28 Albero di classificazione estratto.	131
Figura 3.29 Albero finale del <i>progetto</i> e matrice di confusione sul test set.	132
Figura 3.30 TANAGRA GUI.	136
Figura 3.31 Introduzione dell'attributo <i>status</i> nel data set.	139
Figura 3.32 Dialog Box per la scelta del data set, titolo e path per il salvataggio del diagramma.	140
Figura 3.33 Dialog Box per la configurazione del nodo <i>Select Example</i> e <i>Define Status</i>	141
Figura 3.34 Insieme di statistiche ed albero di classificazione prodotti dal sistema.	142
Figura 3.35 <i>Treeview</i> degli operatori del nostro progetto.	142
Figura 3.36 Risultato della classificazione.	143
Figura 4.1 <i>Treeview</i> rispettivamente di YALE (1.a), TANAGRA (1.b) e GhostMiner Developer (1.c).	148
Figura 4.2 <i>Wizard</i> di YALE.	153
Figura 4.3 Catena di operatori YALE per la lettura e l'applicazione di un modello e rispettiva versione in XML.	155
Figura 4.4 Albero degli operatori per l'estrazione di due modelli in YALE.	156
Figura 4.5 Albero degli operatori per l'estrazione di due modelli in TANAGRA.	156
Figura 4.6 Estrazione di due modelli dallo stesso data set.	157
Figura 4.7 Meta-classificatore costruito con GhostMiner Developer.	160
Figura 4.8 Comparazione tra metafora grafica, file XML risultante e disegno del <i>nesting</i> degli operatori in YALE.	161
Figura 4.9 Validazione di una query in YALE.	164
Figura 4.10 Esempio di finestra di Log di WEKA.	165
Figura 4.11 Esempio di meta-dati tipici dei file <i>arff</i> utilizzati da WEKA.	174
Figura 5.1 Query KDDML per estrarre due modelli diversi da dati diversi.	184
Figura 5.2 Query KDDML e corrispondente albero DOM.	185
Figura 5.3 Grafo rappresentante la query di Figura 5.2.	186
Figura 5.4 Estrazione di due modelli dallo stesso data set.	189
Figura 5.5 Albero DOM per la query di Figura 5.1.	189
Figura 5.6 Pseudo-Albero DOM per la query di Figura 5.4.b).	190
Figura 5.7 Schema riassuntivo sull'espansione di KDDML.	195
Figura 5.8 Esempio di processo KDD con utilizzo di meta-dati.	202
Figura 6.1 Architettura ad alto livello del sistema KDDML con l'introduzione del livello GUI.	211

Indice delle tabelle

Tabella 1.1 Gruppo di transazioni di acquisto.....	23
Tabella 1.2 Itemset frequenti relativi alle transazioni di Tabella 1.1.....	24
Tabella 1.3 Training set per l'estrazione di un albero di decisione.....	29
Tabella 2.1 Insieme di regole di associazione.....	51
Tabella 4.1 Tabella riassuntiva dei requisiti.....	180

Introduzione

Nella società moderna, l'utilizzo di sistemi automatici per reperire ed elaborare informazione è diventato un aspetto strategico importantissimo, con il quale ci confrontiamo quotidianamente, quando andiamo al supermercato, in banca o semplicemente su internet a visitare qualche sito.

Ogni giorno, che lo vogliamo o no, la nostra attività è “monitorata” da sistemi che registrano informazioni sui nostri comportamenti.

Questa attività, come ci possiamo aspettare, produce un'enorme quantità di dati che fino a trent'anni fa era inimmaginabile poter gestire. In questi ultimi anni, il drastico abbassamento del rapporto prezzo/capacità dei supporti di memorizzazione, a cui si è aggiunto un notevole potenziamento delle capacità di calcolo dei moderni elaboratori, hanno permesso la costruzione di potenti strumenti per organizzare e raccogliere i dati.

Tali obiettivi sono stati raggiunti con i moderni DBMS (Data Base Management System), che permettono di organizzare, memorizzare e reperire informazioni da database.

Purtroppo, nel corso del tempo, ci si è accorti che il solo utilizzo di strumenti DBMS non è sufficiente a garantire il pieno sfruttamento delle enormi quantità di dati di cui disponiamo. Un DBMS permette infatti di recuperare informazione che sappiamo già a priori essere presente nei dati, poiché memorizzata esplicitamente. Un esempio tipico è interrogare un database per conoscere il nome di tutti gli studenti di informatica che hanno superato l'esame di “Basi di Dati” nel corso del primo appello.

Il passo necessario per incrementare ulteriormente l'utilità dei dati è quindi quello di riuscire a trasformarli in informazione utile, attraverso la quale poter prendere decisioni per migliorare ad esempio la qualità dei servizi offerti ai propri clienti.

La situazione attuale è riassunta da J. Han e M. Kamber [HK00] con questa frase:

“Siamo ricchi di dati ma poveri di informazione”

Negli ultimi anni vi è stato quindi lo sforzo della ricerca, per riuscire ad estrarre informazione dall'enorme "miniera" di dati di cui disponiamo, che ha portato allo studio di un processo chiamato **Knowledge Discovery in Databases (KDD)**. Il termine processo in questo caso va inteso come una serie di stadi sequenziali ed interattivi, il cui il passo principale è comunemente noto come **Data Mining**, e si occupa dello sviluppo di modelli predittivi per evidenziare regolarità e similitudini nei dati.

In quest'ottica, la situazione attuale può essere rapportata a quella dei DBMS nei primi anni '60, quando ogni applicazione doveva essere costruita dal nulla, senza il vantaggio di primitive dedicate, fornite in seguito dal linguaggio SQL.

Riassumendo, possiamo dire che, mentre gli attuali DBMS si occupano di memorizzare ed organizzare i dati di interesse, il KDD si propone di mettere a disposizione dell'analista gli strumenti per analizzare, capire e visualizzare la conoscenza contenuta in tali dati.

Con l'evoluzione della tecnologia informatica è cambiata anche l'interazione che l'uomo ha con essa. L'enorme diffusione dei personal computer in parte è dettata dai costi sempre più contenuti, ma anche dallo sviluppo di applicazioni che garantiscono un'interazione uomo-macchina sempre più semplice ed intuitiva, che permette, anche a chi non ha conoscenze specifiche, di utilizzare sistemi molto complessi come quelli attuali.

Se fino a trent'anni fa per copiare dei file da una cartella all'altra bisognava digitare dei comandi, adesso sono sufficienti pochi movimenti di mouse, senza nemmeno toccare la tastiera. È indubbio che il livello d'interazione tra uomo e macchina in questi anni ha subito una notevole trasformazione, andando ad incrementare enormemente la diffusione e le potenzialità di strumenti che altrimenti sarebbero rimasti confinati solo in alcuni settori specifici. Al giorno d'oggi, il PC è divenuto uno strumento indispensabile per una vasta categoria di professioni, anche molto diverse da quelle del settore informatico.

Quanto appena espresso deve essere quindi legato con le osservazioni riguardanti sia il mondo dei database che il processo di estrazione della conoscenza. L'introduzione di astrazioni grafiche ha semplificato enormemente la vita all'utilizzatore finale, ma ha complicato lo sviluppo delle applicazioni.

Se, ad esempio, nel mondo dei database relazionali, il fatto di avere un query language standard ed ormai affermato come SQL ha favorito anche un fiorente sviluppo di applicazioni grafiche, tanto è vero che oggi si parla ormai di *query-by-example*, nel mondo del data mining e del knowledge discovery non si può dire altrettanto.

È indubbio che un'applicazione di KDD, per essere utilizzata anche al di fuori dei soli contesti di ricerca, deve fornire all'utente potenti strumenti grafici, per rappresentare il processo e per garantirne un'ottima interattività, che come vedremo nel capitolo 1, è una caratteristica fondamentale nell'ambito di tale settore.

La mancanza di standard precisi ha portato i vari sviluppatori a seguire cammini proprietari, per la rappresentazione del processo, creando varie filosofie di rappresentazione.

Nel corso di questa tesi, il nostro compito sarà pertanto quello di valutare alcuni dei maggiori software di data mining, sviluppati in ambito universitario e di ricerca, al fine di studiarne i linguaggi grafici per rappresentare il processo KDD che essi offrono all'utente.

Questo studio sarà indirizzato alla stesura di alcuni importanti requisiti che un linguaggio grafico deve possedere, ed ampliando il nostro raggio di azione, in taluni casi proporremo anche requisiti che riguardano caratteristiche di un intero sistema di KDD, e non della sola interfaccia grafica.

La necessità di questo studio è essenziale, al fine di introdurre un'interfaccia grafica in un sistema esistente sviluppato presso il Dipartimento di Informatica dell'Università di Pisa. Tale sistema, conosciuto come **KDDML** (Knowledge Discovery in Databases Markup Language), rappresenta un ambiente per l'esecuzione di processi KDD e, come vedremo nel capitolo 2, ha come punto di forza quello di prevedere un vero e proprio query language basato su XML, attraverso il quale esprimere processi KDD.

Purtroppo, la stesura di questi processi è ancora testuale e prevede che un utente scriva il file XML rappresentante la query KDD da sottoporre al sistema, senza l'ausilio di alcuna astrazione grafica che ne semplifichi l'utilizzo.

ORGANIZZAZIONE DELLA TESI

- **Capitolo 1:** viene presentata una panoramica sullo stato dell'arte, con particolare attenzione al processo KDD, ai suoi obiettivi ed alle sue aree applicative. Ci soffermeremo in particolare sulla fase di data mining, soprattutto relativamente a regole di associazione, clustering e classificazione.
- **Capitolo 2:** nel corso di questo capitolo forniamo una breve descrizione sia del sistema KDDML, che del linguaggio basato su XML ad esso associato. Il capitolo si apre con una breve introduzione riguardante XML e PMML.
- **Capitolo 3:** in questo capitolo vengono analizzati diversi sistemi per il knowledge discovery, in particolar modo in riferimento ai loro aspetti visuali, soprattutto valutando il livello di interattività e la metodologia offerta per la costruzione e l'esecuzione del processo KDD. Per ogni software analizzato è infine fornito un esempio, sia per chiarire meglio alcuni concetti espressi in corso di analisi, che per presentare un primo nucleo comune di confronto.
- **Capitolo 4:** vengono approfondite e confrontate alcune caratteristiche rilevate nel capitolo precedente. Gran parte del confronto riguarda aspetti grafici e di interazione tra software ed utente. Vengono inoltre analizzati elementi riguardanti la visualizzazione dei risultati e le caratteristiche di espandibilità dei sistemi.

Il capitolo si conclude con la redazione di una serie di requisiti che, a nostro avviso, un sistema di data mining, ed in particolare la sua interfaccia grafica, devono soddisfare. Tali requisiti, elaborati sulla base dello studio condotto nei capitoli 3 e 4, cercano di considerare e condensare molteplici aspetti relativi alla costruzione di un software di knowledge discovery.
- **Capitolo 5:** in quest'ultimo capitolo, cerchiamo di valutare se e come i parametri stilati nel capitolo 4 sono soddisfatti dal sistema (e linguaggio) KDDML,

andando ad analizzare non solo se questi possono adattarsi al sistema esistente, ma proponendo i requisiti per una futura implementazione dell'interfaccia grafica per tale sistema.

Capitolo 1

Data Mining & Knowledge Discovery in Databases

1.1 INTRODUZIONE

L'introduzione di strumenti sempre più efficaci per la raccolta e la generazione dei dati, insieme al costante progresso delle tecnologie di memorizzazione ed elaborazione delle informazioni, ha reso disponibile una straordinaria quantità di dati.

Per citare qualche esempio, l'osservatorio terrestre della NASA ha stimato di dover raccogliere ed elaborare ogni ora oltre 50 gigabyte di dati provenienti dai suoi sistemi di monitoraggio terrestre e spaziale [WS91].

Un altro importante studio sul genoma umano ha portato alla creazione di un enorme database in costante crescita, in cui sono collezionati gigabyte di dati sul DNA umano [FCK94].

L'informazione e la conoscenza estratta da questa sterminata quantità di dati può avere ambiti applicativi pressoché illimitati, dal campo economico, come nell'amministrazione dei piani di gestione di un'azienda, fino a quello scientifico e biologico-medico.

Purtroppo, raccogliere e organizzare molti dati non significa necessariamente disporre esclusivamente di informazione utile, in quanto nella maggior parte dei casi

tali informazioni sono spesso nascoste all'interno di grossi volumi di dati di per sé poco significativi o incompleti.

Il data mining (DM) può essere quindi visto come il risultato della naturale evoluzione tecnologica volta alla ricerca di informazioni potenzialmente utili, in modo che possano essere comprese dall'uomo ed utilizzate come supporto alle decisioni.

Dare una definizione rigorosa e unica di DM è pressoché impossibile, poiché nella letteratura attuale sono presenti un'infinità di caratterizzazioni. Alcune definizioni sono legate alla presenza di tecniche specifiche, come le reti neurali o algoritmi genetici, mentre altre presentano il DM come soluzione a tutti i problemi aziendali. Tutte queste caratterizzazioni non prendono in considerazione il fatto che il DM non è legato ad alcuna tecnica specifica, e non rappresenta una soluzione universale.

Possiamo essenzialmente definire il DM come *“un procedimento atto all'estrazione di conoscenza da vaste quantità di dati, e rappresentata attraverso un piccolo insieme di preziose informazioni”* [HK00].

Molto spesso il DM è visto come sinonimo di un altro termine ampiamente usato come **Knowledge Discovery in Databases (KDD)**, mentre in alcuni casi il DM è accreditato come un passo essenziale del processo di knowledge discovery (KD).

Di seguito, faremo riferimento a questa seconda definizione, adottando una visione di DM come ingranaggio fondamentale di un meccanismo piuttosto complesso e variegato come il KDD.

Nel prossimo paragrafo cercheremo di chiarire il concetto di conoscenza, e proporremo una breve sintesi sul KDD, e sulle principali tecniche di DM. Concluderemo infine il capitolo con alcune considerazioni volte a capire le direzioni di sviluppo futuro del KDD.

1.2 LA CONOSCENZA

Prima di introdurre il processo di KDD, al fine di comprendere meglio gli obiettivi di tale meccanismo, vogliamo soffermarci brevemente sul concetto di conoscenza

che intendiamo adottare. La conoscenza [WB98] è un insieme di informazioni che vengono estratte dai dati, ma come tale può essere considerata sotto due diversi aspetti

- **Conoscenza reale (Actual Knowledge):** rappresenta l'informazione contenuta nei dati, e che può essere realmente conosciuta o meno.
- **Metaconoscenza (Metaknowledge):** rappresenta l'informazione che potrebbe essere presente nei dati e che, a priori, si pensa correttamente o erroneamente, di conoscere o non conoscere.

Questi due aspetti diversi della conoscenza vengono legati e rappresentati dalla Figura 1.1.

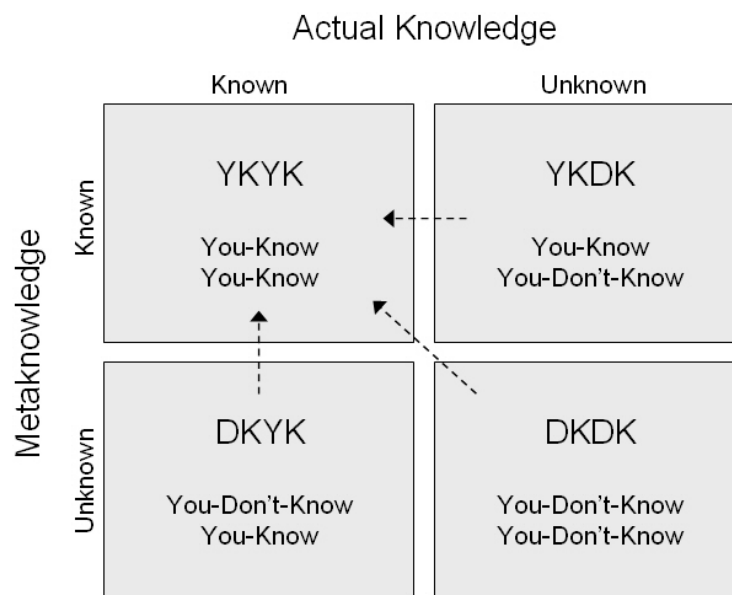


Figura 1.1 Conoscenza e Metaconoscenza.

Come si può notare, la Figura 1.1 è divisa in quattro parti, che rappresentano i vari stadi della conoscenza:

- **So di sapere (YKYK – You know you know):** è il caso più semplice, in cui le informazioni esistono realmente nei dati, e si ha la consapevolezza della loro esistenza. Ad esempio, si sa che un'automobile senza carburante non cammina, che la temperatura di ebollizione dell'acqua è 100°C, e che se passiamo con il rosso davanti ad un vigile urbano, è molto probabile ricevere una multa. Molte di queste informazioni riguardano conoscenze evidenti, stabilite e molto spesso date per scontate, e molti sistemi attualmente in uso nelle industrie si basano in gran parte su questo tipo di informazioni, poiché sono estremamente affidabili e possono essere facilmente introdotte all'interno di regole.
- **So di non sapere (YKDK – You know you don't know):** si tratta di informazioni che si potrebbero conoscere, ma che non sono evidenti o direttamente accessibili nei dati, per conoscere le quali sono necessarie ulteriori ricerche. Tuttavia, appartengono a questa categoria anche conoscenze che non rientrano nella normale attività di analisi, come quelle relative a domande del tipo: *“se passo col giallo ed il vigile mi vede, mi fa la multa?”*. La risposta in questo caso è incerta e dipende da vari fattori, come se il semaforo è diventato giallo mentre la mia auto stava già attraversando l'incrocio, se ci sono condizioni meteorologiche avverse che possono scoraggiare il vigile dall'abbandonare il suo veicolo, se il vigile è permissivo, è giunto alla fine del suo turno di lavoro etc. Il KDD consentirebbe di raggiungere comunque lo stato precedente.
- **Non so di sapere (DKYK – You don't know you know):** questo è il caso in cui le informazioni rilevanti sono presenti nei dati e sono anche facilmente ricavabili, ma non si ha la loro metaconoscenza, ovvero non si pensa che possano esistere. Immaginiamo adesso che un'auto attraversi l'incrocio con il verde. Dal punto di vista del vigile tutto appare normale, ma supponiamo di poter fare in quell'istante un controllo sulla targa, e che la macchina risulti rubata. La situazione in quel caso cambierebbe radicalmente. È qui che il

KDD raggiunge la sua massima utilità, poiché vi è la possibilità di scoprire interessanti informazioni che esistono già nei dati, ma non vengono al momento utilizzate, in quanto non immediatamente evidenti. Lo scopo degli esperimenti di DM riguardo alla conoscenza di tipo “*non so di sapere*” è quello di trasformarla nel più breve tempo possibile in informazione della categoria “*so di sapere*”.

- **Non so di non sapere (DKDK – You don’t know you don’t know):** è il caso in cui vi sono importanti lacune o mancanze nei dati, e non si ha la consapevolezza di ciò, perché non abbiamo a disposizione nemmeno la metacoscienza. Supponiamo che a condurre la macchina di cui sopra, vi sia un conducente senza patente, o sotto l’effetto di stupefacenti. Dal momento che il vigile in questo caso sta semplicemente controllando gli automobilisti che passano con il semaforo rosso, è molto probabile che le infrazioni commesse dal nostro guidatore rimangano impunte, a meno che il vigile non venga richiamato da un comportamento anomalo alla guida. Qui il KDD non ha alcun obiettivo da raggiungere.

Questa è solo una prima visione delle problematiche legate alla conoscenza, definite in maniera generale. Un altro punto di vista, legato in particolar modo al mondo dei database, la suddivide in base al suo livello di profondità nel reperimento delle informazioni, fornendo principalmente tre livelli:

1. **Conoscenza superficiale:** si è consapevoli a priori di ciò che si vuole trovare, dove e come trovarlo, e l’informazione è memorizzata esplicitamente nel database e recuperabile con diversi strumenti di interrogazione.
2. **Conoscenza nascosta:** qui si presume che vi siano degli elementi nei dati presenti nel database, che permettano di estrarre delle caratteristiche che contraddistinguono alcune entuple da altre. Questo è il regno del KDD.

3. **Conoscenza profonda:** è l'informazione presente nel database, che può essere trovata solo se si ha un indizio riguardo a dove cercarla. In questa situazione non vi è alcuna ipotesi di base.

Queste caratterizzazioni ci permettono di capire quanto i dati, ciò che noi conosciamo e ci aspettiamo da essi, siano proprietà fondamentali per mettere in luce cosa realmente significa ottenere della conoscenza.

1.3 KNOWLEDGE DISCOVERY IN DATABASES

I sistemi classici di memorizzazione dei dati, chiamati DBMS (DataBase Management System) permettono di memorizzare ed accedere ai dati in maniera sicura, efficiente e veloce. Tuttavia, non sempre si è avuta la possibilità di affiancare a questi ultimi anche strumenti per favorire l'estrazione di informazioni utili come supporto alle decisioni.

Fortunatamente negli ultimi anni si è compresa l'importanza di estrarre informazione non esplicitamente memorizzata nella base di dati, e grazie al contributo sinergico di diverse discipline, tra cui intelligenza artificiale, reti neurali, statistica ed apprendimento automatico, sono apparse tecnologie in grado di svolgere questo compito. Infatti oggi un numero sempre crescente di RDBMS (Relational DBMS) integra al proprio interno funzionalità tipiche del KDD.

Anche nel caso del KDD, non è semplice trovare una definizione univoca, ci limitiamo pertanto a fornire una di quelle di maggior successo data da Fayyad, Piatetsky-Shapiro, Smyth e Uthurusamy in [FPSSU96]:

“Per KDD si intende il processo non banale di identificazione dei modelli validi, nuovi, potenzialmente utili e comprensibili sui dati”

Questa definizione ha indubbiamente il pregio di esporre in maniera sintetica quelli che sono gli elementi essenziali del KDD. Una caratteristica che tale definizione mette subito in risalto è che il KDD è un **processo**, e come tale prevede

una serie di fasi iterative, che in seguito descriveremo più in dettaglio. Poniamo ora la nostra attenzione sul significato di alcuni di questi termini:

- **Dati:** rappresentano un insieme di fatti (tipicamente tuple di un database).
- **Modelli:** un modello è rappresentato da un'espressione che determina un insieme di fatti ricavati dall'insieme di partenza. Per i modelli sono richieste delle caratteristiche ben precise che comprendono:
 - **Validità:** un modello deve essere attendibile e quindi garantire anche su nuovi dati risultati simili a quelli raggiunti sull'insieme di partenza, con un certo grado di certezza. Tale misura è una funzione che lega il modello ad un valore che ne indica la validità.
 - **Novità:** un modello deve essere nuovo, o dal punto di vista del cambiamento sui dati, attraverso il confronto dei valori attuali con quelli previsti, oppure dal punto di vista della conoscenza espressa, considerando quanto la nuova è in relazione con quella esistente in passato.
 - **Utilità potenziale:** i modelli individuati dovrebbero potenzialmente condurre l'utente a compiere azioni utili.
 - **Facilità di comprensione:** i modelli ottenuti devono essere di semplice comprensione e devono permettere all'utente di facilitare e migliorare la comprensione dei dati.

Da quanto appena espresso, risulta chiaro che la conoscenza rappresentata dai modelli è fortemente in relazione con le misure di novità, validità, utilità, e facilità di comprensione. Un aspetto importante che lega tali caratteristiche e le riassume in un unico valore è la misura di *interestingness*, che viene generalmente assunta come misura generale della bontà di un modello.

1.3.1 FASI DEL KDD

Gli stadi che caratterizzano un processo KDD sono stati identificati nel 1996 in [FPSSU96] e sono rappresentati in Figura 1.2. Nell'elencare e descrivere queste fasi tali studiosi pongono particolare attenzione sullo stadio del DM, ovvero gli algoritmi per l'esplorazione e lo studio dei dati.

Il DM è ritenuta la fase più importante dell'intero processo KDD, e questa sua enorme importanza è alla base della confusione tra la distinzione di processo KDD e DM. All'interno della nostra trattazione, come detto in precedenza, cercheremo costantemente di separare i due concetti, e considereremo il DM come la fase più significativa del processo KDD, ma non perfettamente coincidente con esso. Iniziamo quindi una breve descrizione delle fasi costitutive del processo, che ricalca sostanzialmente il modello ideato dagli studiosi appena citati.

Il processo è interattivo, in quanto necessita di un intervento umano su alcune decisioni, soprattutto sulla fase di DM, durante la scelta dell'algoritmo specifico e del modello di dati da utilizzare. Come si può osservare dalla Figura 1.2, il processo parte dai dati grezzi in input, provenienti da sorgenti diverse e produce come output informazioni utili, che sono acquisite come il risultato dei seguenti passi:

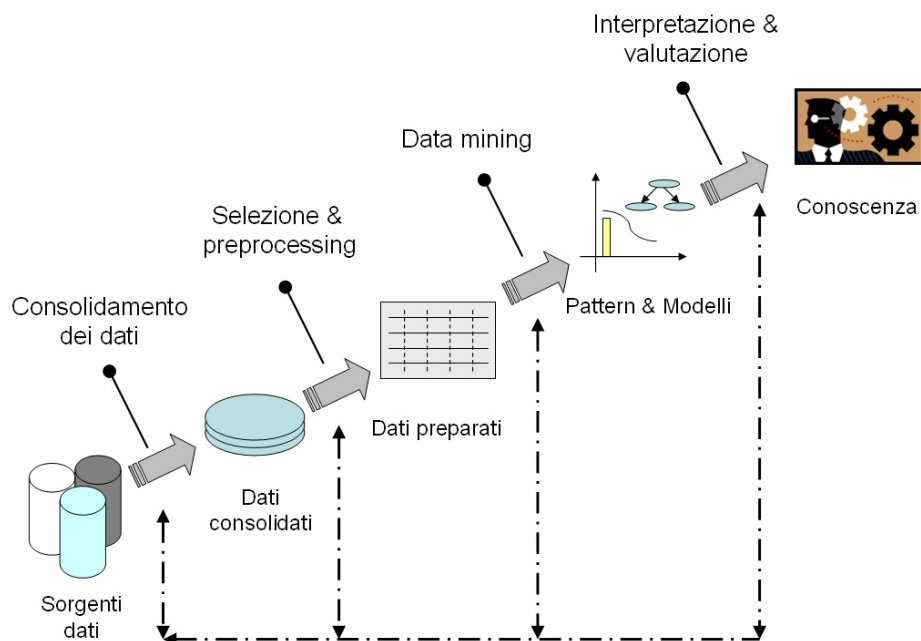


Figura 1.2 Fasi del processo KDD.

- **Consolidamento dei dati:** i dati grezzi, prelevati da sorgenti eterogenee, come DB relazionali, file di testo, etc., attraverso questa fase, vengono segmentati e selezionati secondo alcuni criteri, al fine di pervenire ad un sottoinsieme di dati, che rappresenterà il nostro insieme operativo.

Risulta chiaro come questa fase serva per costruire una visione uniforme degli stessi a prescindere dal tipo di sorgente da cui siano stati prelevati. Tipicamente il risultato di questo passo conduce alla creazione di un database, detto **data warehouse**, utilizzato per il supporto alle decisioni e mantenuto separato dal database operativo dell'organizzazione, che è invece utilizzato per le normali operazioni di transaction processing.

- **Selezione & Preprocessing:** una volta costruito il nostro target data, è necessario trasformare ulteriormente i dati. Questo passo ha il duplice scopo di eliminare eventuali informazioni presenti ma inutili alla nostra analisi, e di rendere i dati adatti agli algoritmi di DM, uniformandone il formato. Tali algoritmi e tool richiedono, infatti, che i dati forniti siano in un formato ben preciso e privi di errori, mentre i dati ottenuti dal passo precedente provengono da fonti eterogenee, e l'integrazione prodotta può aver portato a possibili inconsistenze, ridondanze o errori.

Sono tipici di questa fase i seguenti task:

- **Data Cleaning** ("pulizia dei dati"): si occupa di ripulire i dati e di prepararli per la fase di DM. In questa direzione, i problemi che il data cleaning cerca di risolvere sono sostanzialmente due:
 - *Gestione di valori mancanti:* pone l'attenzione su come sostituire i valori mancanti presenti in un target data, proponendo svariate soluzioni, dalle più semplici come ignorare le entuple in cui è presente un valore mancante, o sostituire ogni mancanza con un

valore di default, a tecniche più complesse basate su calcoli probabilistici.

- *Gestione di errori ed eccezioni*: questa voce si riferisce alla decisione dei meccanismi di comportamento in caso di dati considerati estranei, ad esempio nel caso in cui il loro valore si discosti troppo dalla media.
- **Data Integration** (“integrazione dei dati”): si occupa di rimuovere tutte le eventuali ridondanze ed inconsistenze presenti nei dati. Risulta piuttosto frequente incontrare problemi di sinonimia, in quanto la molteplicità delle sorgenti da cui i dati vengono prelevati porta ad avere un attributo che rappresenta lo stesso oggetto con nomi differenti, oppure valori in realtà equivalenti che si possono presentare sotto nomi diversi, perché memorizzati con formalismi differenti. I dati ridondanti possono essere individuati con tecniche di correlazione, ed il problema può essere risolto rendendo omogenea la nomenclatura a cui fare riferimento.
- **Data Reduction** (“riduzione dei dati”): si occupa di stabilire quali sono i dati da utilizzare, e quali risultano inutili durante il processo. In questa fase vengono eliminati gli attributi non significativi per l’analisi da attuare, o vengono campionate le tuple, al fine di ridurre ulteriormente la mole dei dati in input alla fase successiva. Tale riduzione deve comunque mantenere l’integrità dei dati, favorendo l’efficienza del processo senza, comprometterne i risultati.
- **Data Transformation** (“trasformazione dei dati”): si occupa sostanzialmente di combinare tra loro gli attributi, e di ridurre il range dei valori degli stessi. Tipica di questa fase è la normalizzazione dei dati, in base alla quale, al fine di ottenere risultati coerenti con l’analisi svolta, si

rendono omogenei i dati. I criteri per tale trasformazione variano a seconda degli scopi del processo.

Questo stadio di preparazione dei dati non solo è necessario, ma è considerato uno dei più impegnativi dell'intero processo di estrazione della conoscenza.

Durante la fase di preprocessing, può essere utile ricorrere a tecniche di visualizzazione dei dati, che mettono a disposizione un modello concettuale del data warehouse. I tool **OLAP (On Line Analytic Processing)** [CD97], ad esempio, si basano su strutture multidimensionali, dette **Data Cube**, che offrono la possibilità di aggregare i dati in base ad ogni dimensione, visualizzare grafici o eseguire calcoli statistici.

- **Data Mining:** attraverso questa fase, i dati trasformati vengono processati usando una o più tecniche di DM, in modo da poter estrarre il tipo di informazione desiderata. Questo stadio include la scelta del tipo di metodo, dell'algoritmo da utilizzare per la ricerca, per la scoperta dei modelli e delle regolarità presenti nei dati. Mentre si effettua questo passo, può essere necessario attuare ulteriori trasformazioni sui dati originali, oppure accedere ad informazioni aggiuntive, ovvero ripercorrere alcuni passi visti nelle fasi precedenti.

Le tecniche di DM devono inoltre trattare considerevoli quantità di dati in tempi accettabili e considerati “normali” dall'utente. Dato un insieme di dati, si tratta di capirne la struttura ed il contenuto e scegliere la tecnica più adatta al caso. I risultati ottenuti dall'analisi devono permettere di comprendere meglio la dinamica dei fenomeni in studio.

La fase di DM è sicuramente il passo principale e più importante di tutto il processo, e per questo approfondiremo alcune tecniche nel paragrafo 1.4.

- **Interpretazione e valutazione:** questa fase rappresenta la componente conclusiva del processo KDD, in cui i modelli estratti ed identificati dal

processo rappresentano conoscenza potenzialmente utile, utilizzabile dall'analista o dal manager come supporto alle decisioni.

Lo scopo principale dell'interpretazione non è soltanto quello di visualizzare l'output del passo di DM, ad esempio attraverso grafici, ma anche di filtrare, ovvero valutare in che misura la conoscenza che verrà presentata risulta utile. Bisogna evidenziare il fatto che non tutta la conoscenza estratta rappresenta informazione utile, il processo infatti può produrre un gran numero di pattern la cui utilità risulta nulla. È necessario quindi definire uno schema, che in base a misure di *interestingness*, permetta di selezionare solo la conoscenza effettivamente rilevante all'interno di quella estratta dal sistema.

I parametri utilizzati per valutare la bontà di una soluzione possono essere sia di tipo oggettivo, basati su statistiche, che di tipo soggettivo, quindi basati su peculiarità presenti nel modello, che l'utente reputa importanti. La tecnologia attuale permette di avvalersi di criteri di visualizzazione, come istogrammi o animazioni, utili all'analista per comprendere l'utilità della conoscenza estratta e stabilire le decisioni finali.

È inoltre possibile, in caso di risultati non perfettamente soddisfacenti, ripetere una o più fasi dell'intero processo, ed affinare e adattare la conoscenza estratta alle proprie esigenze.

1.3.2 CICLO VIRTUOSO

Come abbiamo potuto osservare, il KDD è un processo “*non banale*”, il cui compito non si limita a scoprire della conoscenza, ma ne richiede anche la comprensione, in modo da poter agire su di essa e trasformare i dati in informazione, l'informazione in azione e l'azione in valore.

Questo concetto viene presentato in Figura 1.3, ed è noto in letteratura come *ciclo virtuoso* [BL97].

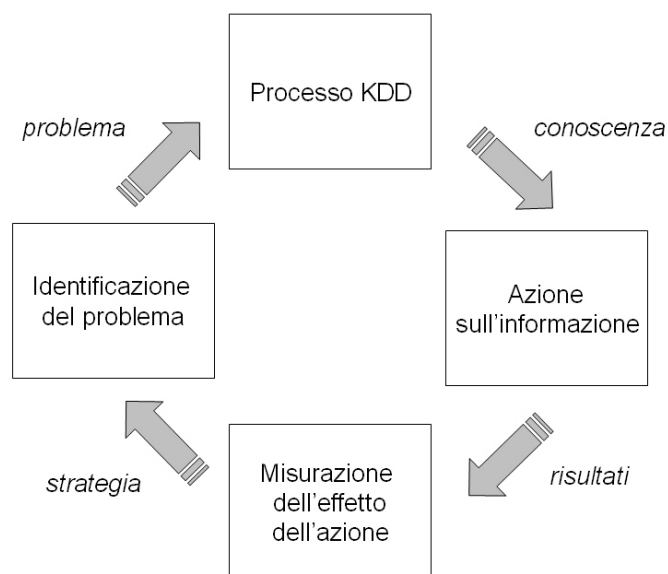


Figura 1.3 Ciclo virtuoso.

Il ciclo virtuoso si compone di quattro fasi, in cui ogni fase dipende dai risultati ottenuti dalla precedente. Forniamo ora una breve descrizione di tali fasi:

1. **Identificazione del problema:** lo scopo di questo stadio è identificare i settori in cui la scoperta della conoscenza possa essere di supporto alle strategie da intraprendere, al fine di incrementare il guadagno ottenibile. Spesso, questi settori sono legati a problematiche ed osservazioni sollevate a vari livelli di un'azienda. Il tutto parte da semplici domande del tipo: *“Perché i negozi in California vendono meno di quelli del Sud?”*, oppure *“Qual è il prodotto che potremmo promuovere nel negozio del Colorado?”*, per approdare a pianificazioni di vendita, alla caratterizzazione delle abitudini dei clienti, etc. Bisogna sottolineare che, senza l'accesso ai dati, molte delle osservazioni non sarebbero possibili e molte domande non troverebbero risposta.
2. **Knowledge Discovery in Databases:** stabilita l'area di applicazione e formulato il problema, si procede alla raccolta dei dati, e si utilizza il processo KDD che, partendo da tali dati grezzi derivati da formati

eterogenei, produce dei risultati, ovvero modelli o pattern che possono essere utilizzati nelle fasi successive. In questa fase risulta cruciale identificare le opportune sorgenti dati, raccogliere e proteggere i dati giusti.

3. **Azione sull'informazione:** una volta ottenuta la conoscenza dal processo KDD, questa viene analizzata e valutata da esperti del settore e conseguentemente utilizzata all'interno del supporto decisionale. In questo modo, le azioni diventano parte integrante del ciclo virtuoso. Si può così decidere, sempre in riferimento al marketing, di promuovere un nuovo prodotto, proporre incentivi e sconti ai clienti, oppure aprire un nuovo punto vendita localizzato in un'area geografica di interesse.
4. **Misurazione dell'effetto delle azioni:** dopo aver eseguito le azioni pianificate, è necessario confrontare i risultati ottenuti con le nostre previsioni. Grazie a questa fase, si comprende se e come le azioni compiute hanno effettivamente portato alla risoluzione del problema che era stato proposto. Queste misure forniscono, quindi, lo stimolo per continuare a migliorare i risultati ed identificare nuove opportunità. Il processo a questo punto può ricominciare, per ottenere risultati ancora più soddisfacenti all'iterazione successiva. I metodi di misura dipendono ovviamente dal tipo di azioni trattate, ad esempio, ci si può chiedere *“come siano incrementate le vendite in California, in seguito alla promozione del nuovo prodotto”*.

Come risulta chiaro, sia dalla struttura del processo KDD, che dal ciclo virtuoso appena definito, il ruolo chiave e centrale di tutto il meccanismo di *“costruzione della conoscenza”* è riservato all'uomo. Il processo KDD è difficilmente automatizzabile, e la natura stessa del problema richiede che il processo sia interattivo ed iterativo, e richiede inoltre il coinvolgimento di varie professionalità.

Come osservato da Fayyad, Piatetsky-Shapiro, Smyth e Uthurusamy in [FPSSU96], *“la cieca applicazione di metodi di DM può essere un'attività pericolosa, che conduce sovente alla scoperta di modelli privi di senso”*.

L'interazione umana va applicata alle diverse fasi del KDD, ed è essenziale per l'effettivo utilizzo di algoritmi di DM, e per il successo dell'intero processo KDD, che si riflette di conseguenza sull'intero ciclo virtuoso.

La partecipazione umana è quindi prevista in varie fasi, e comprende i contributi di diverse figure professionali tra cui:

- **Analista:** provvede a stabilire i punti base del problema, esamina la situazione attuale e le soluzioni esistenti, studiando inoltre a fondo le potenzialità che un processo KDD può far emergere in tale dominio.
- **Ingegnere della conoscenza:** si occupa dell'analisi concreta degli obiettivi del DM, ed è di sua competenza la preparazione dei dati per l'applicazione degli algoritmi. Applica inoltre le tecniche appropriate per generare i risultati per la fase di mining. Dopo aver ripetuto tali passaggi con differenti valori, stabilisce quale dei sistemi sperimentati è il migliore, e si occupa della creazione di una generica procedura che segua automaticamente tali passi.
- **Sviluppatore di applicazioni:** si occupa principalmente della realizzazione di sistemi appropriati per il KDD, e provvede ad integrarli nell'ambiente commerciale. Sono di sua competenza le analisi delle condizioni tecniche e organizzative del mercato.
- **Utente finale:** ignora le tecnologie di DM e il processo KDD, ma conoscendo il problema concreto, cerca di risolverlo attuando politiche reali, attraverso l'utilizzo di tali tecniche.

1.3.3 ARCHITETTURA TIPICA DI UN SISTEMA KDD

Nel paragrafo precedente, abbiamo visto le varie figure che ruotano attorno al KDD, ed abbiamo parlato di sviluppo e realizzazione di tali sistemi. Per mostrare più

chiaramente la complessità di tale compito, forniamo adesso una breve descrizione degli elementi fondamentali che caratterizzano l'architettura di un sistema KDD [HK00]. Tale architettura, mostrata in Figura 1.4, è comunque di carattere generale e comprende solo le componenti principali.

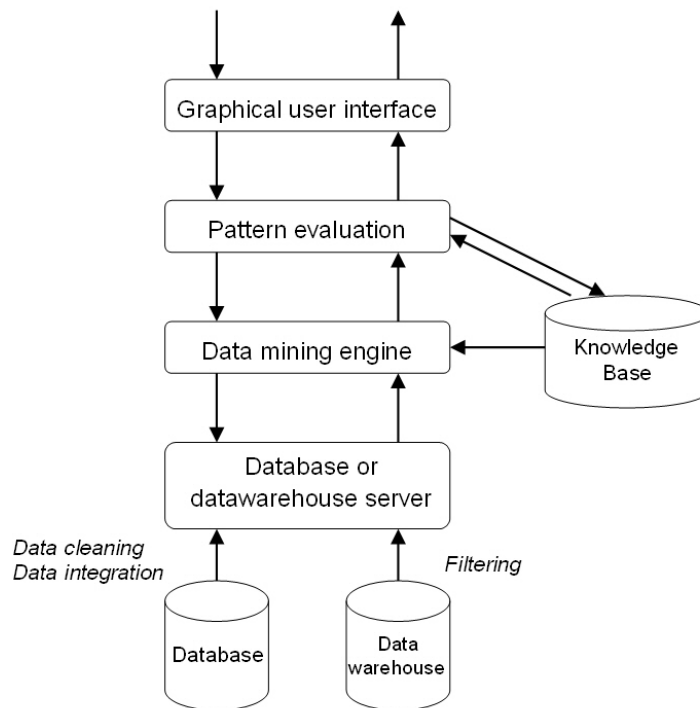


Figura 1.4 Architettura di un sistema per KDD.

Partendo dal livello più basso, e salendo troviamo:

- **Database, data warehouse, or other information repository:** rappresentano le sorgenti dei dati su cui attuare il processo. Oltre a uno o più database (o data warehouse), possiamo avere *spreadsheet* o altri tipi di *information repository*.
- **Database o data warehouse server:** questa componente si occupa di garantire ai livelli superiori il flusso dei dati rilevanti, basati sulle richieste di DM effettuate dall'utente.

- **Knowledge base:** rappresenta il dominio di conoscenza che è usato per guidare la ricerca e la valutazione dei pattern risultanti. In tale conoscenza può essere incluso il concetto di gerarchia, usato per organizzare attributi a diversi livelli di astrazione.
- **Data mining engine:** rappresenta il cuore di ogni sistema KDD. In esso sono racchiuse tutte le funzionalità di DM che il sistema fornisce.
- **Pattern evaluation module:** questa componente si occupa principalmente di fornire misure di *interestingness*, interagisce con il modulo **Data mining engine**, al fine di focalizzare la ricerca solo sui pattern realmente interessanti. Grazie a questo modulo, possono essere eliminati tutti i pattern ritenuti non importanti, poiché sotto una certa soglia di *interestingness*. Questo modulo può anche essere integrato nel modulo di DM.
- **Graphical user interface (GUI):** si occupa della comunicazione tra l'utente ed il sistema e permette all'utente di interagire con esso. Attraverso la specifica di task KDD, fornisce informazioni per mettere a fuoco la ricerca e per esplorare i risultati ottenuti.

1.3.4 SETTORI APPLICATIVI DEL KDD

Da quando detto finora, possiamo dunque affermare che il KDD può essere visto come la naturale evoluzione delle tecnologie che trattano l'informazione.

C'è infatti un crescente entusiasmo attorno al DM, ed in generale al processo KDD, e sono molti i settori in cui queste tecniche vengono utilizzate. Questi domini applicativi spaziano dalla gestione degli investimenti di una società finanziaria, all'astronomia, dal rilevamento di possibili malfunzionamenti in un sistema, alla classificazione di oggetti presenti in un ambiente da parte di un robot.

Inoltre, l'importanza del KDD è in rapido aumento in settori, come quello assicurativo, in cui lo studio del comportamento del cliente è centrale per qualsiasi

decisione da intraprendere, oppure quello della sanità, al fine di migliorare le diagnosi ed effettuare un monitoraggio mirato dei pazienti, riducendo i costi del servizio. Altre possibili applicazioni riguardano i sistemi di telecomunicazione, banche, vendita al dettaglio e marketing.

Bisogna inserire nel nostro elenco anche la vertiginosa espansione di internet, la quale ha generato nuovi domini applicativi per il DM. Basti pensare al commercio elettronico ed a tutte le attività ad esso associate, che stanno creando, e continueranno a creare, enormi quantità di dati utilizzabili per scopi commerciali e sociali.

Altro argomento, in parte legato al mondo del web, riguarda le applicazioni di **text-mining**, che permettono, ad esempio, analizzando pagine di testo da siti web a carattere finanziario, di prevedere gli andamenti dei titoli in borsa.

L'utilizzo del DM applicato ad internet non si ferma al text-mining, ma coinvolge anche la ricerca di strumenti per rendere i motori di ricerca più efficienti, veloci e precisi, insieme a nuovi algoritmi di *proxy catching intelligente* (**Web Mining**).

Anche l'area dei **dati rilevati in maniera remota** rappresenta un'altra possibile applicazione del KDD, essendo la più vasta sorgente di dati disponibile. Gran parte di questi dati deriva dalle immagini satellitari, mentre altri dati provengono da rilevamenti effettuati sulla terra e sul mare, o dai radar. Il loro impiego è nato dall'esigenza di monitorare e capire le principali cause di inquinamento del nostro pianeta. Nel campo satellitare, le reti neurali sono correntemente utilizzate come strumento di riconoscimento e di ricostruzione di immagini.

Perfino il mondo dello **sport** può essere interessato all'applicazione di tecniche di DM. In [Hed96] viene presentato un esempio di sistema KDD sviluppato per l'NBA (National Basketball Association), che è stato utilizzato da ben 14 delle 29 squadre della lega americana. Il sistema memorizza informazioni statistiche sui dati degli eventi di ogni singolo incontro, come passaggi, tiri liberi, falli commessi, etc., che vi vengono inseriti durante il gioco. Utilizzando tale sistema, gli allenatori possono, in tempo reale, porre quesiti al software, per trovare schemi di gioco che portino la squadra ad assumere una strategia ritenuta vincente.

1.4 DATA MINING

Come abbiamo osservato in precedenza, il DM rappresenta la fase fondamentale del processo KDD, e consiste nell'estrarre modelli e pattern dai dati, allo scopo di ottenere informazione utile e comprensibile. Tale estrazione deve avvenire in modo completamente “*automatico*”, in quanto gli strumenti e le tecniche “*manuali*” di analisi non sono in grado di supportare e di gestire l'enorme quantità di dati necessari per l'estrazione.

Cerchiamo ora di concretizzare il concetto di modello, andando a descrivere le principali tecniche utilizzate nell'ambito del DM, che sono classificate in base al tipo di risultato che riescono a ricavare.

Tali tecniche sono essenzialmente di due tipi:

- **Tecniche descrittive:** descrivono i pattern esistenti nei dati, al fine di aiutare chi deve prendere una decisione. Appartengono a questa categoria le regole di associazione e il clustering.
- **Tecniche previsionali:** utilizzano dati in loro possesso, per poter prevedere il valore che la variabile dipendente da tali dati assumerà in futuro. Appartengono a questa categoria gli alberi di decisione e le reti neurali.

Senza perdita di generalità, possiamo dire che il compito fondamentale del DM è quello di applicare algoritmi specializzati per inferire conoscenza. Sarà poi compito del KDD, ed in particolare della fase di interpretazione-valutazione del processo, interpretare e presentare i modelli estratti nella fase di DM.

Il DM condensa in sé molte caratteristiche derivate da altre discipline quali l'apprendimento automatico, la statistica e il pattern recognition, ed eredita da questi ultimi in particolare i concetti di:

- **Classificazione:** studia l'appartenenza di ogni tupla di un insieme di dati ad un gruppo di classi o categorie ben definite, supponendo di avere come punto di partenza un insieme di dati con le rispettive categorie, chiamato **training**

set. Il problema è quindi quello di costruire un modello che possa essere applicato a dati non classificati.

- **Clustering (Unsupervised learning):** raggruppa un insieme di oggetti, fisici o astratti, in classi di oggetti simili. Un cluster non è altro che una collezione di dati che sono simili ad altri nello stesso cluster, e dissimili da oggetti di altri cluster.
- **Regole di associazione:** individuano relazioni interessanti e regolarità tra i termini di un determinato insieme di dati.

Questi concetti verranno approfonditi nei paragrafi successivi. Verrà inoltre maggiormente approfondita la parte riguardante la classificazione, illustrandone non solo i concetti di base, ma fornendo anche alcuni esempi specifici di classificatori e di tecniche per valutare la bontà del classificatore appena estratto. Tale approfondimento è necessario, al fine di introdurre alcuni concetti che verranno utilizzati nel corso dei prossimi capitoli.

Tenteremo ora di definire le caratteristiche comuni a tutti gli algoritmi di DM, per poi descrivere le principali tecniche utilizzate nell'ambito del DM.

Tali caratteristiche sono:

- **Rappresentazione del modello:** è lo strumento utilizzato per descrivere i modelli estratti. È necessario stabilire quale deve essere il linguaggio di rappresentazione dei modelli estraibili. Tale scelta è condizionata da due fattori in contrasto tra loro:
 - Il linguaggio non deve essere troppo complesso, per non rendere incomprensibile la fase di interpretazione.

- Il linguaggio non deve essere troppo riduttivo, in modo da non limitare il potere espressivo del metodo, in riferimento a determinate caratteristiche dei dati.

Esempi di rappresentazione sono gli alberi di decisione e le regole di associazione.

- **Valutazione del modello:** stima quanto un particolare modello concorda con i criteri di valutazione scelti nel processo KDD.
- **Metodo di ricerca:** è costituito a sua volta da due componenti, in parte legate alle caratteristiche sopra descritte:
 - Ricerca del parametro: individua i parametri che ottimizzano il modello di valutazione rispetto ai dati e al modello specificati.
 - Ricerca del modello: cicla sul metodo di ricerca del parametro, ed opera considerando ciascun modello di una determinata famiglia, e valutandolo.

Il DM prevede un'infinità di metodi per estrarre i modelli, ma al fine di garantire maggior chiarezza e semplicità, nel seguito verranno affrontati soltanto i più noti, che dovrebbero comunque fornire sufficiente supporto per i restanti capitoli della tesi.

1.4.1 REGOLE DI ASSOCIAZIONE

Le **regole di associazione** (RdA) sono un modello proposto da [AIS93], il cui compito è quello di scoprire alcune regolarità sui dati. Informalmente possiamo definire una regola di associazione come un legame di causalità valido tra gli attributi dei record di un database.

Una definizione formale proposta da Agrawal, Imielinsky e Swami in [AIS93] per esprimere le regole di associazione è la seguente:

“ Una **regola di associazione** (*RdA*) è un’implicazione del tipo $X \Rightarrow Y$, con $X \subseteq I$, $Y \subseteq I$, $X \cap Y = \emptyset$, dove I è l’insieme degli **item**, X è detto **body** e Y **head** della regola”.

Non tutte le regole che possono essere estratte, ad esempio da un database di transazioni, sono interessanti. Una regola, infatti, per essere considerata **forte**, deve soddisfare due parametri definiti dall’utente, che sono il *minimo supporto* e la *minima confidenza*. Inoltre un insieme di item X (itemset) viene considerato **frequente** se soddisfa il minimo supporto.

A tal fine, diciamo che, definendo D come l’insieme delle transazioni prese in esame per estrarre le regole, dove ogni transazione T è un sottoinsieme di item, tali che $T \subseteq I$, ogni regola è corredata da misure di:

- **Supporto**: indica la frequenza con cui gli item delle regole si presentano nelle transazioni, rispetto al totale delle transazioni esaminate. Un alto valore significa che la regola costituisce una parte considerevole del database. Si dice che la regola di associazione ha un **supporto** s su D se l’ $s\%$ delle transazioni in D contengono sia X che Y , e si calcola:

$$\text{supporto}(X \Rightarrow Y) = \frac{\#\{T \mid X \cup Y \subseteq T\}}{\#D} = \text{supporto}(X \cup Y)$$

- **Confidenza**: misura la forza dell’implicazione, rappresenta il rapporto tra le transazioni in cui sono presenti sia X che Y , rispetto a quelle in cui è presente solo X , quindi rappresenta una stima della probabilità condizionata, in quanto esprime una misura della validità dell’implicazione logica.

Si dice che la regola di associazione ha una **confidenza** c se il $c\%$ delle transazioni in D che contengono X contengono anche Y , e si calcola:

$$\text{confidenza} (X \Rightarrow Y) = \text{supporto} (X \cup Y) / \text{supporto} (X)$$

ESEMPIO

Un tipico esempio di applicazione di RdA riguarda l'analisi effettuata sui prodotti venduti all'interno di un supermercato, detta **Market Basket Analysis (MBA)** [BL97]. Tale tecnica viene utilizzata per individuare le correlazioni tra i vari prodotti che un cliente acquista, attraverso l'analisi dei dati contenuti negli scontrini di cassa. Le informazioni risultanti possono essere utili al settore marketing nel progettare sconti o promozioni particolari, oppure nella progettazione di una diversa disposizione dei prodotti in posizioni più strategiche, in base ai gusti dei clienti.

L'analisi MBA non è relegata al solo contesto di un supermercato, ma trova applicazione anche in molti altri settori, come quello delle telecomunicazioni, la meteorologia, le analisi mediche, etc.

Consideriamo ora l'insieme di item $I = \{A,B,C,D,E,F\}$, e l'insieme delle transazioni riportate in Tabella 1.1. Fissiamo inoltre le soglie per il supporto e la confidenza minime a 0.50 (50%).

Identificatore transazione	Item acquistati
1	A,B,C
2	A,C
3	A,D
4	B,E,F

Tabella 1.1 Gruppo di transazioni di acquisto.

Si ottiene quindi:

- Supporto ($\{A\}$) = 0.75, quindi $\{A\}$ è un itemset frequente
- Supporto ($\{A,B\}$) = 0.25, quindi $\{A,B\}$ non è un itemset frequente

- Supporto ($A \Rightarrow C$) = Supporto ($\{A,C\}$) = 0.50
- Confidenza ($A \Rightarrow C$) = Supporto ($\{A,C\}$) / Supporto ($\{A\}$) = 0.66

Quindi la regola $A \Rightarrow C$ è una regola forte, perché soddisfa i requisiti minimi di supporto e confidenza.

L'insieme degli itemset frequenti è riportato in Tabella 1.2.

Itemset frequenti	Supporto
{A}	0.75
{B}	0.50
{C}	0.50
{A,C}	0.50

Tabella 1.2 Itemset frequenti relativi alle transazioni di Tabella 1.1.

ALGORITMI PER L'ESTRAZIONE DI RDA

L'estrazione delle regole di associazione è fortemente basata sui concetti espressi nel paragrafo precedente, ed avviene principalmente in due passi:

1. **Ricerca di tutti gli itemset frequenti**, in base ad un supporto minimo prefissato.
2. **Generazione di regole di associazione forti dall' itemset frequente**, individuato al passo 1.

La maggior parte degli algoritmi proposti in letteratura implementano questi due passi. Il primo passo, dal punto di vista computazionale, è indubbiamente il più oneroso, e determina le prestazioni degli algoritmi.

L'algoritmo più noto in letteratura è *Apriori* [AIS93], il cui nome deriva dal fatto che sfrutta la conoscenza a priori delle proprietà degli insiemi frequenti.

L'algoritmo si basa su un processo iterativo che, partendo dal calcolo degli itemset frequenti di cardinalità 1 (1-itemset), procede calcolando gli itemset

frequenti con cardinalità 2 (2-itemset) e così via. Il processo si arresta solo quando l' n -itemset è vuoto per un certo n .

Una proprietà importante a sostegno dell'algoritmo Apriori è la seguente:

“Dato B un insieme di item, se B è frequente e $A \subseteq B$, allora anche A è frequente”.

Infatti, ogni transazione che contiene B contiene anche A . Una conseguenza diretta di questa proprietà è che se A non è frequente, è inutile generare gli itemset che contengono A . Ad esempio, riprendendo l'esempio della Tabella 1.1, con un supporto minimo di 0.3, l'itemset $\{D\}$ non è frequente, ed è quindi inutile controllare i 2-itemset o i 3-itemset che contengono l'item D .

Durante il processo, di conseguenza, l'algoritmo utilizza la proprietà precedente per effettuare delle *potature* (pruning) che, ad ogni passo, riducono le dimensioni degli itemset candidati.

La ricerca attuale ha prodotto molte varianti per migliorare l'efficienza dell'Apriori, attraverso tecniche basate su tabelle *hash* [PCY95], tecniche per ridurre la scansione del database, oppure il numero delle transazioni scandite nelle iterazioni future [AS94], [HF95], [PCY95] e tecniche basate sul partizionamento per trovare gli itemset candidati [SON95].

CLASSIFICAZIONE DELLE REGOLE DI ASSOCIAZIONE

Le regole di associazione possono essere classificate in vari modi in base a diversi fattori [HK00]:

- **In funzione del tipo di valori trattati nella regola:**

- (a) **RdA booleane:** il loro valore rappresenta la presenza oppure l'assenza di un item.

- (b) **RdA quantitative:** sono utilizzate per attributi con valore discreto. Invece di verificare solo la presenza di un item, in questo caso è

necessario verificare anche se l'item appartiene ad un intervallo discreto di valori. Ad esempio:

$$\begin{aligned} età(X, "30..34") \text{ and } reddito(X, "42M..48M") &\Rightarrow \\ compra(X, "TV ad alta risoluzione") \end{aligned}$$

- **In funzione della dimensione dei dati trattati:** una RdA può essere *a singola dimensione* o *multidimensionale*, a seconda che ogni item o attributo contenuto all'interno della regola si riferisca o meno ad una singola dimensione. Ad esempio la regola seguente, si riferisce alla sola dimensione *compra*:

$$compra(X, "Computer") \Rightarrow compra(X, "Software")$$

- **In funzione del livello di astrazione:** una RdA può essere considerata a livello *singolo* o *multilivello*, se l'analisi avviene o meno con un solo livello di astrazione. Nel secondo caso, un attributo può essere specializzato o generalizzato in accordo ad una gerarchia di concetti. Ad esempio, se *stampante* rappresenta un'astrazione di livello più alto rispetto a *stampante a colori*, le possibili regole di associazione estratte a due livelli differenti sono:

$$\begin{aligned} età(X, "30-34") &\Rightarrow compra(X, "stampante") \\ età(X, "30-34") &\Rightarrow compra(X, "stampante a colori") \end{aligned}$$

1.4.2 CLASSIFICAZIONE

Come abbiamo osservato precedentemente, la classificazione consiste nell'esaminare il comportamento di un nuovo oggetto di interesse e di assegnarlo ad una classe predefinita. Si tratta di una costante umana, infatti regolarmente noi classifichiamo e cataloghiamo tutto ciò con cui entriamo in contatto, da oggetti inanimati a specie di animali, etc. Il problema della classificazione, quindi, si riconduce all'identificazione di schemi o insiemi di caratteristiche che determinano il gruppo di appartenenza di un dato elemento non ancora classificato, ovvero privo di classe di appartenenza. Legato al concetto di classificazione vi è anche quello di

predizione. La distinzione tra i due concetti riguarda la tipologia di oggetti da classificare, nel primo caso dobbiamo assegnare classi di appartenenza ai vari oggetti etichettandoli, mentre nel secondo caso ci occupiamo di predire valori continui. Anche le tecniche per attuare i due processi sono diverse, per la predizione si utilizzano principalmente tecniche di regressione, mentre nella classificazione uno dei metodi più diffusi è l'albero di decisione.

La classificazione prevede l'utilizzo di insiemi esistenti e già classificati a priori, come ad esempio i clienti perduti da un'azienda commerciale, e porta alla definizione di alcuni schemi su tali insiemi. Si possono quindi scoprire le caratteristiche dei clienti, apparentemente fedeli, che invece non acquisteranno più un dato prodotto ed identificare le promozioni che sono state efficaci.

Applicazioni tipiche vengono riscontrate nel campo del marketing, per classificare clienti in base ai loro gusti, oppure nelle diagnosi mediche, per la classificazione delle malattie in base ai sintomi riscontrati.

La classificazione è un classico problema estensivamente studiato in statistica, apprendimento automatico e reti neurali.

Anche per quanto riguarda la classificazione possiamo distinguere due fasi:

1. **Costruzione del modello:** ha il compito di estrarre il modello vero e proprio. A tal fine, è necessario innanzitutto scegliere l'attributo dei dati che si desidera classificare, il cui valore deve essere stimato. Questo attributo è detto **attributo di classificazione**.

Successivamente, occorre disporre di un insieme campione, detto **training set**, costituito da esempi già classificati, nel quale ogni tupla è già stata osservata e classificata, in modo da contenere già un valore ben preciso per l'attributo di classificazione. Per questa caratteristica del training set, questo passo è anche conosciuto come *Supervised learning*. Il modello costruito può assumere varie forme, ad esempio regole di classificazione, alberi di decisione, reti neurali oppure formule matematiche.

2. **Utilizzo del modello:** una volta costruito il modello, lo si può applicare ad un nuovo insieme di tuple di cui ignoriamo l'attributo di classificazione. La Figura 1.5 riassume questi due passi del processo.

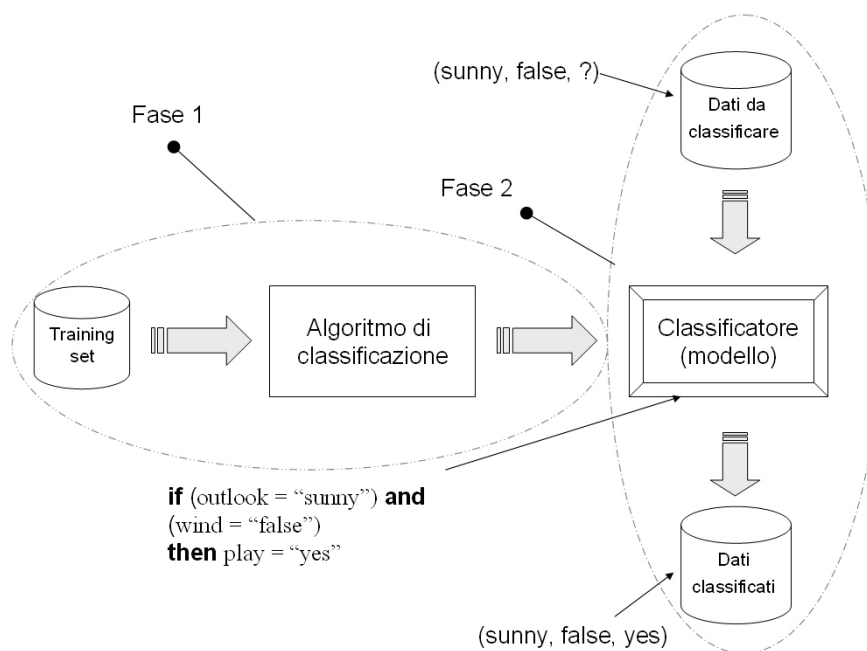


Figura 1.5 Fasi tipiche della classificazione.

Nel seguito faremo riferimento solo alla classificazione, approfondendo alcuni argomenti che risulteranno utili nei capitoli successivi.

ALBERI DI CLASSIFICAZIONE

Gli **alberi di classificazione** sono il modello fondamentale a cui si riferisce la classificazione. I nodi dell'albero sono etichettati con il nome degli attributi, gli archi rappresentano i possibili valori dell'attributo del nodo padre, mentre le foglie rappresentano i differenti valori, assunti dall'attributo da classificare.

Grazie a questa soluzione, un oggetto può essere facilmente classificato, seguendo lungo l'albero il cammino che dalla radice porta alle foglie, percorrendo ogni volta il giusto arco, grazie al valore ed all'attributo discriminato dal nodo in cui ci troviamo.

Gli alberi di classificazione tradizionali presentano difficoltà nella gestione di

attributi con valori di tipo continuo, come età e stipendio, che per necessità devono essere espressi sotto forma di intervalli discreti.

Alcuni sistemi sono tuttavia in grado di manipolare dati di tipo continuo, ed in questo caso l'albero di classificazione risultante viene detto **albero di regressione**.

Un caso particolare di albero si ha quando l'attributo di classificazione assume soltanto due valori, "yes" e "no". In questo caso, il modello estratto prende il nome di **albero di decisione**.

ESEMPIO DI ALBERO DI DECISIONE

Come abbiamo appena visto, il primo input da fornire al sistema per costruire il modello riguarda il training set. In questo caso utilizziamo per tale scopo quello riportato in Tabella 1.3, da cui vogliamo estrarre un albero di decisione che permetta di classificare nuove tuple relative ai potenziali clienti che acquisteranno un computer. L'attributo di classificazione in questo caso è *Compra_computer*, che può assumere soltanto due valori possibili.

Età	Reddito	Studente	Stima del credito	Compra_computer
<=30	Alto	No	Discreto	No
<=30	Alto	No	Eccellente	No
31..40	Alto	No	Discreto	SI
>40	Medio	No	Discreto	SI
>40	Basso	SI	Discreto	SI
>40	Basso	SI	Eccellente	No
31..40	Basso	SI	Eccellente	SI
<=30	Medio	No	Discreto	No
<=30	Basso	SI	Discreto	SI
>40	Medio	SI	Discreto	SI
<=30	Medio	SI	Eccellente	SI
31..40	Medio	No	Eccellente	SI
31..40	Alto	SI	Discreto	SI
>40	Medio	No	Eccellente	No

Tabella 1.3 Training set per l'estrazione di un albero di decisione.

L'albero costruito consentirà di stabilire se un nuovo cliente sarà un potenziale

acquirente di computer o meno. Il classificatore ottenuto è mostrato in Figura 1.6, e come si può notare, le foglie contengono solo uno dei possibili valori dell'attributo di classificazione (*si*, *no*).

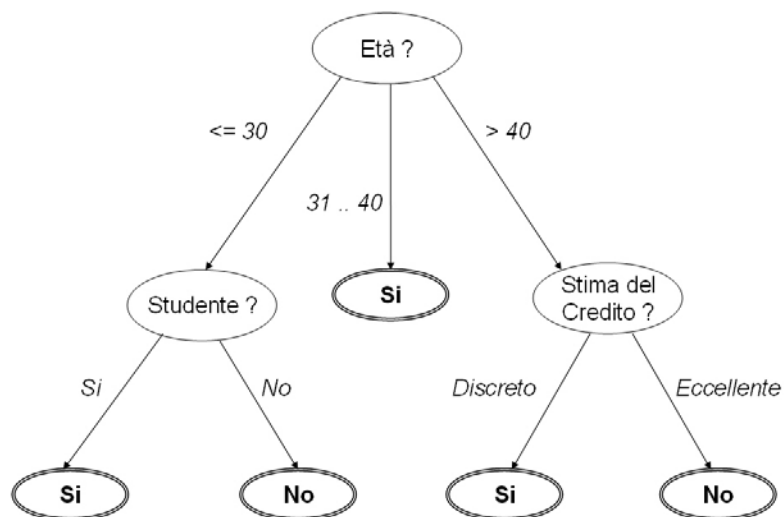


Figura 1.6 Albero di decisione estratto dal training set di Tabella 1.3.

Ogni nuovo cliente verrà quindi classificato in primo luogo in base all'età, e successivamente al fatto che sia studente o dalla stima del credito. Il valore di classificazione da assegnare all'attributo *Compra_computer* sarà quello della foglia raggiunta.

Quando un albero di decisione viene costruito, può accadere che alcuni dei suoi archi riflettano anomalie contenute nel training set, in quanto i dati utilizzati possono contenere valori non attendibili, dovuti ad imperizie o errori nella fase di raccolta o pre-classificazione. In queste circostanze, e più in generale per ridurre la complessità dell'albero ottenuto, si provvede ad utilizzare la tecnica della potatura (pruning), che si occupa di tagliare i rami che, grazie a metodi statistici, sono stati ritenuti non validi o poco importanti.

La costruzione di un albero di classificazione avviene attraverso un procedimento ricorsivo, in cui ad ogni passo ci si basa su euristiche o misure statistiche, per determinare l'attributo da inserire in un nodo.

Algorithm Generate_decision_tree

/* Genera un albero di decisione da un dato training set*/

Input: *samples, attribute_list*;

/* Rappresentano rispettivamente il training set e la lista di attributi significativi, su cui costruire l'albero. In questa versione dell'algoritmo si suppone che tutti i sample siano rappresentati da attributi a valori discreti */

Output: *decision_tree*;

- (1) crea un nodo *N*;
- (2) **if** *samples* sono tutti nella stessa classe *C* **then**
- (3) **return** *N* come foglia etichettata con la classe *C*;
- (4) **if** *attribute_list* è vuoto **then**
- (5) **return** *N* come foglia etichettata con la classe più comune in *samples*; // majority voting
- (6) seleziona *test_attribute* da *attribute_list*; // si sceglie quello con *information gain* maggiore
- (7) etichetta il nodo *N* con *test_attribute*;
- (8) **foreach** valore conosciuto *a_i* di *test_attribute*
- (9) costruisci un arco dal nodo *N* per la condizione *test_attribute* = *a_i*;
- (10) *s_i* = insieme delle tuple in *samples* che soddisfano la condizione *test_attribute* = *a_i*;
- (11) **if** *s_i* è vuoto **then**
- (12) attacca una foglia etichettata con la classe più comune in *samples*;
- (13) **else**
- (14) attacca il nodo ritornato da **Generate_decision_tree** (*s_i*, *attribute_list* - *test_attribute*);

Figura 1.7 Algoritmo *greedy* per la costruzione di un albero di decisione.

Riassumendo, per la costruzione di un albero di classificazione (o di decisione), in generale sono necessarie due fasi:

1. **Fase di costruzione (Tree induction):** l'algoritmo base per la costruzione di un albero di classificazione è un algoritmo di tipo *greedy*, che attraverso ad un approccio *top-down*, ed utilizzando la tecnica divide-et-impera costruisce ricorsivamente il classificatore.

L'algoritmo è mostrato in Figura 1.7, i suoi concetti base sono i seguenti:

- L'albero inizia con un singolo nodo rappresentante il training sample (1).
- Se i *samples* appartengono tutti alla stessa classe, allora il nodo diviene una foglia etichettata con la classe alla quale appartengono i *samples* (2),(3).
- In caso contrario l'algoritmo utilizza una misura basata sull'entropia, *information gain*¹, come euristica, al fine di selezionare l'attributo che

¹ La misura dell'*information gain* è utilizzata per selezionare un *test attribute* ad ogni nodo dell'albero. Tale misura viene generalmente indicata con il nome di *misura di selezione attributo* o

meglio separerà i *samples* in classi individuali (6). Tale attributo diviene il *decision_attribute* o *test_attribute* (7). In questa versione dell'algoritmo, tutti gli attributi sono rappresentati da valori discreti, in caso contrario gli attributi a valori continui richiedono di essere discretizzati.

- Un arco viene creato per ogni valore conosciuto del *test_attribute*, e l'insieme dei *samples* viene partizionato di conseguenza (8),(9),(10).
- L'algoritmo usa lo stesso processo ricorsivamente, per formare un albero di decisione per i *samples* di ogni partizione. Una volta che un attributo occorre ad un nodo, non è necessario reconsiderarlo nei nodi discendenti (13).
- La ricorsione si ferma quando si verifica una delle seguenti condizioni:
 - (a) Tutti i *samples* per un dato nodo appartengono alla stessa classe (2),(3).
 - (b) Non ci sono ulteriori attributi con cui suddividere ulteriormente i *samples* (4). In questo caso per assegnare una classe alla foglia risultante si utilizza la tecnica del **major voting** (5), ovvero la classe che ricorre più frequentemente nei *samples*.
 - (c) Non ci sono *samples* per l'arco etichettato $test_attribute = a_i$ (11). Anche in questo caso viene creata una foglia etichettata con la classe di maggioranza presente nei *samples* (12).

2. **Fase di potatura (Tree pruning):** si pota l'albero costruito al passo 1, eliminando rami dovuti a rumore o fluttuazioni statistiche. I metodi di potatura, quindi, cercano di risolvere problemi di *overfitting* sui dati. In generale possiamo riscontrare più politiche, basate su considerazioni statistiche per attuare questa seconda fase. Le più adottate sono:

misura della bontà dello split. L'attributo con *information gain* più alto (o maggiore riduzione di *entropia*) viene scelto come *test attribute* per il nodo attuale. Questo attributo minimizza l'informazione necessaria per classificare i *sample* nelle partizioni risultanti, e riflette la minore casualità o "impurità" in queste partizioni.

Questo tipo di approccio teorico all'informazione minimizza il numero di test attesi, necessari per classificare un oggetto, e garantisce che venga trovato un albero più semplice (ma non necessariamente il più semplice).

- (a) **prepruning.** Secondo questo approccio, un albero viene potato interrompendo la sua costruzione nella fase iniziale, ovvero decidendo di non dividere o partizionare ulteriormente il subset di training sample ad un dato nodo. Al momento dell'interruzione, il nodo diventa una foglia, che può contenere la classe più frequente tra i subset sample o la distribuzione di probabilità di tali sample.
- (b) **postpruning.** In base a questo metodo i rami vengono tagliati da un albero adulto, ed il nodo di un albero viene potato rimuovendo i suoi rami. Un esempio di approccio postpruning è l'algoritmo di pruning "cost complexity". In questo caso, il nodo più basso non potato diventa una foglia ed è etichettato in base alla classe più frequente tra i rami precedenti. Per ogni nodo "non-foglia" nell'albero, l'algoritmo calcola il tasso di errore atteso che potrebbe verificarsi se il sottoalbero del nodo venisse potato. Inoltre, il tasso di errore atteso, che si otterrebbe, se il nodo non venisse potato, è calcolato utilizzando i tassi di errore per ogni ramo, combinato con il peso secondo la proporzione delle osservazioni su ogni ramo. Se la potatura di un nodo conduce ad un maggior tasso di errore atteso, il sottoalbero viene mantenuto, altrimenti viene potato. Dopo aver generato un set di alberi potati progressivamente, un set test indipendente è utilizzato per stimare l'accuratezza di ogni albero, e viene preferito l'albero di classificazione che minimizza il tasso di errore atteso.

In letteratura sono presenti un gran numero di algoritmi per l'estrazione di alberi di classificazione, la maggior parte dei quali implementano una strategia di tipo top-down, come quella appena vista, ovvero iniziano la costruzione dell'albero a partire dalla radice. Tra gli altri citiamo due algoritmi proposti da Quinlan, ID3, su cui è basato il pseudocodice di Figura 1.7 [Qui86], e la sua derivazione C4.5 [Qui93]. Il primo opera soltanto su attributi nominali, mentre il secondo riesce a trattare anche attributi numerici.

VALUTAZIONE DI UN CLASSIFICATORE

Oltre all'estrazione del classificatore, è importante testare anche la sua qualità [HK00]. Mentre nel caso delle RdA avevamo il supporto e la confidenza che ci fornivano indicazioni precise su tale caratteristica, nel caso dei classificatori dobbiamo utilizzare diverse tecniche per testarne e migliorarne la qualità. Nel seguito introdurremo brevemente le tecniche più utilizzate per valutare la bontà di un classificatore.

MATRICE DI CONFUSIONE E ACCURATEZZA

Una valutazione complessiva delle prestazioni di un classificatore si può ottenere utilizzando la così detta “matrice di confusione”, stimata su un test set.

Si tratta di una matrice che ha un numero di righe e colonne pari al numero di classi che il classificatore deve predire. Le righe della matrice rappresentano la reale classe di appartenenza dei vari oggetti testati, mentre le colonne riguardano la classe assegnata dal classificatore.

Il generico elemento (i,j) contiene un valore che rappresenta la stima della probabilità, calcolata sul test set, che il classificatore assegni un oggetto della classe C_i alla classe C_j , commettendo così un errore. Sulla diagonale troviamo le probabilità di corretta classificazione per le diverse classi.

		Classi Predette					
Classi Reali		Classe 1	Classe 2	...	Classe j	...	Classe N
	Classe 1	23	1		4		0
	Classe 2	34	34		0		2
	
	Classe i	0	2		34		2
	
	Classe N	0	1		2		2

Figura 1.8 Esempio di matrice di confusione.

La Figura 1.8 rappresenta una generica di matrice di confusione. Per avere una stima della precisione del classificatore, basta sommare tutti gli elementi presenti nella diagonale e dividere tale somma per il numero di oggetti che compongono il test set. In questo esempio, in ogni generica cella (i,j) non è presente una stima di probabilità, ma il numero di oggetti del test set della classe C_i classificati come appartenenti alla classe C_j .

Quanto appena detto ci consente di definire un altro parametro legato alla bontà di un classificatore, ovvero l'accuratezza. Quest'ultima ci permette di valutare come un dato classificatore classificherà dati mai visti, ovvero dati sui quali il modello non è stato costruito. Ad esempio, se i dati relativi alle vendite precedenti vengono utilizzati per estrarre un classificatore, al fine di predire il comportamento dei clienti relativamente agli acquisti, vorremmo poter stimare l'accuratezza con la quale il classificatore riesce a prevedere il comportamento dei futuri acquirenti. Grazie a questa misura, è inoltre possibile comparare diversi classificatori.

Nel caso specifico dei classificatori, l'accuratezza è definita come:

“il numero dei campioni correttamente classificati, rispetto al numero totale dei campioni classificati”.

Nei prossimi paragrafi discuteremo in specifico tecniche per valutare e per incrementare l'accuratezza dei classificatori.

METODI PER VALUTARE L'ACCURATEZZA

L'utilizzo di training set per derivare un classificatore e successivamente stimarne l'accuratezza, può dar luogo a valutazioni oltremodo ottimistiche, che possono risultare fuorvianti, a causa dell'alta specializzazione dell'algoritmo di learning (o modello) per i dati. *Holdout* e *cross-validation* sono due tecniche comunemente impiegate per valutare l'accuratezza di un classificatore, basate su partizioni casuali dei dati forniti in ingresso.

HOLDOUT

In questo metodo, i dati forniti al passo di costruzione del classificatore vengono partizionati casualmente in due insiemi indipendenti. Tipicamente, due terzi dei dati vengono allocati nel *training set*, mentre i rimanenti vanno a comporre un nuovo insieme chiamato test set.

Come è facile attendersi, i dati che appartengono al primo insieme vengono utilizzati, come detto in precedenza, per ricavare il classificatore, mentre quelli che appartengono al secondo sono impiegati per valutarne l'accuratezza.

La stima è pessimistica, dato che solo una porzione di dati iniziali è utilizzata per derivare il classificatore. Una possibile variante, chiamata **Random Subsampling** prevede che il metodo *holdout* venga ripetuto k volte. L'accuratezza complessiva viene ottenuta attraverso il calcolo della media delle accuratèzze ottenute ad ogni iterazione.

K-FOLD CROSS-VALIDATION

In base a questo metodo, i dati iniziali sono casualmente partizionati in k sottoinsiemi ("fold"), mutualmente esclusivi, S_1, \dots, S_k , ognuno dei quali ha all'incirca la stessa dimensione. Il training ed il testing vengono eseguiti k volte, con la caratteristica che alla generica iterazione i , con $1 \leq i \leq k$, il sottoinsieme S_i viene utilizzato come test set, mentre i restanti sottoinsiemi sono collettivamente utilizzati come training set. L'accuratezza complessiva è ottenuta come il numero di oggetti correttamente classificati dalle k iterazioni, rispetto al numero totale di oggetti presenti nei dati iniziali.

Nella **stratified cross-validation**, i sottoinsiemi sono partizionati in modo tale che la distribuzione degli oggetti rispetto alla classe da predire in ogni fold, è approssimativamente la stessa riscontrabile nei dati iniziali.

Altri metodi simili alla *k-fold cross-validation* includono il *bootstrapping* ed il *leave-one-out*, in cui vengono applicati diversi criteri di scelta ed utilizzo dei vari sottoinsiemi.

In generale una *stratified 10-fold-cross-validation* è raccomandata per stimare l'accuratezza di un classificatore, anche se la potenza computazionale del sistema permette di usare più sottoinsiemi, a causa del suo errore e varianza relativamente bassi. L'uso di queste tecniche per stimare l'accuratezza di un classificatore aumenta il tempo complessivo di computazione. Tuttavia è indispensabile per scegliere tra vari classificatori.

METODI PER INCREMENTARE L'ACCURATEZZA

Oltre alle tecniche riferite al paragrafo precedente, esistono anche tecniche generali per incrementare l'accuratezza di un classificatore. Queste tecniche sono note come *bagging* (o bootstrap aggregation) e *boosting*.

Entrambe combinano una serie T di classificatori, C_1, \dots, C_T , allo scopo di creare un meta-classificatore o "*improved classifier*" composito.

Per comprendere meglio l'idea alla base di questi metodi, facciamo riferimento ad una situazione reale che ben si adatta ad esemplificarne il meccanismo. Supponiamo di essere un paziente e di voler avere una diagnosi in base ai nostri sintomi. Invece di chiedere un parere ad un solo medico, possiamo scegliere di interrogarne più di uno, al fine di avere un maggior numero di diagnosi. Se una certa diagnosi occorre più di altre, possiamo sceglierla come diagnosi finale per il nostro problema. Questa è l'idea dietro al *bagging*.

Supponiamo invece che noi assegniamo un peso, o importanza ad ogni dottore, ad esempio basandoci sull'accuratezza ottenuta da quel medico nelle diagnosi passate. La diagnosi finale in questo caso sarà ottenuta dalla combinazione di queste diagnosi pesate. Questa è la concezione alla base del *boosting*. Diamo ora uno sguardo più approfondito a queste tecniche.

BAGGING

Dato un insieme S di s oggetti (o sample), il *bagging* opera nel modo seguente: alla t -esima iterazione, con $1 \leq t \leq T$, un training set S_t è campionato con rimpiazzamento dall'insieme originale S . Dato che viene impiegata la tecnica di campionamento con

rimpiazzamento, alcuni oggetti dell'insieme originale S possono non essere inclusi in S_t , mentre altri possono occorrere più di una volta.

Un classificatore C_t è allenato per ogni S_t . Per classificare un oggetto sconosciuto X , ogni classificatore C_t fornisce la sua classe di predizione che conta come un voto. Il classificatore “bagged” C^* conta semplicemente i voti, ed assegna ad X la classe con il maggior numero di voti.

Il *bagging* può essere applicato anche per la predizione di valori continui, prendendo la media dei valori predetti da ogni classificatore.

BOOSTING

In questo caso vengono assegnati dei pesi ad ogni training sample. Una serie di classificatori viene estratta da tali sample. Dopo che un classificatore C_t è stato estratto, i pesi vengono aggiornati per permettere al successivo classificatore, C_{t+1} di “fare maggior attenzione” agli errori di misclassificazione commessi da C_t . Il classificatore “boosted” finale C^* combina i voti di ogni classificatore individuale, dove il peso di ogni voto dato dal classificatore è in funzione della sua accuratezza. Anche in questo caso questo metodo può essere esteso alla predizione di valori continui.

ALTRE MISURE

Purtroppo la sola accuratezza non è sufficiente per riuscire a giudicare la bontà di un classificatore. Vi possono essere infatti altre caratteristiche, in base alle quali scegliere un classificatore invece di un altro, indipendentemente dal suo grado di accuratezza.

Queste caratteristiche includono la velocità di esecuzione del modello, la robustezza, valutata ad esempio dall'accuratezza su noisy data, la scalabilità e l'interpretabilità del modello. Alcune peculiarità che contraddistinguono una misura da un'altra riguardano inoltre la possibilità di ottenere misure accurate, come la scalabilità, o soggettive, come l'interpretabilità. Per la prima, infatti, riusciamo ad ottenere una misura precisa, ad esempio andando ad osservare il numero di

operazioni di I/O invocate da un dato algoritmo su una data macchina, al crescere dei dati in input.

Per la seconda, invece, ci limitiamo a considerazioni che cercano di valutare la complessità del modello ottenuto, ad esempio nel caso degli alberi di classificazione, considerando il numero di nodi che lo compongono.

Oltre a queste misure dobbiamo considerare anche condizioni inerenti a specifiche realtà, e trovare altre misure che si adattino ai casi specifici.

Per approfondire meglio questo discorso, supponiamo di avere estratto un classificatore da dati medici, e che il suo compito sia quello di classificare soggetti in “malati” e “sani”.

Un’accuratezza del 90% sembra garantire che il classificatore estratto sia piuttosto accurato, ma che cosa possiamo dire su di esso, se sappiamo che solo il 4-5% dei soggetti usati come training risulta effettivamente malato? Chiaramente tale accuratezza, seppur elevata, non può risultare accettabile, poiché, ad esempio, il classificatore potrebbe classificare correttamente solo i soggetti sani. Al contrario, il nostro scopo deve essere quello di capire in che misura il classificatore riesca a riconoscere sia i soggetti malati che quelli sani.

Le misure di **sensitivity** e **specificity** servono rispettivamente a questo scopo. Inoltre possiamo integrare queste due misure con la **precision**, in modo da comprendere in che percentuale i soggetti classificati malati sono realmente tali.

$$\text{sensitivity} = t_pos / pos$$

$$\text{specificity} = t_neg / neg$$

$$\text{precision} = t_pos / (t_pos + f_pos)$$

Dove:

- **t_pos** è il numero dei *true positives*, ovvero i soggetti malati correttamente classificati come tali.
- **pos** è il numero totale di soggetti malati.
- **t_neg** è il numero dei *true negatives*, ovvero i soggetti sani correttamente classificati come tali.

- *neg* è il numero totale di soggetti sani.
- *f_pos* è il numero dei *false positives*, ovvero i soggetti sani incorrettamente classificati come malati.

Ci possono essere inoltre altri casi in cui l'accuratezza può risultare inappropriata. Nei problemi di classificazione, si tende comunemente ad assumere che tutti gli oggetti considerati siano unicamente classificabili, ovvero che ogni training sample possa appartenere solo ad una classe. Tuttavia, a causa dell'ampia diversità dei dati memorizzati in grossi database, non è sempre ragionevole assumere che tutti i sample siano univocamente classificabili, mentre è lecito supporre che ogni oggetto possa appartenere a più di una classe.

Tale misura di accuratezza in questo caso non è appropriata, poiché non tiene conto della possibilità che i vari oggetti possano appartenere a più classi contemporaneamente. In questo caso è augurabile che un classificatore, piuttosto che ritornare una classe precisa, ritorni una distribuzione di probabilità sulle classi a cui un determinato sample può appartenere.

Come misura di accuratezza può essere usata un'euristica "second guess", dove la classe predetta è giudicata corretta, se in accordo alla prima o seconda classe più probabile.

Inoltre, per valutare la bontà di un classificatore possono essere usati anche approcci grafici, come il *LIFT Curve*, in cui vengono messi graficamente a confronto i risultati di classificazione ottenuti con l'utilizzo o meno del classificatore.

1.4.3 CLUSTERING

Immaginiamo di avere un insieme di oggetti da analizzare, ma a differenza della classificazione, non conosciamo il nome della classe di ciascun oggetto. Il clustering è una tecnica di analisi dei dati, che raggruppa gli oggetti in base a misure di similitudine, o in base ad un criterio prestabilito.

Il suo obiettivo è massimizzare le similitudini tra oggetti appartenenti ad uno stesso cluster, minimizzando quelle tra elementi appartenenti a cluster differenti.

Tale misura di similitudine è basata sui valori degli attributi che descrivono l'oggetto.

Anche il clustering, come la classificazione, è un'attività insita nella natura umana, basti pensare alla prima infanzia, in cui impariamo ad esempio la distinzione tra animali e piante, ed abbiamo un incremento continuo di schemi subconsci di classificazione.

Come metodo analitico, il clustering è molto utilizzato nel marketing, per raggruppare i clienti in base alle loro similarità nei comportamenti d'acquisto, in biologia al fine di estrarre tassonomie di piante e animali e nel mondo delle assicurazioni allo scopo di identificare gruppi diversi di premi assicurativi.

Ulteriori applicazioni si possono riscontrare nella prevenzione di terremoti, attraverso il raggruppamento degli epicentri lungo le faglie terrestri, o nell'urbanistica, al fine di individuare insiemi di edifici in accordo al loro utilizzo tipico, al valore o alla loro ubicazione geografica.

CONCETTI DI BASE DEL CLUSTERING

Occorre distinguere il clustering dalla classificazione. In parte i due concetti sono già stati distinti nelle prime righe del paragrafo precedente, ma approfondiamo ora il concetto, affermando che il clustering si differenzia dalla classificazione, in quanto non si basa sul calcolo di un singolo attributo, ma lavora su elementi definiti da un certo numero di attributi e li assegna ad un gruppo, detto **cluster**, in base alle loro somiglianze.

Nel clustering, non sono presenti classi predefinite e l'analisi non avviene su un training set di esempio, come succedeva per la classificazione.

Grazie a queste caratteristiche, il clustering è un esempio di *unsupervised learning*.

La qualità di un clustering si misura sempre in base a due parametri:

- Alta similarità tra oggetti appartenenti allo stesso cluster.
- Bassa similarità tra oggetti appartenenti a cluster diversi.

Generalmente un database contiene diversi tipi di variabili, a valori booleani, nominali, continui o discreti. È quindi necessario utilizzare la misura di similarità più consona ad ogni tipo di variabile e combinare insieme i loro effetti, attraverso una formula pesata.

ALGORITMI DI CLUSTERING

Anche in questo caso esistono un gran numero di algoritmi di clustering presenti in letteratura. La scelta dell'algoritmo da applicare dipende sia dal tipo dei dati disponibili, che dalla particolare applicazione.

La maggior parte dei metodi di clustering possono essere classificati nelle seguenti 5 categorie [HK00]:

1. **Metodi di partizionamento:** dato un database di n oggetti, questo metodo costruisce k partizioni di dati, con $k \leq n$, dove ciascuna partizione rappresenta un cluster. Con questo procedimento si classificano i dati in k gruppi, che devono soddisfare le seguenti proprietà:
 - a. ciascun gruppo deve contenere almeno un oggetto.
 - b. ciascun oggetto deve appartenere soltanto ad un gruppo.

Il più importante algoritmo di questa classe è il *k-means* [Mac67], che è basato su metodi euristici, in cui ciascun cluster è rappresentato dal valor medio degli oggetti presenti nel cluster.

2. **Metodi gerarchici:** un algoritmo basato su questo metodo crea una decomposizione gerarchica dell'insieme di oggetti, rappresentante i dati, e si può classificare in *agglomerativo* o *divisivo*, a seconda di come si viene a formare la decomposizione. Nel primo caso la decomposizione inizia con ciascun oggetto che forma un gruppo separato, e successivamente unisce gli

oggetti o gruppi vicini tra loro, mentre nel secondo abbiamo tutti gli oggetti considerati in un unico cluster, ed in seguito divisi in cluster più piccoli.

Tra gli algoritmi più noti di questa classe segnaliamo AGNES, DIANA [KR90], BIRCH [ZLR96] e CURE [GRS98].

3. **Metodi basati sulla densità:** la maggior parte dei metodi di partizionamento raggruppa gli oggetti, basandosi sulla *distanza* tra questi. In questa categoria, invece, il partizionamento avviene grazie alla nozione di densità, ovvero il numero di oggetti presenti all'interno di un dato raggio. In questo caso vengono aggiunti oggetti ai cluster, finché la densità non supera una soglia minima prefissata.

Gli algoritmi di rilievo in questa categoria sono DBSCAN [EKSX96], Optics [ABKS99] e DENCLUE [HK98].

4. **Metodi basati su griglia:** questi metodi codificano lo spazio degli oggetti da analizzare utilizzando un numero finito di celle, che formano una struttura a griglia. Tutte le operazioni di clustering sono eseguite all'interno di tale struttura. Il principale vantaggio di tale approccio è dato dalla velocità di esecuzione che non è dipendente dal numero di oggetti, ma dal numero di celle, in cui ogni dimensione dello spazio è quantificata. Tra gli algoritmi che adottano tale procedimento ricordiamo STING [WYM97], CLIQUE [AGGR98] e WaveCluster [SCZ98].

5. **Metodi basati su modello:** cercano di rappresentare i dati secondo un modello di tipo matematico, basandosi sull'assunzione che i dati sono generati secondo distribuzioni probabilistiche ben precise. Un algoritmo di questo tipo può collocare i cluster costruendosi una funzione di densità, che riflette la distribuzione spaziale degli oggetti. Soluzioni che adottano tale procedimento sono COBWEB [Fis87] e CLASSIT [GLF89].

1.4.4 ASPETTI TEMPORALI NEL DATA MINING

Fino ad ora nel trattare le varie problematiche di DM, non ci siamo mai confrontati con un elemento che è chiave nella maggior parte delle azioni che svolgiamo giornalmente, ovvero il tempo.

Il tempo è un concetto importante nell'ambito delle basi di dati; i database infatti dispongono di metodi opportuni per operare sul concetto di tempo, in modo da poter risolvere compiti quali:

“Selezionare gli articoli venduti da almeno una settimana”

“Selezionare i ricavi per giorno della settimana relativamente al mese corrente”

Anche all'interno del DM è importante utilizzare modelli che prevedono il fattore tempo. I modelli proposti in letteratura sono delle varianti dei modelli attuali, ed in parte precedentemente descritti, che permettono di esprimere la dimensione temporale.

Senza dilungarci troppo su tali metodi, e rimandando alla letteratura specializzata, possiamo affermare che i principali modelli temporali comprendono:

- Le **RdA cicliche** e le **RdA calendriche**: [ORS98] sono due esempi di estensione del modello delle RdA, in grado di trattare il fattore tempo. Rispetto a quelle precedentemente introdotte, che consideravano le transazioni come un unico grande contenitore, senza valutare possibili segmentazioni dipendenti dal tempo, queste contengono una “marca temporale” (**timestamp**), che indica quando la transazione è stata effettuata.
- I **Pattern sequenziali**: sono un modello di DM che permette di estrarre tutte le sottosequenze da una sequenza di transazioni, il cui supporto è maggiore di una soglia minima specificata dall'utente. Ad esempio, l'informazione estratta può assumere la forma seguente: “il 95% dei clienti ha acquistato prima il prodotto X, e poi il prodotto Y”. Nell'ambito dei pattern sequenziali, le transazioni sono corredate da una data, oppure un valore progressivo che

specifica l'ordine della transazione, e si definiscono **sequenze** gli insiemi delle transazioni associate ad ogni cliente ordinate nel tempo [AS94].

- **Serie Temporal**i: rappresentano una sequenza di osservazioni nel tempo [Cha03]. Ad esempio, fissata come data iniziale il primo gennaio, e stabilito come unità di tempo il giorno, la seguente è una serie temporale che indica le temperature registrate in una data località tutti i giorni della prima settimana dell'anno:

$$\langle -3^{\circ}, -1^{\circ}, 0^{\circ}, 0^{\circ}, -1^{\circ}, ?, -1^{\circ} \rangle$$

1.5 IL FUTURO DEL KDD

Come abbiamo potuto notare, sia il DM che il KDD in generale sono strumenti divenuti ormai necessari, per poter sfruttare appieno le nostre capacità di immagazzinare dati. Senza di essi questa mole di dati risulterebbe scarsamente utile. Purtroppo la relativa gioventù, la natura del processo ed il vasto campo applicativo li rendono scarsamente standardizzabili. Alcuni problemi, come la rappresentazione dei dati, la complessità della ricerca dei modelli e l'uso a priori di domini di conoscenza sono questioni tuttora aperte.

A queste problematiche principalmente teoriche, ne seguono inoltre alcune di tipo applicativo e relative alla ricerca nel KDD. Esse riguardano principalmente la dimensionalità e l'*overfitting* sui dati, i valori mancanti o il rumore, ed in genere i dati incompleti, le misure di *interestingness* dei pattern scoperti, le questioni riguardanti l'interazione uomo-macchina, i problemi di gestione della privacy e l'integrazione con altri strumenti etc.

In questi ultimi anni il processo KDD è stato quindi oggetto di numerose ricerche, che hanno portato da un lato alla definizione delle sue fasi, e dall'altro allo sviluppo di strumenti specifici per ogni fase del processo. Questo ha condotto allo sviluppo di software in grado di trattare solo singole fasi del processo KDD. In modo analogo, anche per la fase di DM, i vari modelli erano rimasti isolati, ed ogni singolo tool veniva ottimizzato solo per uno specifico modello dei dati. La situazione era

paragonabile a quella dei primi anni '60 nel mondo dei DBMS, in quanto non esistevano primitive ad alto livello dedicate a trattare ed interrogare grandi quantità di dati. Queste primitive sono state offerte successivamente da SQL².

Negli ultimi anni, la ricerca si è quindi spostata verso lo studio di un **KDDMS** (KDD Management System), in grado di fornire un ambiente adatto a supportare tutte le fasi del processo e che, nello stesso tempo, permette di utilizzare e combinare i diversi modelli di DM.

Rispetto ad un DBMS, un KDDMS deve poter trattare oggetti molto più complessi ed eterogenei. Di conseguenza, anche le query sono più complesse, e devono poter combinare operatori ed algoritmi che si riferiscono a modelli di dati differenti. In questa direzione, è quindi necessario implementare un query language (QL) dotato di un forte potere espressivo.

Da questo punto vista in realtà le strade intraprese sono essenzialmente due:

1. Estensione dei QL esistenti, con costrutti specifici attraverso i quali esprimere problemi di mining.
2. Definizione di ambienti ad hoc in cui esprimere l'intero processo.

In entrambi i casi, bisogna osservare che un buon ambiente deve essere molto elastico, a causa dell'incessante progresso del settore KDD e della continua scoperta di nuovi algoritmi di DM.

Al momento attuale non esiste ancora un QL stabile per KDDMS che giochi lo stesso ruolo di SQL per quanto riguarda i DBMS, ma la ricerca ha comunque prodotto numerose proposte che aspirano a diventare una base sicura per l'obiettivo richiesto.

Un esempio di estensione di QL è rappresentato da MINE RULE, che fornisce un primo lavoro in tale direzione. Il modello presentato da Meo, Psaila e Ceri [MPC96]

² Structured Query Language, linguaggio standard di interrogazione dei database relazionali, che permette la consultazione e la modifica dei dati. La dicitura "di interrogazione", che si usa di solito, è piuttosto riduttiva, dal momento che esso contiene, oltre ai costrutti di interrogazione (QML - Query Manipulation Language), anche quelli di definizione dei dati (DDL - Data Definition Language), e quelli per la loro manipolazione (DML - Data Manipulation Language).

permette una descrizione uniforme del problema di estrazione di regole di associazione. Il modello presenta un operatore chiamato MINE RULE, che estende il linguaggio SQL, e si pone come obiettivo quello di catturare tutti i problemi di mining su regole di associazione proposti in letteratura, e di formularne di nuovi.

In letteratura sono presenti comunque molteplici strumenti per il supporto al DM ed al KD, come rilevano i siti specializzati [KDN].

Il processo KDD è concepito come un sistema, e rappresenta il risultato di un continuum di attività che appartengono a discipline anche molto diverse tra loro, e come tale richiede che diverse figure professionali siano implicate nei vari step del processo. Questa concezione richiede più ricerca rivolta alla corretta e coerente integrazione dei risultati ottenuti nelle varie discipline coinvolte nel processo.

Altre problematiche di ricerca riguardano caratteristiche nuove, legate principalmente al processo visto nella sua totalità, come ad esempio il problema delle performance del sistema. L'utente richiede una risposta dal processo in tempi accettabili, caratteristica particolarmente importante, quando si ha a che fare o con database di dimensione rilevanti, o con tecniche algoritmiche piuttosto complesse, utilizzate nelle varie fasi del processo.

Il secondo problema riguarda come assistere l'utente nella selezione e nel matching degli strumenti e delle tecniche adatte ai propri scopi. Nel KDD questo è un problema particolarmente importante da risolvere, anche nel caso in cui l'utente sia un ricercatore che ha sviluppato delle tecniche specifiche, in quanto è l'intero sistema che è chiamato alla risoluzione di un problema.

Mentre ci sono degli sviluppi nelle Knowledge Discovery Workbenches e nei sistemi integrati, che includono più di una fase del processo KDD, allo stato attuale non si è ancora riusciti a risolvere completamente il problema di un sistema user friendly che possa essere utilizzato da un analista. L'analista, infatti, ha il compito di interpretare correttamente i dati disponibili, utilizzando le tecniche del KDD, ma non è necessariamente un esperto di knowledge discovery. È necessario che un sistema KDD si integri nell'ambiente esistente, per fornire soluzioni adeguate all'analista. Pertanto, una grossa sfida a questo proposito sta nell'enfatizzare sia il processo KDD nella sua totalità, che gli strumenti di supporto alle sue varie fasi.

Dovrebbe essere dato maggior risalto all'interazione uomo-macchina, piuttosto che ad una totale automatizzazione del processo, allo scopo di fornire supporto sia ad utenti esperti che a principianti. Il tipo di interazione uomo-macchina prevista dal processo KDD permette, come definito da Uthurysamy in [FPSSU96], sia la “human-assisted computer discovery” che la “computer-assisted human discovery”. Lo sviluppo di appropriati strumenti per la visualizzazione, l'interpretazione e l'analisi dei pattern scoperti è di grande rilevanza, in quanto una tale interazione potrebbe fornire una soluzione pratica a molti problemi quotidiani in maniera più rapida, rispetto all'analisi dei dati effettuata separatamente da un utente o da un computer, riducendo notevolmente i tempi per l'interpretazione dei dati complessi.

Concludendo, la nostra ricerca dovrebbe focalizzare gli sforzi futuri nello sviluppo di strumenti e tecniche atte alla risoluzione di problematiche relative alla qualità e quantità dei dati da immagazzinare ed alla loro interpretazione. Inoltre, dovrebbe determinare come svolgere un compito di analisi dei dati che richiede da parte dell'utente caratteristiche di giudizio, conoscenza ed interazione.

Capitolo 2

KDDML

2.1 INTRODUZIONE

Nel contesto di ricerca del KDD, uno dei progetti più ambiziosi e degni di nota riguarda la definizione di un QL esteso a dare supporto all'intero processo KDD. L'input per la definizione di un QL esteso con funzionalità di KDD è stato fornito da Imielinski e Mannila [IM96], i quali pongono le basi per trattare il KDD come un processo di query, e dettano le condizioni che dovrebbe soddisfare un QL in grado di supportarlo. Alla base di queste condizioni domina il **principio di chiusura delle query**, secondo il quale è possibile annidare all'interno della stessa query le diverse invocazioni degli algoritmi di DM e degli operatori utilizzati, ovviamente nel caso in cui il tipo della sottoquery sia compatibile con il tipo dell'operatore principale.

Sulla base delle osservazioni contenute in questo articolo, presso il Dipartimento di Informatica dell'Università di Pisa è stato sviluppato un ambiente di sviluppo per applicazioni di estrazione di conoscenza da database. Tale sistema, conosciuto con la sigla **KDDML** (Knowledge Discovery in Databases Markup Language) [RRT05], ha avuto come obiettivo principale quello di consentire un alto livello di interoperabilità tra i diversi metodi di DM e di rappresentazione della conoscenza. Questo scopo è stato raggiunto attraverso la definizione di un modello capace di descrivere in modo uniforme le varie problematiche di mining.

Nel seguito del capitolo introdurremo innanzitutto uno strumento chiave su cui è basato KDDML, ed un formato standard per la rappresentazione dei modelli, che risulterà utile sia nel contesto di KDDML, sia nel capitolo successivo come caratteristica interessante da valutare.

2.2 XML

Negli ultimi anni la crescita esponenziale del web e dell'interesse attorno ad esso ha portato alla necessità di sviluppare markup language più flessibili del noto HTML. Lo scopo di quest'ultimo è la visualizzazione dei dati e, per questo motivo prende in considerazione soltanto concetti riguardanti il modo in cui le informazioni vengono mostrate, mentre altre proprietà come il tipo o la struttura di tali informazioni non vengono prese in considerazione.

L'esigenza di avere a disposizione un formalismo più flessibile di HTML, attraverso il quale poter esprimere anche la struttura logica delle informazioni a nostra disposizione, ha portato alla definizione di un sottoinsieme dello Standard Generalized Markup Language (SGML), ormai noto come eXtensible Markup Language (XML) [Har01].

XML è un metalinguaggio di markup aperto e basato su testo, che fornisce informazioni di tipo strutturale e semantico sui dati e permette quindi di rappresentare documenti e facilitare lo scambio dei dati sul web. Si caratterizza per la semplicità con cui è possibile scrivere documenti, dividerli e trasmetterli nel web.

A differenza dell'HTML, XML non fissa a priori un insieme di tag, ma l'utente è libero di definirsi i propri tag in maniera arbitraria. Lo scopo di questo paragrafo, tuttavia, non è quello di fornire una guida completa ed esaustiva al linguaggio XML, ma solo quello di introdurre alcune caratteristiche che risulteranno utili per comprendere meglio i concetti che verranno espressi, quando andremo a trattare il sistema KDDML. Nel prossimo paragrafo daremo pertanto solo una breve descrizione della struttura di un tipico documento XML.

2.2.1 DOCUMENTI XML

Ogni documento XML è costituito da una serie di oggetti, ciascuno composto da uno o più **elementi** che ne rappresentano la componente logica. Tali elementi possono contenere al loro interno altri elementi, testo oppure essere vuoti. Ogni elemento è delimitato da uno **start-tag**, *<nome elemento>* e da un **end-tag** *</nome-elemento>*. Dal punto di vista sintattico, quindi, XML è simile ad HTML, ma presenta una sintassi più rigida e severa. Questa caratteristica è stata introdotta dal gruppo di studio del W3C [W3CXML] per facilitare lo sviluppo delle applicazioni basate su XML.

Come accade in HTML, l'annidamento tra gli elementi è controllato: se uno start-tag è nella portata di un altro elemento, anche l'end-tag deve essere nella portata dello stesso elemento. A differenza di HTML, quindi, tutti i tag devono essere chiusi esplicitamente.

Questo significa che tutti gli elementi racchiusi tra uno start-tag ed il rispettivo end-tag, devono annidarsi regolarmente uno dentro l'altro. Ad un elemento è possibile associare un insieme di **attributi** che lo caratterizzano, rappresentati da un insieme di coppie (*nome, valore*) che occorrono all'interno dello start-tag. A differenza degli elementi, gli attributi non sono strutturati ed il loro valore è sempre atomico.

Al fine di chiarire meglio come si presenta un documento XML, ne introduciamo un semplice esempio. Supponiamo di voler rappresentare le seguenti RdA (par. 1.4.1), fornite nella Tabella 2.1, attraverso un documento XML.

Id	Body	Head	Supporto	Confidenza
1	Pane	Pasta	0.45	0.68
2	Latte	Biscotti, Cereali	0.7	0.98

Tabella 2.1 Insieme di regole di associazione.

La Figura 2.1, mostra la struttura di tale documento. Si possono individuare cinque elementi, *RdA*, *SingleRdA*, *Body*, *Head*, *Item*. Solo l'elemento *Item* è atomico e non contiene nella sua portata nessun altro elemento. L'elemento *SingleRdA* ha associati gli attributi *Id*, *Supporto*, *Confidenza*.

```

<?xml version="1.0" encoding = "UTF-8"?>
<!DOCTYPE RdA SYSTEM "regole.dtd">
<RdA>
  <SingleRdA Id = "1" Supporto = "0.45" Confidenza = "0.68">
    <Body>
      <Item> Pane </Item>
    </Body>
    <Head>
      <Item> Pasta </Item>
    </Head>
  </SingleRdA>
  <SingleRdA Id = "2" Supporto = "0.7" Confidenza = "0.98">
    <Body>
      <Item> Latte </Item>
    </Body>
    <Head>
      <Item> Biscotti </Item>
      <Item> Cereali </Item>
    </Head>
  </SingleRdA>
</RdA>

```

Figura 2.1 Possibile rappresentazione in XML delle RdA di Tabella 2.1.

Allo scopo di poter definire elementi e attributi personalizzati, XML mette a disposizione un modello formale dei dati conosciuto come **Document Type Definition (DTD)**, grazie al quale è possibile descrivere, attraverso espressioni regolari, la struttura di una classe di documenti.

Attraverso le DTD siamo, di fatto, in grado di definire come gli elementi devono essere annidati tra loro, nonché il numero ed il tipo di attributi ad essi associati. Un esempio di DTD per il documento XML di Figura 2.1 è mostrato in Figura 2.2.

```

<?xml version="1.0" encoding = "UTF-8"?>
<!ELEMENT RdA (SingleRda)+ >
<!ELEMENT SingleRda (Body, Head) >
<!ATTLIST SingleRda    Id ID #REQUIRED
                      Supporto CDATA #REQUIRED
                      Confidenza CDATA #REQUIRED >
<!ELEMENT Body (Item)+ >
<!ELEMENT Head (Item)+ >
<!ELEMENT Item (#PCDATA) >

```

Figura 2.2 DTD relative ai dati di Figura 2.1.

Un documento XML si dice **valido**, se rispetta le specifiche definite nella corrispondente DTD. L'esempio di Figura 2.1 costituisce dunque un documento valido rispetto alle DTD di Figura 2.2.

2.2.2 ELABORAZIONE DI DOCUMENTI XML

Uno degli obiettivi richiesti dal W3C durante la definizione del linguaggio è stato quello di facilitare lo sviluppo di programmi che utilizzano ed elaborano documenti XML.

Il *DOM (Document Object Model)* [W3CDOM] è uno standard del W3C che definisce una serie di interfacce per la rappresentazione e la manipolazione di documenti XML. Il DOM fissa un modello ad oggetti, in grado di rappresentare i documenti attraverso una struttura ad albero, al fine di uniformare il modo in cui le applicazioni processano i documenti.

Un'implementazione del DOM, è integrata nelle ultime versioni del JDK sviluppato dalla Sun Microsystems [SUN].

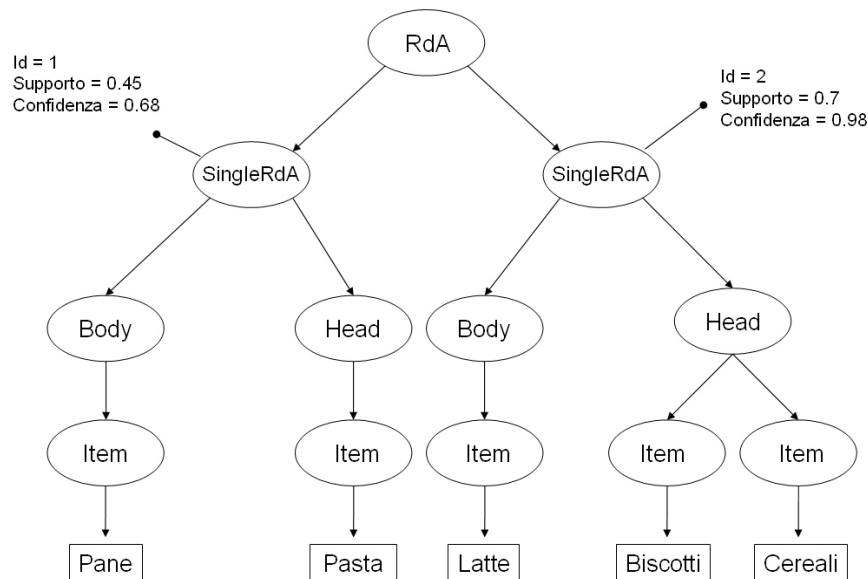


Figura 2.3 Albero DOM del file XML di Figura 2.1.

In Figura 2.3 è rappresentato l'albero DOM per i dati visti in Figura 2.1. L'albero DOM è generato come risultato del processo di parsing del documento.

Un'altra tecnologia proposta dal W3C è l'*Extensible Stylesheet Language (XSL)* [W3CXSL], che include un linguaggio per la creazione di fogli di stile associati ai documenti XML. Attraverso questo meccanismo viene definito come il documento deve venire rappresentato all'interno di un browser. Anche questa libreria è utilizzata nel sistema KDDML, ed il suo compito riguarda la trasformazione di documenti XML, che rappresentano i risultati finali ed intermedi in moduli HTML, al fine di garantirne la loro visualizzazione tramite un browser.

Appare ora evidente la scelta di utilizzare XML all'interno di KDDML. XML, grazie alla possibilità di definire nuovi nomi di tag, permette di garantire non solo un alto grado d'interoperabilità, ma anche di soddisfare le proprietà di espressività e flessibilità richieste dalla specifica di KDDML. Grazie alla flessibilità di XML, è infatti possibile rappresentare oggetti molto complessi ed eterogenei e quindi esprimere con una rappresentazione comune sia i risultati degli operatori del linguaggio, sia l'uso degli operatori stessi.

Al tempo stesso, facilita lo sviluppo di applicazioni che elaborano documenti XML. Attraverso il DOM, ogni documento può essere trattato come un oggetto vero e proprio, e i programmi sono in grado di accedervi dinamicamente per leggere il suo contenuto ed eventualmente modificarlo. Sono previste interfacce per rappresentare ognuno dei markup di XML, che consentono di accedere sia al contenuto degli elementi, sia alla loro struttura. Con riferimento, all'implementazione del DOM contenuta nel JDK 1.4, sono disponibili metodi quali *getFirstChild()*, *getChildren()*, *appendChild()* per modificare i sottoelementi dell'albero, oppure *getAttribute()* e *setAttribute()* per manipolare i valori degli attributi. Grazie all'albero DOM, viene dunque semplificato il compito dell'implementatore che ora dispone di una vista concettuale dei dati.

Queste considerazioni mostrano quali siano i vantaggi dell'utilizzo di XML come linguaggio di markup per la definizione di un ambiente di supporto alla fase di DM.

2.3 PMML

Il **Predictive Model Markup Language (PMML)** [DMG] è un linguaggio basato su XML, che fornisce soluzioni per applicazioni che definiscono modelli statistici di

DM, e per condividere modelli tra le applicazioni basate su PMML.

Sviluppato da DMG (Data Mining Group), PMML garantisce alle applicazioni un metodo “vendor independent” per definire i modelli, in modo che i problemi proprietari e le incompatibilità non siano più una barriera allo scambio di modelli tra le applicazioni. Esso permette agli utenti di sviluppare modelli all’interno di una propria applicazione e di usare altre applicazioni, sviluppate anche da terzi, per visualizzare, analizzare e valutare i modelli. Precedentemente questo era molto difficile, ma con l’introduzione di PMML lo scambio di modelli tra applicazioni avviene in maniera diretta.

Ogni documento PMML presenta una definizione non procedurale di modelli analitici parametrizzati. Il parsing di un documento PMML avviene usando qualsiasi parser standard XML, ed attraverso di esso l’applicazione può determinare il tipo dei dati in input e output utilizzati dal modello, i dettagli sulla forma del modello, e come interpretare i risultati ottenuti in termini di DM.

La struttura dei modelli è descritta attraverso una DTD chiamata PMML-DTD. Un documento PMML per essere valido deve rispettare la PMML-DTD e deve obbedire ad un numero di regole di vario genere descritte da alcuni punti presenti nella specifica PMML.

La prima versione prodotta dal DMG ha fornito un piccolo insieme di DTD che specificava le entità e gli attributi per documentare modelli predittivi (alberi decisionali) e modelli di regressione. Successivamente, il pacchetto è stato esteso introducendo anche regole di associazione, clustering, reti neurali e rappresentazioni sequenziali.

La Figura 2.4 mostra un esempio di file PMML generato da KDDML, per la rappresentazione di un albero di classificazione. Nel paragrafo successivo verranno approfonditi i componenti che costituiscono la struttura tipica di un documento PMML.

Possiamo comunque osservare che al suo interno sono salvate informazioni riguardanti l’applicazione che ha generato il modello `<Application name = ...>`, i meta-dati (`<DataDictionary>`), il nome del modello, il ruolo degli

attributi (active, predicted, ...) nell'estrazione del modello, ed infine il modello effettivo.

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

Figura 2.4 Un esempio di modello PMML.

2.4 KDDML

Ricordiamo che il processo KDD è un lavoro complesso, che dipende fortemente dal problema e dai dati che abbiamo a disposizione. Come descritto nel paragrafo 1.3.1, esso consiste in una serie di fasi, con compiti piuttosto diversi tra loro, e con la caratteristica che il risultato prodotto da una fase è l'input per quella successiva. Chi sviluppa software per KDD ha la necessità di specificare i compiti previsti per ogni fase e definire le interazioni e le dipendenze tra le fasi.

L'obiettivo di KDDML è quello di fornire un linguaggio (ed un sistema) *middleware* per supportare lo sviluppo di applicazioni che necessitano di usufruire di strumenti per l'accesso ai dati, e di algoritmi per il preprocessing ed il DM.

In questo contesto, XML sembra l'anello di congiunzione tra la tecnologia dei database e gli strumenti di DM. Il suo uso nei sistemi esistenti appare limitato allo scambio di dati e modelli di DM tra le applicazioni.

La nostra idea è stata quella di concepire un linguaggio basato su XML, in cui l'uso di quest'ultimo non fosse legato solo allo scambio di dati e modelli, ma rendesse possibile la definizione dell'intero processo di KDD. Nel prossimo paragrafo faremo una breve panoramica sulle caratteristiche più interessanti del KDD Markup Language.

2.4.1 IL LINGUAGGIO

La sintassi di KDDML è XML-based, in cui ogni operatore presente nel linguaggio è modellato attraverso un tag. L'uso di XML favorisce la rappresentazione e la processabilità delle query KDD. La semantica del linguaggio è puramente "funzionale", e ciò assicura allo stesso la composizionalità degli operatori e vede rispettato il principio di chiusura.

Ogni espressione KDDML di seguito verrà chiamata *query*, per mettere in evidenza il fatto che l'attenzione di tutto il processo rappresentato dal file XML è sul risultato.

Data la sua natura puramente funzionale, andiamo a vedere cosa intendiamo per operatore in KDDML. A livello semantico, un operatore non è altro che una funzione definita per mezzo di una *signature*, grazie alla quale vengono specificati il dominio, attraverso il numero ed il tipo dei suoi argomenti, ed il codominio, ovvero il tipo ritornato dall'operatore. Una *signature* è quindi rappresentata nel seguente modo:

$$f_{\langle \text{OPERATOR_NAME} \rangle}: t_1 \times \dots \times t_n \rightarrow t$$

OPERATOR_NAME è una funzione che riceve in input n argomenti, rispettivamente di tipo t_1, t_2, \dots, t_n , e ritorna un argomento di tipo t . Questo operatore può essere usato ogniqualvolta è richiesto un tipo t all'interno della query, soddisfacendo quindi il principio di chiusura.

L'insieme dei tipi tuttora definiti in KDDML sono:

- *table/PPtable*: definiscono le sorgenti dei dati su cui i vari operatori lavorano e rappresentano sia le tabelle relazionali lette direttamente dal repository (*table*), che tabelle, ottenute attraverso operazioni di preprocessing (*PPtable*). In generale un KDDML *table* è caratterizzato da:
 - uno *schema*, che rappresenta la struttura logica dei dati fisici, includenti il numero ed il tipo degli attributi, insieme a semplici statistiche sul valore degli attributi;
 - i *dati fisici*, rappresentati attraverso un file di testo in CSV (comma-separated value);

Nel caso di KDDML *PPtable*, oltre alle già citate informazioni abbiamo

- i *preprocessing data*: rispetto ai *table*, includono informazioni di preprocessing sui dati, come ad esempio un contrassegno per identificare i dati fisici su cui viene attuato tale procedimento. Questo tipo di dato viene utilizzato solo nella fase di preprocessing, per isolarla dalle altre. Il linguaggio mette a disposizione gli operatori `<TABLE_2_PP_TABLE>` e `<PP_TABLE_2_TABLE>` necessari per trasformare i dati da *table* a *PPtable* e viceversa³.
- *tree, rda, sequences, clusters, hierarchy*: vi è un tipo per ogni modello di DM che il sistema può estrarre. Tali tipi rappresentano rispettivamente alberi di classificazione, regole di associazione, pattern sequenziali, cluster e gerarchie sui dati.

Ogni modello KDDML è espresso attraverso un'estensione di PMML 2.0 (par. 2.3), ed ogni modello è composto da:

³ Ogni volta che si entra nella fase di *preprocessing* è necessario l'utilizzo dell'operatore `<TABLE_2_PP_TABLE>`, mentre quando si conclude tale fase è necessario `<PP_TABLE_2_TABLE>`.

- un dizionario dei dati (*data dictionary*), che contiene una serie d'informazioni, come nome e tipo, dei campi utilizzati nel file impiegato per l'estrazione del modello di mining;
- un *mining schema*, che contiene una lista di campi che vengono utilizzati dal modello, ad esempio presenta una lista degli attributi che un utente deve fornire al fine di applicare il modello. Tale lista infatti indica quali tra gli attributi presenti nel data set svolgono un ruolo attivo all'interno dell'estrazione del modello.
- una descrizione del modello (*model description*), che rappresenta il modello vero e proprio. La sua rappresentazione varia a seconda del tipo di conoscenza che il modello esprime. Ad esempio, in Figura 2.4 del precedente paragrafo abbiamo presentato la struttura di un albero di classificazione.

Il linguaggio utilizza un'estensione di PMML, per fornire ulteriori informazioni sui modelli estratti, o per rappresentare modelli non ancora previsti da tale specifica. Il meccanismo di estensione, attuato attraverso il tag `<Extension>`, viene utilizzato da KDDML per fornire la matrice di confusione degli alberi di classificazione e per permettere il salvataggio di meta-classificatori (par. 1.4.2).

- *scalar*: rappresenta il tipo per definire numeri o stringhe;
- *alg*: definisce l'algoritmo di DM o preprocessing da utilizzare all'interno di un determinato operatore. È composto da:
 - *algorithm name*: identifica lo specifico algoritmo.
 - *parameter specifications*: rappresenta la lista dei parametri previsti per *algorithm name*. Ogni parametro è fornito attraverso la coppia formata dal nome del parametro e dal valore ad esso associato.

- *cond*: questa entità è usata per rappresentare specifiche condizioni. La condizione può essere usata per valutare operazioni booleane sugli attributi di una tabella e/o su costanti.
- *expr*: questo tipo viene utilizzato per esprimere espressioni del linguaggio. Un'espressione è simile ad una condizione, solo che ritorna uno scalare anziché un valore booleano.

Le query KDDML, quindi, non sono altro che termini definiti su un'algebra di operatori, rappresentati attraverso documenti XML costituiti da:

- *XML tag*: corrispondono ad operazioni su dati e/o modelli.
- *XML attribute*: corrispondono ai parametri dei vari operatori.
- *XML sub-element*: definiscono gli argomenti passati agli operatori.

Grazie a questa interpretazione, la semantica di una query KDDML equivale ad un'esecuzione funzionale stretta dei termini corrispondenti.

Di seguito forniamo la valutazione di un generico frammento XML:

```
<OPERATOR_NAME xml_dest="result.xml", attr_1="v_1" ... attr_m="v_m">
  <ARG_1_NAME> ... </ARG_1_NAME>
  ...
  <ARG_n_NAME> ... </ARG_n_NAME>
</OPERATOR_NAME>
```

1. ricorsivamente valutiamo uno dopo l'altro i frammenti da `<ARG_1_NAME> ... </ARG_1_NAME>` a `<ARG_n_NAME> ... </ARG_n_NAME>`. Se l'argomento *i*-esimo di `<OPERATOR_NAME>` è richiesto di tipo *t*, allora la valutazione di `<ARG_i_NAME> ... </ARG_i_NAME>` deve ritornare un elemento di tipo *t*.

2. successivamente valutiamo gli attributi previsti per `<OPERATOR_NAME>`. La valutazione di `attr_1="v_1" ... attr_m="v_m"`, ritorna un insieme di scalari.
3. chiamiamo infine $f_{\langle \text{OPERATOR_NAME} \rangle}$, utilizzando i risultati ottenuti ai punti 1 e 2, e producendo il risultato finale del frammento. In questo caso una copia del risultato è memorizzata nel *repository*, dato che l'attributo `xml_dest` è specificato. I *repository* utilizzati per dati e modelli sono persistenti, così da permettere il riutilizzo della conoscenza estratta.

La validazione della query, come visto nel paragrafo 2.2 per i documenti XML, avviene tramite le DTD, attraverso le quali si ottiene un'analisi statica della query KDDML.

KDDML fornisce diverse tipologie di operatori che possono essere classificati sulla base del loro tipo di ritorno. Abbiamo quindi operatori per:

- *Accedere ai dati*: attraverso questi operatori è possibile riferire, mediante un identificatore, una tabella presente nel *data repository*, leggere un file di tipo *arff* o leggere tuple da un database relazionale. Ad esempio il seguente operatore

```
<ARFF_LOADER xml_dest="Iris.xml"
              arff_file_path=".../MyData/"
              arff_file_name="Iris.arff">
```

legge il file `Iris.arff`, contenuto nella cartella specificata da `arff_file_path` e lo converte automaticamente nel formato utilizzato dal sistema, copiandolo inoltre nel *data repository*⁴.

Come è lecito attendersi, su tali tipi di import vi sono le corrispondenti operazioni di output.

⁴ Se l'attributo `arff_file_path` non è presente, il sistema cerca il file richiesto dall'attributo `arff_file_name` direttamente nel *data repository*.

- Attuare il preprocessing: rappresenta sicuramente, insieme al DM, una delle fasi più importanti e complesse del processo di KDD. Il preprocessing dei dati include parecchie operazioni, come la selezione dei dati, la discretizzazione, il cleaning, il filtraggio, l'aggiunta di nuovi attributi, etc. Com'è facile prevedere, KDDML include al suo interno diversi operatori di preprocessing. A titolo esemplificativo, presentiamo un frammento di linguaggio che utilizza un operatore di preprocessing `<PP_FILTER_ATTRIBUTES>`, per eliminare un attributo (*age*) da una tabella in input, fornita dall'operatore `<TABLE_LOADER>`:

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

- Estrarre *modelli di mining*: i modelli di mining vengono estratti da un determinato data set, attraverso l'applicazione di un algoritmo di mining. La struttura tipica di un operatore per l'estrazione di modelli, in questo caso di RdA, è la seguente:

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

Il primo argomento è il data set su cui estrarre la conoscenza (`BasketData.arff`), il secondo è l'algoritmo specifico per estrarre RdA, in questo caso DCI, con i suoi parametri. Inoltre il modello estratto (`MineBasket.xml`) viene salvato nel repository dei modelli.

Risulta quindi chiaro che, per ogni tipo di modello presente nel sistema possiamo avere più algoritmi che effettuano l'estrazione vera e propria.

- *Accedere a modelli precedentemente estratti:* KDDML mette a disposizione dell'utente una serie di operatori per accedere ed importare sia i modelli estratti, presenti nel repository, che modelli generati attraverso altri tool, purché in formato PMML 2.0.

Ad esempio il seguente frammento:

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

permette di importare un albero di decisione rappresentato nel file `DecisionTree.xml` presente nel model repository, mentre

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

permette di accedere in remoto, via ftp, al file `RdA.xml`, e di importarlo all'interno del sistema nel file `ExternRdA.xml`.

Per ogni tipo di modello gestito dal sistema, ci sono i corrispondenti operatori per importarlo sia da repository che in formato PMML.

- *Applicare, valutare modelli di mining e creare meta-modelli:* il sistema permette inoltre di applicare i modelli su nuovi dati, al fine di predirne caratteristiche, o selezionarli in accordo alla conoscenza memorizzata in un modello. Ad esempio l'operatore `<TREE_CLASSIFY>` permette di attuare il passo di classificazione su nuovi dati attraverso un albero di classificazione precedentemente estratto.

Sono inoltre presenti operatori per attuare ulteriori operazioni sui modelli estratti, come la loro combinazione con altri modelli. Fanno parte di questa categoria gli operatori `<TREE_COMMITTEE>` e `<RDA_FILTER>`. Il primo, utilizzando k alberi di decisione, crea un meta-classificatore, mentre il

secondo filtra le RdA presenti in un modello, in base a determinate condizioni definite dall'utente.

Oltre agli operatori appena descritti, KDDML offre operatori tipici di un linguaggio di programmazione, e non strettamente legati a problemi di KDD. Brevemente diciamo che abbiamo a disposizione operatori per:

- *Invocare programmi esterni o eseguire procedure RDBMS (Relational DBMS)*: grazie a questo operatore (<EXT_CALL>) è possibile, ad esempio, invocare procedure specializzate nell'analisi dei dati non previste dal sistema, oppure eseguire procedure RDBMS.
- *Invocare query KDDML*: grazie alla presenza di questo meccanismo, è possibile suddividere lunghe query KDDML in sottoquery, che possono essere parametriche, e vengono salvate in un opportuno query repository. L'operatore <CALL_QUERY> permette infatti di richiamare le sottoquery, occupandosi del passaggio dei parametri e dell'esecuzione.
- *Definire la modalità di esecuzione* (seriale o parallela) dei vari operatori presenti. Può essere talvolta utile valutare i vari operatori di una query in sequenza o segnalarli al sistema come parallelizzabili. KDDML mette a disposizione dell'utente due operatori (<SEQ_QUERY> e <PAR_QUERY>) per il controllo del flusso di valutazione della query, attraverso i quali l'utente può creare parti di query da eseguire parallelamente o in stretta sequenza.
- *Definire blocchi condizionali e ricorsivi* all'interno della query KDDML. Il sistema fornisce anche altri metodi per il controllo del flusso di una query

KDD. Tali meccanismi sono composti dalla presenza del costrutto condizionale⁵,

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

e dalla possibilità di attuare chiamate ricorsive, permettendo all'operatore <CALL_QUERY> di richiamare la stessa query a cui appartiene.

2.4.2 L'ARCHITETTURA

Dopo aver trattato le caratteristiche salienti del linguaggio andiamo a fornire anche una breve descrizione dell'architettura di KDDML. L'architettura del sistema è stata a lungo studiata per garantire un'ottima modularità ed offrire caratteristiche che permettano una facile estensione del sistema.

KDDML prevede tre tipi di estendibilità:

- *Data source*: l'aggiunta di un nuovo tipo di sorgente dati nel linguaggio avviene semplicemente aggiungendo un nuovo *tag*, con i vari attributi e sottoelementi, e specificando come individuare i dati.

Dal punto di vista del sistema, questo richiede l'inserimento di wrapper per la manipolazione dei dati fisici dal nuovo formato a quello utilizzato dal sistema e viceversa.

- *Algoritmi*: questa caratteristica riguarda la possibilità di estendere il sistema con nuovi algoritmi da utilizzare, ad esempio, all'interno di operatori per il

⁵ La valutazione del costrutto condizionale, a differenza dei costrutti visti fino ad ora, avviene attuando una valutazione non stretta dei due rami. In base al valore della condizione COND, infatti, si valuta solo il ramo THEN o solo il ramo ELSE.

preprocessing o per l'estrazione di modelli tra quelli già previsti dal linguaggio. L'introduzione di un nuovo algoritmo deve avvenire in maniera piuttosto semplice, dato che si pensa che questo tipo di estensione sia quello più utilizzato dall'utente. Ad esempio, un utente può voler utilizzare un proprio algoritmo, non presente in KDDML per l'estrazione di RdA.

Dal punto di vista del linguaggio, l'introduzione di un nuovo algoritmo non richiede modifiche ad elementi già esistenti, in quanto sia il nome dell'algoritmo che i suoi parametri non sono parte della sintassi del linguaggio.

- *Operatori*: riguarda la possibilità di estendere il linguaggio con nuovi operatori, per incrementare le funzionalità offerte dal sistema. La loro integrazione è molto simile a quella prevista per gli algoritmi.
- *Modelli*: estendere il sistema con nuovi modelli vuol dire aggiungere nuove forme di conoscenza estraibile con KDDML. Questo tipo di estensione è sicuramente quella più delicata, poiché richiede l'introduzione di diverse componenti a vari livelli del sistema. È improbabile che un utente si cimenti nell'introduzione di nuovi tipi di conoscenza, tuttavia il software deve essere concepito per permettere l'introduzione di nuovi modelli, in modo da non richiedere la modifica dell'intero sistema.

Questa caratteristica è molto importante, in quanto il mondo del DM è in costante evoluzione ed un sistema deve essere in grado di introdurre tempestivamente nuovi modelli.

KDDML è implementato in JAVA [SUN], è quindi portabile e consiste di oltre 500 classi. La sua architettura è strutturata a livelli, ed ogni livello implementa funzionalità specifiche, e fornisce ai livelli superiori una serie di interfacce per accedere alle varie operazioni implementate dal livello.

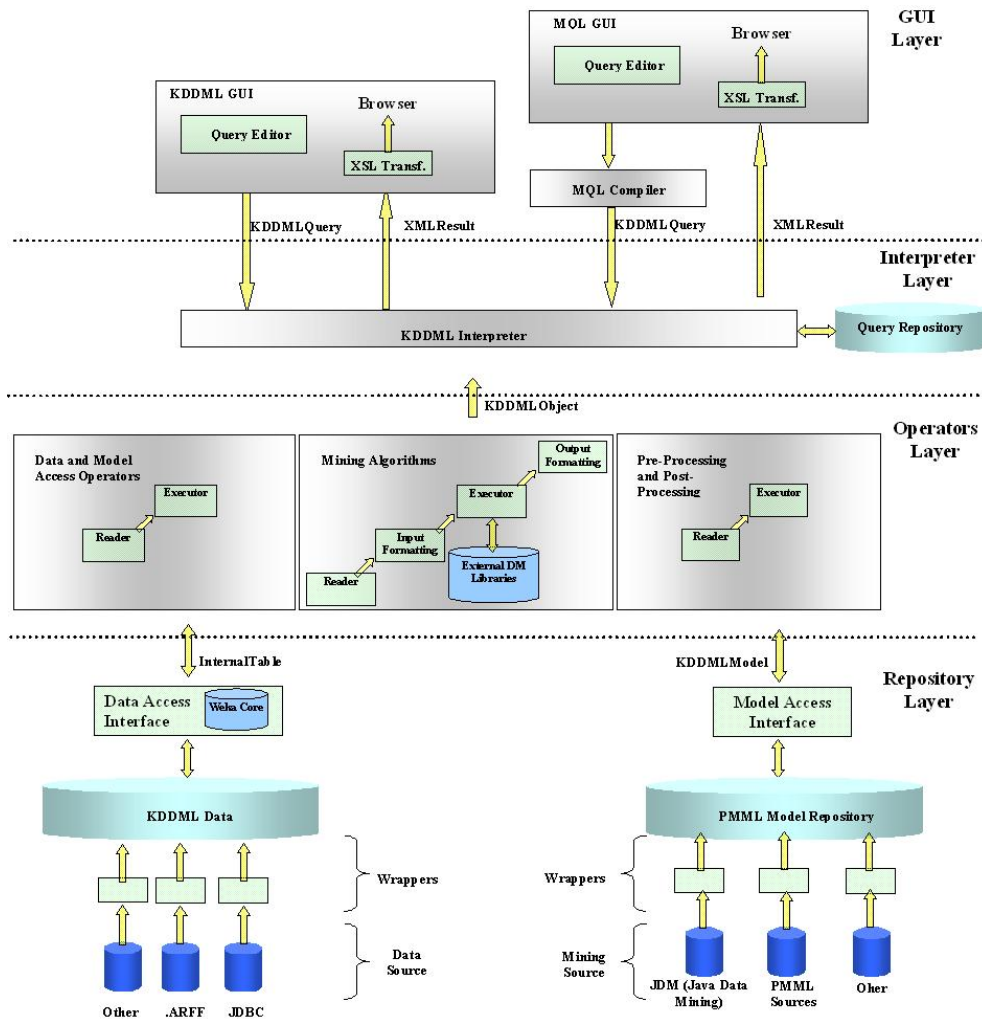


Figura 2.5 Architettura del sistema KDDML.

Com'è possibile vedere dalla Figura 2.5, i livelli che costituiscono il sistema, partendo dal basso, sono:

- **Livello *Repository* (*repository layer*):** rappresenta il livello più basso del sistema, quello che si interfaccia fisicamente con i dati. I compiti di questo livello riguardano gli accessi in lettura/scrittura ai vari *repository*, per fornire ai livelli più alti la possibilità di accedere sia alle tabelle proprietarie contenute nel *data repository*, che ai modelli, sia da *repository* che da altre fonti e da sorgenti dati esterne come RDBMS.

Questo livello può ulteriormente essere diviso in due grandi sub-layer:

- *Data & models manager*: fornisce ai livelli più alti un'interfaccia per manipolare tabelle e modelli di mining. L'accesso ad una tabella produce un oggetto che soddisfa l'interfaccia interna `InternalTable`, che presenta metodi per leggere/scrivere nella tabella, accedere ai meta-dati e presentare statistiche sul data set.
L'accesso ad un modello fornisce oggetti diversificati, in base al tipo di modello richiesto (`AssociationModel`, `TreeModel`, ...), garantendo metodi per accedere in lettura/scrittura al modello.
- *Data & models factory*: include tutti i *wrapper* per accedere e importare tabelle e modelli, sia presenti nei vari *repository* che da fonti esterne. I *wrapper* si occupano di trasformare i dati dal loro formato originale alla loro rappresentazione interna (`InternalTable`, `AssociationModel`, ...).
I *wrapper* sono organizzati in una *factory*, al fine di permettere la separazione dell'accesso alle sorgenti, dalla gestione degli oggetti per dati e modelli.
- *Livello operatori e algoritmi (operators & algorithms layer)*: contiene le implementazioni di tutti i vari operatori del linguaggio. Gli operatori sono suddivisi in base al compito che svolgono all'interno del sistema, e dal punto di vista della loro implementazione si diversificano in:
 - *Operatori per accedere a dati o modelli*: sono implementati attraverso chiamate dirette al *repository layer*, al fine di caricare o memorizzare dati o modelli.
 - *Operatori di mining*: invocano un algoritmo per estrarre o applicare un modello, che tipicamente è un programma esterno che richiede un proprio formalismo per i dati in input e fornisce il risultato in formato proprietario.
L'implementazione deve quindi farsi carico di effettuare le giuste trasformazioni sia per gli input che per gli output, al fine di garantire il

ritorno di un appropriato `KDDMLObject` come risultato.

- Operatori di pre/post processing: la loro implementazione è molto simile a quella degli operatori di mining, mentre la principale differenza è rappresentata dal fatto che essi sono direttamente implementati nel sistema, e non sono programmi esterni come gli algoritmi definiti sopra, ed inoltre accedono direttamente ad oggetti di tipo `InternalTable` o `KDDMLModel`. Questa scelta è dettata dal fatto che solitamente gli operatori che fanno parte di questa categoria trasformano dati. In tale trasformazione, che in genere è time-consuming, deve pertanto essere garantita la massima efficienza.
- Livello interprete (*interpreter layer*): è il motore dell'intero sistema, accetta, valida ed esegue query KDDML, si occupa inoltre del salvataggio del risultato finale nel *repository* e ritorna un `KDDMLObject`.

In Figura 2.6 mostriamo il codice dell'interprete. Come si può notare, esso visita ricorsivamente l'albero DOM della query, producendo un `KDDMLObject` come risultato.

La figura in questione mostra un interprete piuttosto semplificato, il cui scopo è solo quello di fornire un'intuizione del suo funzionamento; mancano infatti tutti i controlli di correttezza, e la parte riguardante gli operatori speciali, come `<IF>` o `<CALL_QUERY>`, che devono essere trattati con opportuni accorgimenti.

L'interprete può essere quindi diviso in tre parti principali, corrispondenti alle 3 parti di valutazione di un frammento XML visto nel paragrafo 2.4.1:

Fase 1

Partendo dal tag radice della query, si valutano tutti i sottoelementi ricorsivamente.

Una volta terminata la valutazione di tutti i sottoelementi, otteniamo un vettore di `KDDMLObject` (*parameters*), che rappresenta i parametri da utilizzare nell'applicazione del nodo *root* chiamante.

Fase 2

Si vanno a valutare tutti gli attributi previsti per il *tag* in esame. Gli attributi sono raggruppati in una tabella *hash* di scalari (KDDMLScalar).

Fase 3

Si esegue l'operatore associato al *tag*, passandogli come parametri il vettore *parameters* di KDDMLObject e l'*hash table* di scalari.

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

    KDDMLOperator op = KDDMLFactory.getOperator( query.getTagNames() );
    Vector params = new Vector();

    for (int i=0; i<children.size(); i++) {
        Element elem = (Element) children.get(i);
        KDDMLObject obj = resolve(elem, op.paramType(i) );
        params.add(obj);
    }
    NamedNodeMap list = query.getAttributes();
    Hashtable attributes = new Hashtable();

    for (int i=0; i<list.getLength(); i++) {
        Node n = list.item(i);
        KDDMLScalar expr = exprEval( n.getNodeValue() );
        attributes.put( n.getNodeName(), expr );
    }

    KDDMLObject result = null;

    result = op.execute( params, attributes );

    return result;
}
```

*Fase 1**Fase 2**Fase 3*

Figura 2.6 Codice dell'interprete KDDML.

In tutti i casi, se l'attributo *xml_dest* è specificato, il risultato viene memorizzato dal sistema automaticamente nell'appropriato *repository*.

Altre osservazioni da fare riguardano l'invocazione degli algoritmi di mining. Quest'ultima avviene attraverso una ricerca per nome, tramite il caricamento dinamico della classe che corrisponde al nome derivato dall'identificatore dell'algoritmo.

Una serie di controlli sono inoltre previsti, sia sulla presenza dei nomi degli algoritmi, sia sul tipo dei parametri, e sul risultato atteso dalla valutazione dei sottoelementi.

2.4.3 ESEMPIO DI APPLICAZIONE DI KDDML

Mostriamo in questo paragrafo un esempio di applicazione di KDDML, al fine di chiarire alcuni concetti delineati nei due paragrafi precedenti.

Innanzitutto partiamo dai dati, e vediamo un esempio di come KDDML li rappresenta. Una tabella in KDDML è rappresentata attraverso due file, un file XML per rappresentare la struttura logica dei dati, ed un file CVS in cui sono memorizzati i dati fisici.

```
<?xml version="1.0" encoding="UTF-8"?>
<KDDML_OBJECT>
  <KDDML_TABLE data_file="weather.csv">
    <SCHEMA logical_name="weather" number_of_attributes="5" number_of_instances="20">
      <ATTRIBUTE name="outlook" number_of_missed_values="2"
        number_of_missed_values_perc="10%" type="nominal">
        <NOMINAL_DESCRIPTION number_of_values="3">
          <VALUE value="rainy" cardinality="5" cardinality_perc="28%"/>
          <VALUE value="sunny" cardinality="8" cardinality_perc="44%"/>
          <VALUE value="overcast" cardinality="5" cardinality_perc="28%"/>
        </NOMINAL_DESCRIPTION>
      </ATTRIBUTE>
      <ATTRIBUTE name="temperature" number_of_missed_values="0"
        number_of_missed_values_perc="0%" type="numeric">
        <NUMERIC_DESCRIPTION mean="74.0" variance="35.36842105263158"
          sum="1480.0" sumSq="110192.0" min="64.0" max="85.0"/>
      </ATTRIBUTE>
      <ATTRIBUTE name="humidity" number_of_missed_values="4"
        number_of_missed_values_perc="20%" type="numeric">
        <NUMERIC_DESCRIPTION mean="81.875" variance="77.18333333333334"
          sum="1310.0" sumSq="108414.0" min="65.0" max="95.0"/>
      </ATTRIBUTE>
      <ATTRIBUTE name="windy" number_of_missed_values="6"
        number_of_missed_values_perc="30%" type="nominal">
        <NOMINAL_DESCRIPTION number_of_values="2">
          <VALUE value="TRUE" cardinality="4" cardinality_perc="29%"/>
          <VALUE value="FALSE" cardinality="10" cardinality_perc="71%"/>
        </NOMINAL_DESCRIPTION>
      </ATTRIBUTE>
      <ATTRIBUTE name="play" number_of_missed_values="0"
        number_of_missed_values_perc="0%" type="nominal">
        <NOMINAL_DESCRIPTION number_of_values="2">
          <VALUE value="no" cardinality="8" cardinality_perc="40%"/>
          <VALUE value="yes" cardinality="12" cardinality_perc="60%"/>
        </NOMINAL_DESCRIPTION>
      </ATTRIBUTE>
    </SCHEMA>
  </KDDML_TABLE>
</KDDML_OBJECT>
```

Figura 2.7 Esempio di meta-dati utilizzati da KDDML.

In Figura 2.7 è rappresentato il file XML che mostra i meta-dati riguardanti informazioni meteorologiche per decidere se giocare o meno una partita di tennis.

Attraverso di esso, il sistema riesce ad indirizzare il file fisico (`weather.csv`), a capire il numero di attributi, in questo caso 5 ed il numero di istanze (20). Inoltre, per ogni attributo vengono dati il tipo ed il numero di valori mancanti. Nel caso di attributi nominali, come ad esempio `outlook`, vengono forniti la cardinalità e la percentuale di occorrenze di ogni valore assunto dall'attributo, mentre nel caso di valori numerici (`humidity`) abbiamo varie statistiche come il valore minimo, massimo, la media, la varianza, etc.

Dopo aver visto come KDDML rappresenta i dati, andiamo ad esaminare un esempio di query vera e propria. A tal fine scegliamo l'estrazione e la successiva applicazione di un albero di decisione. In Figura 2.8 abbiamo il file XML che rappresenta la query in questione.

```
<KDD_QUERY name="sample">
  <TREE_CLASSIFY xml_dest="results.xml">
    <TREE_MINER xml_dest="tree.xml" target_attribute="class">
      <TABLE_LOADER xml_source="trainingSet.xml"/>
      <ALGORITHM algorithm_name="YADT">
        <PARAM name="confidence_for_pruning" value="0.4"/>
        <PARAM name="num_instances_for_leaf" value="0.6"/>
      </ALGORITHM>
    </TREE_MINER>
    <TABLE_LOADER xml_source="testSet.xml"/>
  </TREE_CLASSIFY>
</KDD_QUERY>
```

Figura 2.8 File XML rappresentante una query KDDML.

Andiamo ad analizzare gli elementi costitutivi di tale query KDDML.

Attraverso gli attributi presenti nel tag radice `<KDD_QUERY>`, è possibile associare un identificatore alla query. Inoltre, possiamo dichiarare una serie dei parametri, nel caso si voglia costruire una query parametrica invocabile con il tag `<CALL_QUERY>`.

Scendendo di livello, troviamo l'operatore `<TREE_CLASSIFY>` che si occupa di applicare un albero di classificazione per predire la classe a cui appartengono le tuple di un dato test set. Tramite l'attributo `xml_dest`, informiamo il sistema della necessità di salvare il risultato della classificazione nel *data repository* per ulteriori analisi.

L'albero di classificazione che deve essere applicato è fornito dal primo sottoelemento `<TREE_MINER>`, attraverso il quale è specificata la costruzione di un albero di decisione. I suoi attributi riguardano il nome del file che identifica l'albero nel *model repository* ed il *class_attribute* da predire.

La costruzione vera e propria avviene fornendo a `<TREE_MINER>` sia il training set, attraverso il tag `<TABLE_LOADER>`, che l'algoritmo specifico da utilizzare per l'estrazione (`<ALGORITHM>`).

Il nome di tale algoritmo viene specificato nell'attributo del tag `<ALGORITHM>`, mentre i suoi parametri vengono inseriti attraverso il tag `<PARAM>`. Infine, attraverso il tag `<TABLE_LOADER>`, in basso viene fornito il file su cui eseguire la fase di testing.

Sottolineiamo il fatto che tutti i file riferiti all'interno dei vari tag verranno caricati/memorizzati nei vari repository fissati dal sistema.

Una volta costruita la query, andiamo a vedere come viene eseguita dal sistema. In primo luogo, il sistema costruisce l'albero DOM, ed attraverso questa costruzione effettua anche controlli, grazie alle DTD, sulla validità del documento XML.

Da questo passo otteniamo l'albero DOM rappresentato in Figura 2.9, in cui per semplicità abbiamo omissi gli attributi previsti per il tag `<PARAM>`.

L'esecuzione della query avviene attraverso la valutazione dei vari tag presenti, che corrisponde ad una visita "depth first", da sinistra a destra, dell'albero DOM. In Figura 2.9 sono, inoltre, evidenziati i passi di valutazione con l'utilizzo di frecce più spesse che collegano i vari nodi. Seguendo tali frecce, si capisce la sequenza in cui vengono valutati ed invocati i vari operatori corrispondenti ai tag.

Il primo ad essere valutato è il tag `<TABLE_LOADER xml_source="trainingSet.xml">`. Dato che quest'ultimo non ha figli, si vanno a valutare i suoi attributi, e ad eseguire l'operatore ad esso associato. L'esecuzione di tale operatore fornisce l'oggetto `InternalTable`, creato sul file richiesto, che viene passato al nodo superiore. L'operatore di costruzione dell'albero non può essere ancora invocato, poiché non abbiamo finito di valutare tutti i suoi sottoelementi.

Andiamo quindi a visitare il nodo <ALGORITHM>. Questo rappresenta un caso base della ricorsione, e quindi viene passato insieme ai suoi sottoelementi l'oggetto che permette l'invocazione dell'algoritmo al nodo padre.

A questo punto il nodo <TREE_MINER> va a valutare i propri attributi, ed attraverso un vettore di oggetti (KDDMLObject), riguardante i risultati della valutazione di tutti i suoi sottoelementi, ottiene tutte le informazioni necessarie per la sua esecuzione. Tale esecuzione produrrà a sua volta un KDDMLObject, che andrà a far parte del vettore di oggetti del nodo padre e così via.

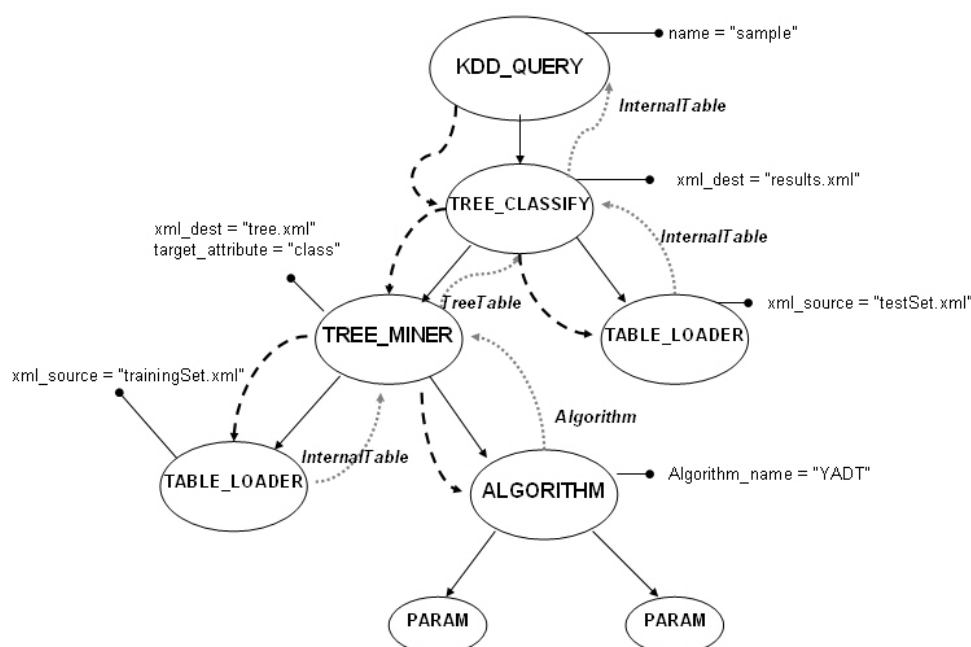


Figura 2.9 Albero DOM per la query di Figura 2.8.

Il risultato finale di tutta l'esecuzione sarà quindi un KDDMLObject.

In Figura 2.10, per completezza viene presentata la gerarchia degli oggetti (KDDMLObject) presenti nel sistema KDDML.

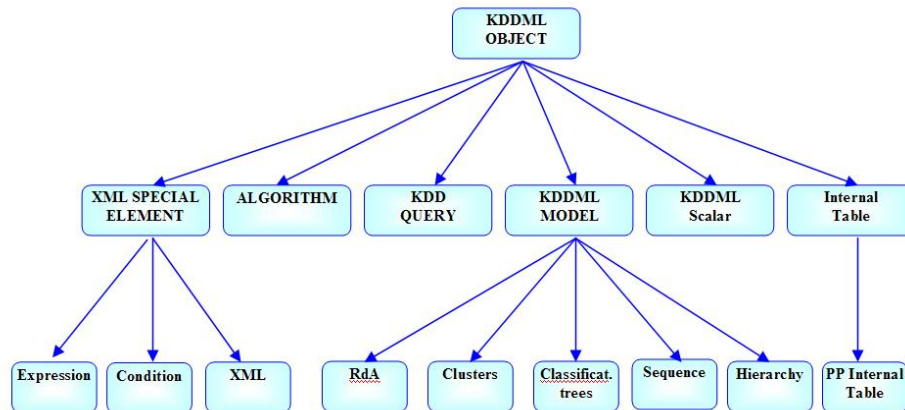


Figura 2.10 Gerarchia degli oggetti del sistema KDDML.

In Figura 2.11, è rappresentato il risultato della query di Figura 2.8. Tale informazione è ottenuta attraverso l'apertura di un browser web. Una volta terminato il processo, il sistema converte il risultato da XML ad HTML.

KDDML Table Visualization							
Schema Informations							
<ul style="list-style-type: none"> Logical Name : iris XML Reference : results.xml Total Number of Attributes : 6 Total Number of Instances : 15 							
Attribute Informations							
Attribute Info							
Field Name	Type	Missed Values (%)	Values: Cardinality (%)	Mean	Variance	Min Value	Max Value
sepal.length	numeric	0 (0%)	n/a	5.66	0.7011428571428512	4.4	6.9
sepal.width	numeric	0 (0%)	n/a	2.9933333333333336	0.200666666666666524	2.3	3.8
petal.length	numeric	0 (0%)	n/a	3.6866666666666667	3.0783809523809498	1.3	5.9
petal.width	numeric	0 (0%)	n/a	1.2866666666666666	0.6883809523809522	0.2	2.5
class	nominal	0 (0%)	Iris-versicolor: 5 (33%) Iris-virginica: 5 (33%) Iris-setosa: 5 (33%)	n/a	n/a	n/a	n/a
class_predicted	nominal	0 (0%)	Iris-versicolor: 6 (40%) Iris-virginica: 5 (33%) Iris-setosa: 4 (27%)	n/a	n/a	n/a	n/a

Figura 2.11 Risultato della query di Figura 2.8.

2.4.4 LA GUI ATTUALE

Attualmente il sistema prevede una GUI (Figura 2.12), che adottando un approccio text-driven, aiuta l'utente, nella stesura di query KDDML.

Attraverso di essa è possibile:

- Caricare e modificare una query esistente.
- Creare nuove query mediante un approccio syntax-driven.
- Validare ed eseguire la query
- Trasformare i risultati in formato HTML, attraverso i fogli di stile XSL.

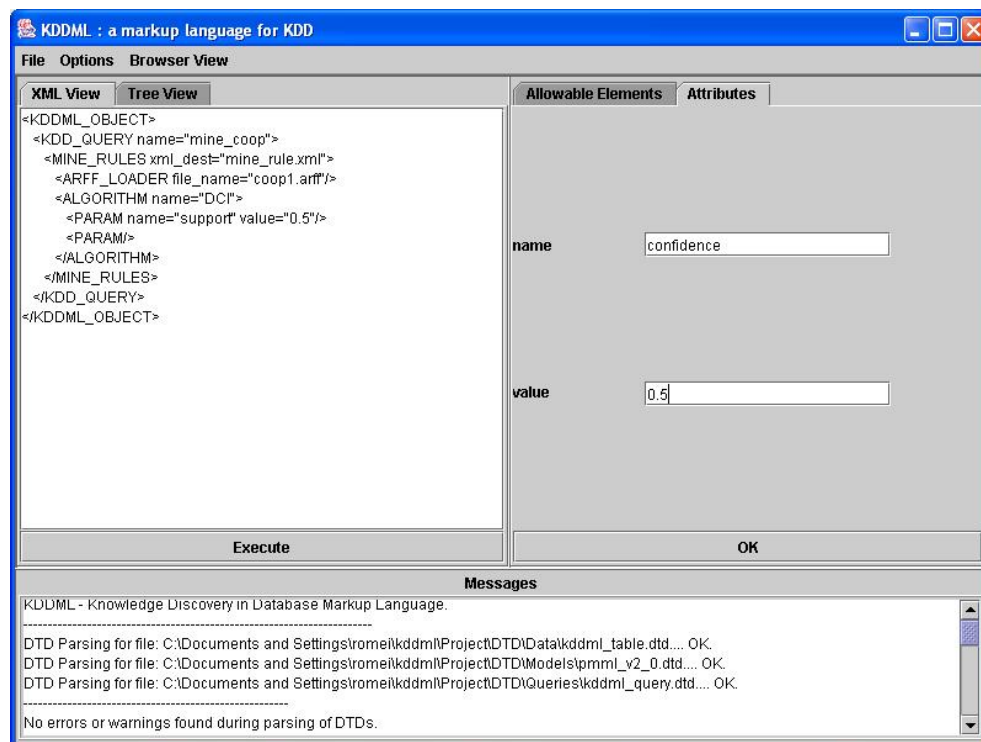


Figura 2.12 La GUI attuale del sistema KDDML.

L'interfaccia non fa parte del core del sistema, e le query KDDML possono essere generate anche da altri programmi.

Non è comunque pensabile fornire all'utente solo questo tipo di interfaccia, poiché come abbiamo potuto renderci conto anche dai semplici esempi di questo capitolo, la stesura di una query KDDML non è un procedimento facile ed immediato, ma richiede un notevole sforzo per comprendere quali operatori il sistema offre e le politiche adottate sia per la rappresentazione che per l'esecuzione delle query.

L'obiettivo futuro è quindi quello di offrire all'utente la possibilità di interagire con il sistema tramite un linguaggio grafico, che permetta di rappresentare al meglio

processi KDD attuabili da KDDML, senza doversi scontrare con il linguaggio KDDML vero e proprio, e cercando allo stesso tempo di mettere in risalto le potenzialità del sistema. Inoltre, altre caratteristiche devono essere implementate, al fine di garantire un buon livello d'immediatezza ed interattività.

Nel prossimo capitolo ci concentreremo sull'analisi di alcuni sistemi che prevedono già metafore grafiche per la rappresentazione di processi KDD.

Capitolo 3

SISTEMI DI KD E LORO ASPETTI VISUALI

3.1 INTRODUZIONE

Negli ultimi anni si è registrato un notevole incremento di software rivolti al DM ed al machine learning. Tali software nella maggior parte dei casi raccolgono all'interno di un unico tool una serie di algoritmi conosciuti da anni, e sviluppati principalmente in ambito accademico ma rimasti per molto tempo confinati in tale settore.

L'obiettivo primario di tali software è quello di fornire strumenti validi per l'estrazione della conoscenza, che si integrino sempre più a strumenti ormai di largo uso in molteplici settori, come database o data warehouse.

In questo capitolo ci poniamo l'obiettivo di offrire una rassegna di alcuni di questi strumenti, e successivamente, di cercare di contrapporre le varie filosofie implementative, al fine di identificare delle linee guida in base alle quali costruire una nostra metafora grafica per KDDML.

Prima di entrare nella fase di rassegna vera e propria, ci concentreremo sulle caratteristiche essenziali che ci hanno permesso di individuare una rosa di candidati a tale scopo.

In un primo tempo abbiamo scelto i tool in base alle seguenti caratteristiche:

1. Sono *free*, e possibilmente open source.

2. Sono dotati di interfaccia grafica, intesa non come un semplice editor text-driven⁶, ma come una sorta di astrazione, metafora grafica in grado di permettere una programmazione visuale di processi di DM, e di fornire una visualizzazione degli output generati dal sistema.

Questi due punti ci hanno consentito di restringere il nostro campo di ricerca, poiché esistono numerosi strumenti di DM, ma soltanto un numero limitato di essi presentano le caratteristiche elencate.

I software scelti per il nostro studio sono:

- **WEKA** [WEKA], uno dei tool più completi e di maggior successo, almeno in ambito accademico.
- **YALE** [YALE], che presenta caratteristiche comuni a KDDML e, come quest'ultimo, utilizza file XML per descrivere query di DM.
- **ORANGE** [ORANGE], un vero e proprio strumento di *visual programming*, attraverso il quale è possibile costruire vere e proprie applicazioni *stand alone*.
- **TANAGRA** [TAN], che offre una visione integrativa sul metodo di costruzione dell'esperimento di DM e sulla rappresentazione dei risultati.

A questi software abbiamo ritenuto opportuno affiancare anche uno strumento sviluppato a fini commerciali. Tale scelta è stata principalmente dettata dalla necessità di avere anche un riscontro con gli standard qualitativi che un software, sviluppato a scopi commerciali, deve possedere.

Il software che è stato aggiunto alla lista è **GHOSTMINER** [FGM], e la nostra scelta in questo caso è stata dettata principalmente dalla possibilità di avere una copia temporanea di valutazione.

⁶ Come previsto attualmente da KDDML.

È necessario precisare che la lista di software qui presentata non è né completa né esaustiva, come si può notare anche attraverso una semplice ricerca su internet. Tuttavia, il nostro obiettivo non è quello di fornire un semplice stato dell'arte o una rassegna di tutte le interfacce grafiche attualmente presenti in letteratura, ma di intraprendere uno studio generalizzato, non limitato ad uno specifico tool, su quelle che sono le possibili metafore grafiche e le caratteristiche rilevanti presenti nei tool analizzati, da applicare nella rappresentazione grafica di query KDD scritte in KDDML.

Il seguito del capitolo tratterà uno ad uno i vari software elencati sopra, fornendo per ognuno di essi sia una breve descrizione di carattere generale, che un approfondimento riguardante la specifica interfaccia grafica.

Ogni analisi sarà corredata da un esempio per chiarire i concetti presentati. Tale esempio riguarderà l'estrazione ed il testing di un albero di classificazione, e verrà riproposto, seppur con alcune differenze, per tutti i software analizzati. Questa scelta è dettata principalmente dalla volontà di offrire un primo nucleo comune di paragone, al fine di evidenziare le differenti filosofie di costruzione e visualizzazione del processo in questione. Tale esempio risulta inoltre piuttosto semplice e si rivela pertanto particolarmente adatto a fornire all'utente anche una visione dei tool in esame.

3.1.1 PARAMETRI GENERALI DI VALUTAZIONE

Prima di entrare nello specifico dei vari tool presenti, forniamo un elenco di parametri in base ai quali valutare un software di DM. Tali parametri sono di carattere generale e devono essere integrati anche con criteri di valutazione soggettivi, o relativi ad una particolare realtà in cui si vuole inserire tale sistema.

- **Tipi di dato e sorgenti:** si riferisce a specifici formati di dati sui quali il sistema opera, ed alla possibilità di utilizzare diverse sorgenti come database o data warehouse, per recuperare i dati. Ad esempio, alcuni sistemi possono operare soltanto su file di testo ASCII presenti localmente, mentre altri

possono lavorare anche su dati ricavati attraverso connessioni a database relazionali.

- **Funzioni e metodologie di Data Mining:** le funzioni previste rappresentano il nucleo operativo di un sistema di DM. In questo caso siamo di fronte ad un'infinita serie di possibilità, in quanto alcuni software possono fornire solo una funzionalità, come la classificazione, mentre altri possono coprire una vasta area di problematiche, come il clustering, la ricerca di similarità, l'analisi dei pattern sequenziali, etc. In più c'è da considerare che, per ogni funzionalità, un sistema può prevedere svariati metodi di risoluzione. I sistemi che presentano più funzioni e più metodi di risoluzione per ogni funzione garantiscono all'utente maggiore flessibilità e potenza.
- **Abbinamento di strumenti di Data Mining a database o data warehouse:** un sistema di DM dovrebbe essere associato ad un sistema di database e/o data warehouse, dove le componenti accoppiate siano perfettamente integrate in un ambiente di *information processing* uniforme. In generale, possiamo avere diverse forme di accoppiamento. Alcuni sistemi di DM lavorano solo con file di dati ASCII e non sono abbinati a nessun sistema di database o data warehouse, mentre altri recuperano i dati direttamente da database (o data warehouse) e possono condividere con questi ultimi alcune primitive di DM.
- **Scalabilità:** è generalmente legata alla possibilità, offerta da un software, di manipolare enormi quantità di dati. Questa misura è legata alla politica di gestione del data set processato. Alcuni software richiedono che il data set venga caricato per intero in memoria per essere processato. Tali sistemi risultano poco scalabili e possono essere inefficienti.
- **Strumenti di visualizzazione:** questi strumenti rappresentano un punto critico per ogni software, e possono essere classificati in varie categorie, come visualizzazione dei dati, dei risultati, etc. Possiamo indubbiamente affermare che la varietà, la qualità, e la flessibilità degli strumenti di

visualizzazione influiscono fortemente sull'usabilità, l'interpretabilità e l'interesse di uno strumento di DM.

- **Graphical user interface:** il DM è considerato un processo esplorativo. Come tale, un'idonea interfaccia grafica, fortemente interattiva e facile da usare, è essenziale per promuovere al meglio tale processo. È da evidenziare, comunque, che a differenza dei sistemi sviluppati per i database relazionali, dove la maggior parte delle interfacce è costruita sulla base di SQL, i sistemi per il DM non condividono alcun sottostante *data mining query language*. La mancanza di tale standard rende complessa la standardizzazione di prodotti sviluppati in tale settore, e limita fortemente l'inter-operabilità tra i sistemi stessi.
- **Sistema Operativo supportato:** bisogna sempre considerare anche il sistema operativo sul quale il software opera. La maggior parte dei sistemi vengono sviluppati per ambienti *Unix* e *Microsoft Windows*, anche se con l'avvento di JAVA [SUN], tale problematica viene in parte superata.

Per quel che riguarda il seguito della nostra trattazione, diciamo che, nella valutazione dei software, ci soffermeremo a definire tali prodotti solo dal punto di vista della loro interfaccia grafica, con particolare riferimento alla metafora grafica per rappresentare query KDD, ed in parte rispetto agli strumenti di visualizzazione dei risultati che essi forniscono.

Questa distinzione viene motivata dal fatto che, volendo costruire una metafora grafica su un sistema già esistente e funzionante indipendentemente da quest'ultima, assume particolare rilevanza solo il livello più elevato del sistema, ovvero la parte di interazione tra uomo e macchina.

3.2 WEKA

Il primo sistema che andremo ad analizzare si chiama WEKA [WEKA]. WEKA, acronimo di Waikato Environment for Knowledge Analysis, è un sistema open source che raccoglie una collezione di algoritmi di machine learning per eseguire

operazioni di DM, ed è sviluppato all'Università di Waikato in Nuova Zelanda. Il sistema è scritto in JAVA, che permette di fornire un'interfaccia uniforme a molti algoritmi di apprendimento, insieme a metodi di pre-post processing e di valutazione del risultato di schemi di apprendimento su un dato insieme di dati.

WEKA fornisce l'implementazione di algoritmi di machine learning [WF00], che possono essere applicati ad un dato insieme di dati da riga di comando. Il sistema si avvale di una grande quantità di strumenti utili a trasformare i dati da un insieme di valori ad un altro, come nel caso degli algoritmi per la discretizzazione, i quali prendono un set di valori continui di un dato attributo e lo riducono ad un insieme di intervalli. È inoltre possibile preprocessare una serie di dati, per adattarli alle esigenze dei vari schemi di apprendimento, ed analizzare le performance del classificatore utilizzato senza la necessità di scrivere codice.

Risulta chiaro che il sistema permette di essere utilizzato a diversi livelli. In primo luogo, è possibile utilizzare WEKA per applicare un metodo di apprendimento ad un insieme di dati, ed analizzare il suo output per estrarre informazioni riguardanti i dati. Un altro metodo possibile è quello di applicare alcuni algoritmi di learning e confrontare le loro performance, in modo da sceglierne uno per fare delle previsioni.

Ad esempio, i classificatori hanno tutti la stessa interfaccia *command-line*, ma si distinguono per le varie opzioni specifiche che ogni classificatore offre. Analogo discorso può essere fatto per gli altri componenti del sistema come ad esempio i filtri (*filters*), attraverso i quali è possibile ridurre, filtrare i dati in input, come ad esempio rimuovere dal data set tutte le tuple con valori mancanti.

WEKA permette anche, in qualche misura, di essere visto come un *middleware*, attraverso il quale è possibile risolvere sottoproblemi di machine learning e DM della propria applicazione con un minimo di programmazione aggiuntiva, semplicemente accedendo ai veri metodi forniti dall'interno del proprio codice.

Questo è possibile, poiché WEKA sfrutta la possibilità, offerta da JAVA, di strutturare i propri applicativi in package, ossia directory contenenti collezioni di classi correlate tra loro, quindi semplicemente importando i package nel proprio programma JAVA, è possibile richiamare dal proprio codice le parti del sistema che occorrono.

Una dettagliata e aggiornata informazione su tutti i package, le loro interfacce e classi è disponibile all'interno della documentazione fornita insieme al sistema e scaricabile gratuitamente dalla sua home page [WEKA].

Fino ad ora abbiamo parlato di WEKA come sistema invocabile attraverso linea di comando, ma sappiamo bene che, con l'avvento di sistemi sempre più complessi, tale approccio risulta essere sempre più abbandonato, preferendo quelle che generalmente vengono indicate come *Graphical User Interface* (GUI). WEKA non è da meno in tale trend, infatti, mette a disposizione dell'utente un'interfaccia grafica che permette di sfruttare le potenzialità del sistema semplicemente utilizzando il mouse.

La nostra attenzione, dunque, si sposta esclusivamente sulla GUI di WEKA, per valutare se la metafora scelta è robusta rispetto alla nostra visione di un query language KDD, molto vicina alla visione di una query SQL.

3.2.1 WEKA GUI

Come abbiamo detto prima, WEKA è uno strumento piuttosto complesso, in grado di coprire svariate problematiche di DM. Questo si riflette di conseguenza sull'interfaccia, che risulta vasta, ricca di caratteristiche che in parte esulano dalla nostra trattazione.

Per brevità non esporremo minuziosamente tutte le caratteristiche presenti, ma dopo averne dato una caratterizzazione generale, ci soffermeremo solo su quelle parti caratteristiche, che secondo noi si possono adattare bene alla nostra visione di un linguaggio KDD.

La GUI si apre con una piccola finestra (Figura 3.1), attraverso la quale è possibile scegliere l'ambiente operativo, ovvero la tipologia d'uso che vogliamo fare del sistema:

- **Simple CLI:** è un ambiente a riga di comando, attraverso il quale è possibile invocare le varie funzionalità di cui WEKA è composto.

L'ambiente Simple CLI è fornito soltanto per coprire quei sistemi operativi, ad esempio *MacOS 9*, che non sono dotati di ambiente a linea di comando.

- **Explorer:** è l'ambiente in genere più usato, infatti, con esso è possibile caricare insiemi di dati, visualizzare graficamente la distribuzione dei valori degli attributi, fare statistiche su di essi, effettuare preprocessing, eseguire algoritmi di estrazione di RdA, clustering, etc. e visualizzare i risultati. Sostanzialmente copre tutte le funzionalità, e per quanto riguarda la visualizzazione dei risultati, espande le potenzialità di WEKA rispetto alla versione priva di GUI.
- **Experimenter:** è una versione batch dell'Explorer, e consente di impostare veri e propri esperimenti di DM. Ad esempio, è possibile effettuare una serie di analisi su vari insiemi di dati e con svariati algoritmi, ed eseguirle alla fine tutte insieme, rendendo attuabile un confronto tra i vari tipi di algoritmi, per determinare qual è il più adatto a risolvere uno specifico problema.
- **Knowledge Flow:** è una variante dell'Explorer, in cui le operazioni da eseguire si esprimono in ambiente grafico, disegnando un diagramma che indica il flusso della conoscenza. È infatti possibile selezionare varie componenti da una tavolozza, collegarle tra loro e creare quello che in genere viene detto data-flow, in quanto mostra il flusso di dati attraverso le varie componenti che abbiamo inserito nel sistema.
Knowledge Flow introduce un vantaggio rispetto all'Explorer, in quanto oltre alla manipolazione a gruppi, permette di manipolare i dati anche incrementalmente, grazie alla presenza di algoritmi appositi.

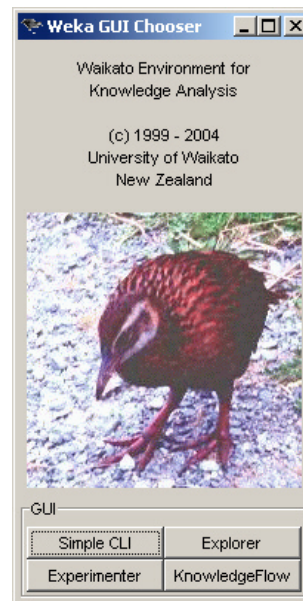


Figura 3.1 WEKA GUI Chooser.

La GUI di WEKA è quindi qualcosa di molto ampio, ma per quanto riguarda la nostra visione di query language KDD, ciò che si avvicina di più è sicuramente l'ambiente **Knowledge Flow**. Ci soffermiamo ora sulla descrizione delle principali caratteristiche e potenzialità offerte da tale interfaccia.

Il **Knowledge Flow** è stato l'ultimo ambiente ad essere inserito, quindi per stessa ammissione degli autori, che lo definiscono un “work in progress”, risulta ancora incompleto. Non tutte le funzionalità presenti nell'Explorer sono ancora disponibili, ciò comunque non intacca la nostra analisi, in quanto non basata sulle funzionalità presenti, ma sulla metafora grafica utilizzata.

Ci accingiamo quindi a darne una breve descrizione, sottolineando il fatto che tale esposizione è basata sulla versione del **Knowledge Flow** presente in WEKA 3.4.4 e rappresentata in Figura 3.2.

Una volta fatto partire l'ambiente, l'utente può selezionare i componenti WEKA da una toolbar posta in alto, posizzionarli in un layout, che ricopre gran parte della GUI, e connetterli insieme per creare un “knowledge flow” per processare ed analizzare i dati.

Com'è prevedibile, l'interfaccia fornisce la possibilità di inserire nel layout:

- **Data Sources:** in cui l'utente può scegliere vari formati di file sorgente, ad esempio l'operatore *ArffLoader*, che permette di leggere in input un file *arff*.
- **Data Sinks:** in cui l'utente può scegliere vari formati di file di destinazione, ad esempio *ArffSaver*.
- **Visualization:** dà la possibilità di scegliere il tipo di visualizzazione che si desidera per il tipo di conoscenza estratta.
- **Filters:** qui sono presenti una gran quantità di filtri, sia “supervised” che “unsupervised”, caratteristica fondamentale in sistemi di knowledge discovery, attraverso cui si riduce la quantità di dati da analizzare, rendendo talvolta fattibili analisi altrimenti di difficile attuazione. Non meraviglia, quindi, il gran numero di operatori a nostra disposizione.
- **Evaluation:** qui sono presenti una serie di operatori che permettono di valutare le performance di un learner, definire training set, test set, ed attuare operazioni ad essi associate. Ad esempio, *TrainingSetMaker* permette di definire un training set, a partire da un generico data set.
- **Classifiers-Clusterers:** dà la possibilità di inserire una serie impressionante di algoritmi per il mining. Solo per citarne alcuni, abbiamo algoritmi per costruire alberi di decisione, cluster, etc.

Una volta visti i componenti a nostra disposizione, non è difficile capire come avviene la costruzione del nostro flusso di conoscenza. Non entrando troppo in dettaglio, diciamo che la costruzione avviene procedendo iterativamente a:

- Selezionare l'oggetto (operatore-algoritmo), cliccandoci col mouse.
- Inserirlo nel layout.
- Configurarlo (cliccando col tasto destro sull'oggetto, appare un menù a tendina, attraverso il quale è possibile vedere le opzioni previste per l'oggetto).
- Connetterlo ad altri oggetti presenti nel layout.

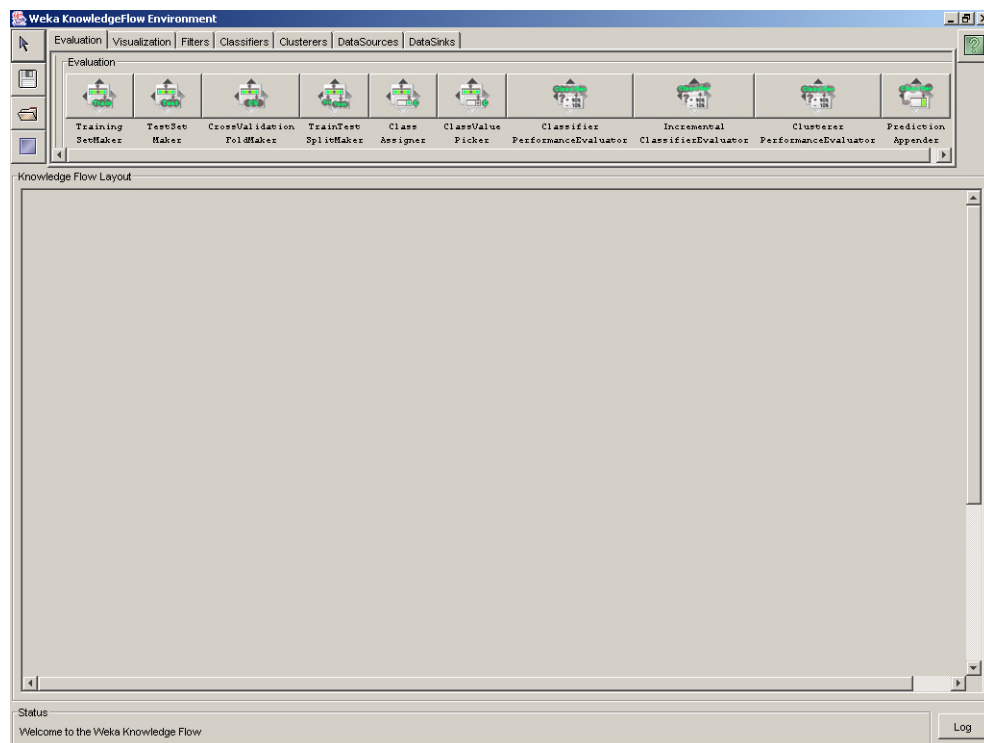


Figura 3.2 Ambiente Knowledge Flow di WEKA.

Tale costruzione viene in parte guidata dal sistema, per evitare situazioni incongruenti. Il software infatti evita la possibilità di connettere tra loro due nodi incompatibili.

Una volta terminata la costruzione, ci troviamo davanti un grafo orientato, i cui nodi sono operatori-algoritmi, e gli archi, etichettati, rappresentano il flusso dei dati tra i vari nodi del grafo. A questo punto possiamo far partire l'esecuzione, andando sull'oggetto **Data Sources**, che rappresenta l'input del nostro flusso.

Quindi, riassumendo sinteticamente le caratteristiche del **Knowledge Flow**, possiamo affermare che esso rappresenta innanzitutto un'interfaccia intuitiva per costruire data-flow. Permette inoltre di:

- processare flussi di dati in parallelo, dove ad ogni flusso è associato un proprio thread.
- vedere i modelli prodotti dai classificatori, per ogni gruppo di dati.

- visualizzare le performance dei classificatori, durante l'esecuzione del processo di KDD.

Dal punto di vista della rappresentazione, questa sembra essere una metafora piuttosto potente. Il grafo suddetto, infatti, rappresenta, in modo a nostro avviso adeguato, ciò che noi intendiamo come una possibile metafora grafica di una query KDD. Osservando il flusso dei dati, ci si rende conto di quanto un processo KDD possa essere complesso, e di come sia importante garantire la composizionalità, intesa sia a livello delle fasi, dove l'output della fase di preprocessing può rappresentare ad esempio l'input della fase di mining, sia a livello di algoritmo, ed in questo caso l'output di un algoritmo può diventare l'input di quello da applicare successivamente.

La GUI permette, oltre alla visualizzazione e/o al salvataggio dei risultati intermedi del flusso, per eventuali utilizzi futuri, anche di salvare/caricare (e quindi successivamente modificare) un intero grafo. Le operazioni svolte sull'interfaccia si riflettono sul prompt dei comandi, attraverso il quale è possibile vedere cosa il sistema invoca di volta in volta, e cosa realmente succede durante l'esecuzione del processo KDD.

Il file rappresentante il grafo che viene salvato non contiene la query in un formato cosiddetto "di scambio", ad esempio KDDML utilizza XML come rappresentazione della query KDD. Questo è dovuto soprattutto al fatto che nel file salvato in formato proprietario, sono contenute anche informazioni a livello grafico, come le posizioni degli oggetti nel layout. Per il resto, essendo la GUI un'estensione del sistema WEKA, la parte di applicazione di algoritmi si riduce a chiamate ai moduli JAVA che li implementano.

Per quanto riguarda la conoscenza estratta attraverso il **Knowledge Flow**, non è possibile, almeno nella versione attuale, salvare i modelli, cosa che è invece possibile con **Explorer**. Si ritiene comunque probabile che tale feature venga introdotta nelle prossime versioni.

Un punto di forza di qualsiasi sistema di DM dovrebbe essere l'espandibilità, ottenuta senza dover andare a toccare il core del sistema. Questo è uno dei problemi principali, quando si progetta qualsiasi sistema, ma nel caso di tool KDD risulta

ancora più delicato. Sta ai progettisti riuscire a trovare un metodo che sia il più semplice possibile per permettere all'utente finale di espandere il sistema anche con propri operatori e algoritmi.

Per quanto riguarda WEKA utilizzato senza GUI, l'espandibilità è ottenuta aggiungendo al sistema una nuova classe JAVA, che prevede di essere inserita in package già esistenti, e con interfacce e metodi predefiniti. Per una trattazione completa su come espandere il sistema, utilizzando completamente la gerarchia delle classi WEKA fare riferimento a [WF00]. In [WF00] è inoltre presente una trattazione generale ed introduttiva del sistema per il suo utilizzo da riga di comando o in applicazioni proprie JAVA.

Per quel che riguarda l'espandibilità della GUI, il materiale a nostra disposizione è piuttosto scarso, e si presume che ciò sia dovuto alla relativa "gioventù" dell'interfaccia.

Tutta la gestione dell'interfaccia, anch'essa in JAVA, è contenuta in una serie di package. Pertanto, andando a toccare i file di configurazione, si riesce a far vedere i propri algoritmi, inseriti precedentemente nelle classi WEKA, anche all'interno dell'Explorer. Questo è possibile, poiché in Explorer la scelta, ad esempio, di un classificatore viene fatta attraverso un menù per utenti *Microsoft Windows* simile ad "esplora risorse", in cui si possono vedere tutti i classificatori attualmente implementati nel sistema.

Ritornando all'ambiente **Knowledge Flow**, quest'ultimo a differenza dell'ambiente Explorer che fa uso di JAVA Swing [SUN], è sviluppato con JAVA BEANS [SUN].

C'è inoltre da notare che non è presente alcun *wizard* che guidi l'utente nella costruzione della sua query. Quest'ultimo è lasciato a sé stesso, cosa che non accade nell'Explorer, che anche se in minima parte, guida implicitamente l'utente, disabilitando le operazioni che non possono essere eseguite.

Inoltre la struttura è piuttosto rigida, l'interfaccia, almeno per ora, non si aggiorna automaticamente in base agli operatori-algoritmi presenti nel sistema, ma permette all'utente di utilizzare solo quelli che sono presenti nella task-bar in alto, e non è previsto nessun metodo, visuale o non, per inserirne di nuovi. Questa rappresenta

una grossa limitazione per quanto riguarda l'uso dell'interfaccia **Knowledge Flow**. Essendo però il sistema completamente open source, è modificabile in ogni sua parte, e quindi andando “a basso livello” è possibile personalizzarlo totalmente. Questa è una strada comunque non percorribile, in quanto, pur avendo ottime conoscenze di JAVA, modificare il sistema grafico comporterebbe un notevole sforzo quasi mai praticabile, da un analista di DM.

Concludendo, possiamo dire che WEKA permette di essere utilizzato nei propri programmi o da solo, ampliato con propri operatori o algoritmi nella versione priva di interfaccia grafica, mentre nell'altro caso l'estensione è in parte possibile nell'Explorer, ma bisogna “sporcarsi le mani” con la struttura interna del sistema grafico.

Per quel che riguarda l'ambiente Knowledge Flow, esso offre una buona metafora, ma non fornisce alcuno strumento per guidare l'utente nella costruzione di query.

Nella nostra trattazione abbiamo volutamente tralasciato spiegazioni anche minime sul funzionamento dell'Explorer in quanto esso, a nostro avviso, pur essendo parte integrante del sistema, non mette in risalto la natura del processo KDD come flusso di dati tra varie fasi.

3.2.2 ESEMPIO DI UTILIZZO DI WEKA

Introduciamo ora un semplice esempio di quella che può essere una possibile applicazione di WEKA.

Avviamo WEKA Knowledge Flow, e per prima cosa scegliamo i dati da utilizzare come training set per la costruzione di un albero di classificazione. A tal fine, abbiamo già precedentemente preparato un data set, che non è altro che una modifica del noto *Iris* data set [Fis36]. Come mostrato in Figura 3.4, abbiamo aggiunto l'operatore *ArffLoader* dal tab *DataSources*, per la lettura del file. Cliccando con il tasto destro su tale operatore, comparirà un menù attraverso il quale sceglieremo il file *arff* da importare nel nostro esempio, come mostrato in Figura 3.3. Lo stesso operatore sarà utilizzato anche per caricare il test set.

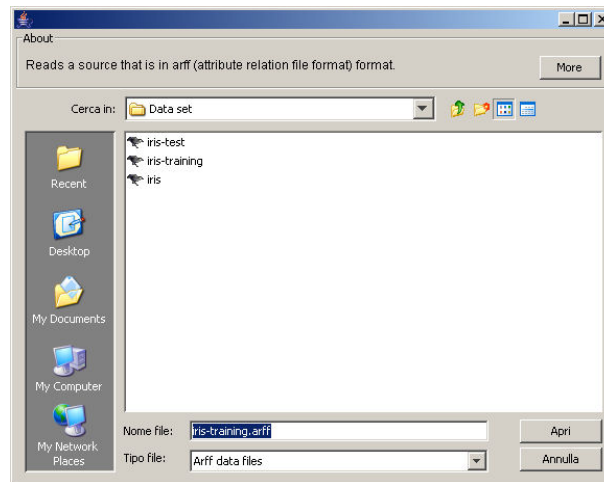


Figura 3.3 Scelta del data set da utilizzare come training set.

Abbiamo inserito anche un operatore per visualizzare il contenuto del file. Tale operatore, selezionato dal tab *Visualization*, si chiama *TextViewer*. Inserito quest'ultimo, non rimane che connetterlo con *ArffLoader*. Per compiere tale operazione, ci spostiamo sul nodo di origine della connessione -in questo caso *ArffLoader*-, e cliccando con il tasto destro del mouse, sul nodo compare il menù precedentemente citato, attraverso il quale è possibile vedere le connessioni esportate da tale nodo.

In questo caso tutto il data set viene spedito al *TextViewer*.

Una volta selezionato il tipo di connessione scelta, vengono evidenziati, se presenti, i nodi compatibili con tale connessione, e semplicemente cliccando sul nodo di destinazione, si crea la connessione (Figura 3.4).

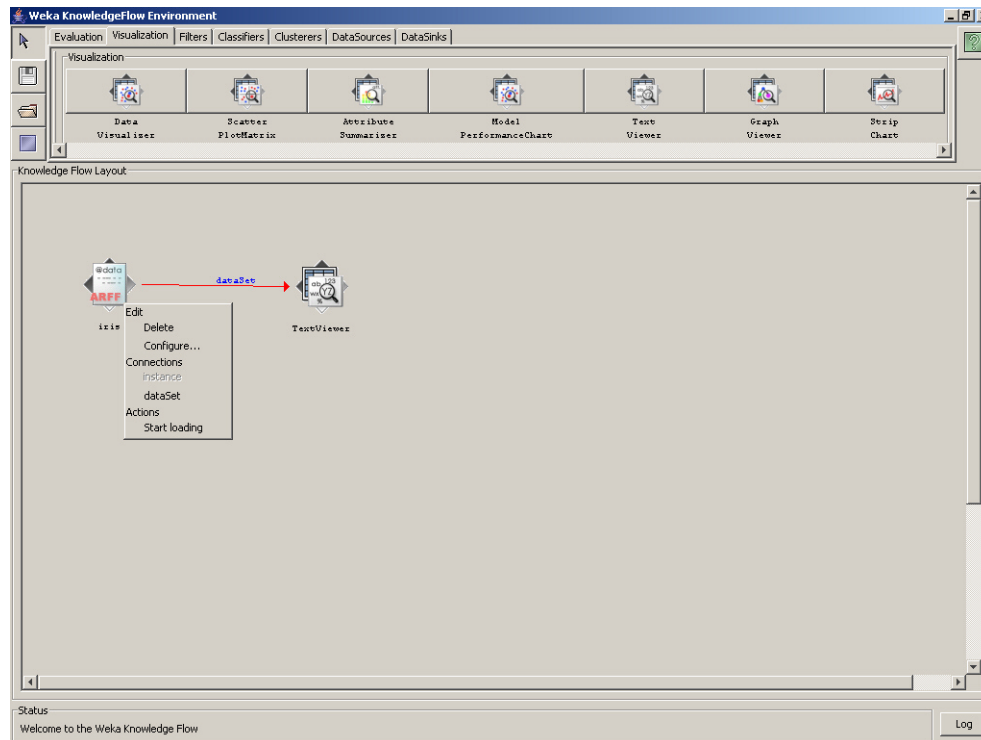


Figura 3.4 Inserimento e collegamento degli operatori per caricare e visualizzare il data set.

Dopo aver creato la connessione, possiamo vedere il file in input, mandando in esecuzione l'esperimento attraverso il nodo *ArffLoader*, e sempre dal menù precedentemente descritto, utilizzare la voce *Start Loading*. Fatto ciò, basterà andare sul nodo *TextViewer*, e selezionare la voce *Show Result*.

Comparirà a questo punto una nuova finestra, divisa in due parti, in cui a sinistra abbiamo la possibilità di scegliere tra una serie di *Result List*, ossia vari output inviati al *TextViewer* da uno o più nodi, come vedremo tra breve, mentre a destra viene mostrato il risultato. In questo caso si ha la visualizzazione del contenuto del file in input.

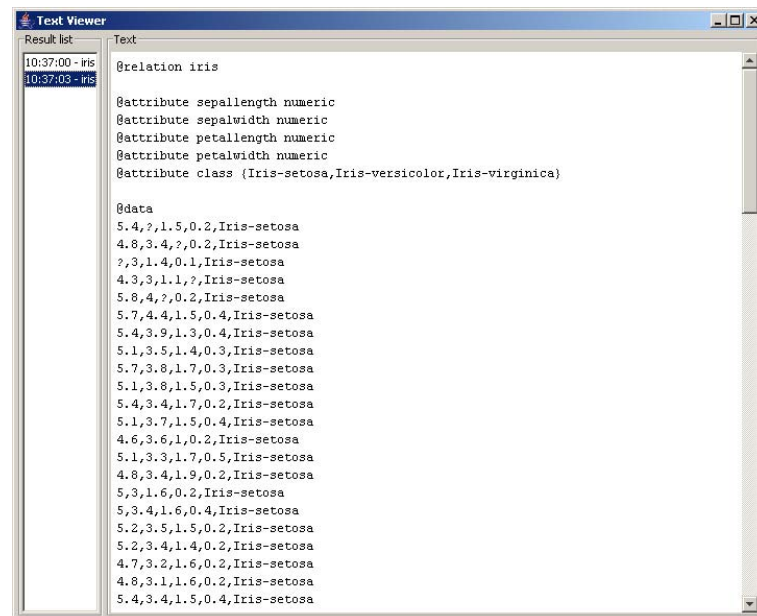


Figura 3.5 Visualizzazione dei dati scelti come training set.

Dalla finestra rappresentata in Figura 3.5, è facile notare che il data set scelto per la costruzione del nostro albero di classificazione presenta dei valori mancanti, rappresentati da punti interrogativi. Questo porta all'introduzione di un passo di preprocessing su tale data set, per renderlo adatto all'algoritmo di costruzione di tale albero⁷.

Ripetendo i passi precedentemente descritti, proseguiamo con l'introduzione ed il collegamento dell'operatore *ReplaceMissingValue*, dal tab *Filters*, che integra i dati mancanti con la media calcolata sui dati presenti per uno specifico attributo. Andando ad eseguire i passi sopra descritti, attraverso il nodo *TextViewer*, è possibile osservare come vengono integrati i valori mancanti nel training set.

Inseriamo adesso l'operatore *TrainingSetMaker* dal tab *Evaluation*, che ci permette di definire il training set vero e proprio da utilizzare per la costruzione del modello⁸. Inseriamo inoltre gli operatori per leggere il file che verrà utilizzato come test set del nostro albero estratto, insieme all'operatore per definire il test set, tutti con le opportune connessioni (Figura 3.6).

⁷ In realtà l'algoritmo funziona ugualmente, ma potrebbe, a nostro avviso, ignorare le ennuple con valori mancanti.

⁸ È richiesto inoltre un operatore (*ClassAssigner*) che definisca quale colonna del file rappresenta la classe di predizione. Se non presente, come in questo caso, il sistema prevede che tale colonna sia l'ultima presente nel file in input.

Come si può notare, è stato inserito anche l'operatore *J48* dal tab *Classifiers*, che implementa l'algoritmo di costruzione di alberi di decisione C4.5 (par. 1.4.2), il quale mantiene i parametri di default.

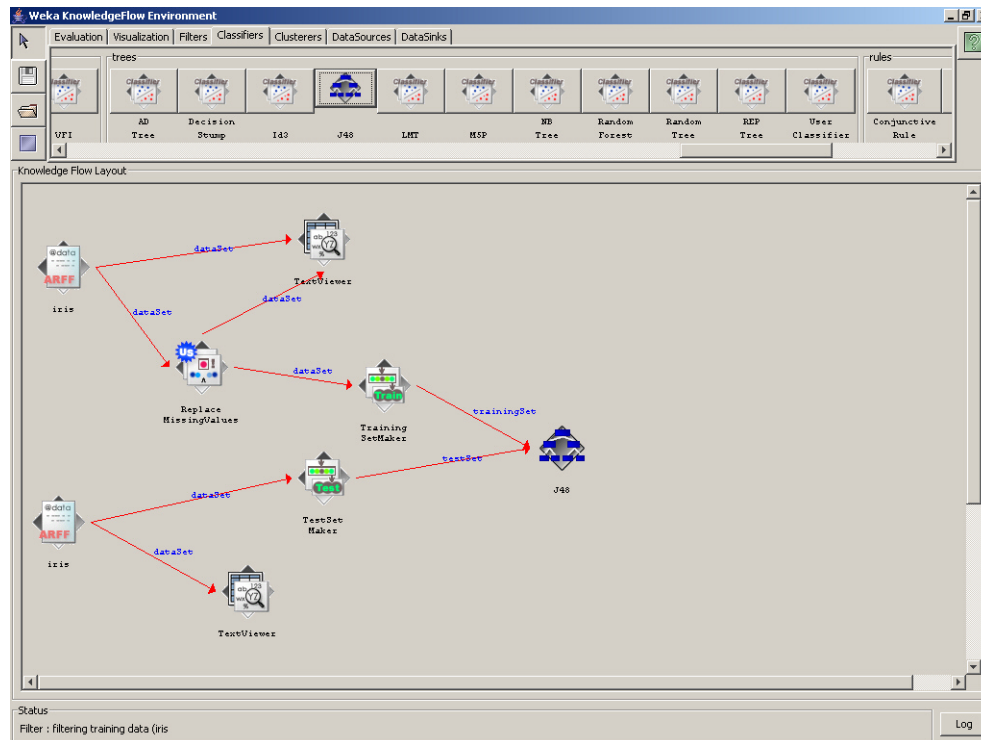


Figura 3.6 Inserimento degli operatori per la costruzione dell'albero di classificazione.

Inseriamo infine una serie di nodi per visualizzare la conoscenza estratta e valutare l'accuratezza del modello ottenuto.

A tal fine, utilizziamo l'operatore *ClassifierPerformanceEvaluator* dal tab *Evaluation*, che fornisce statistiche sulla qualità del modello estratto, validate sul test set. Inoltre inseriamo un operatore per aggiungere nel test set la classe predetta dal modello (*PredictionAppender*). Alla fine otteniamo il flusso di dati mostrato in Figura 3.7.

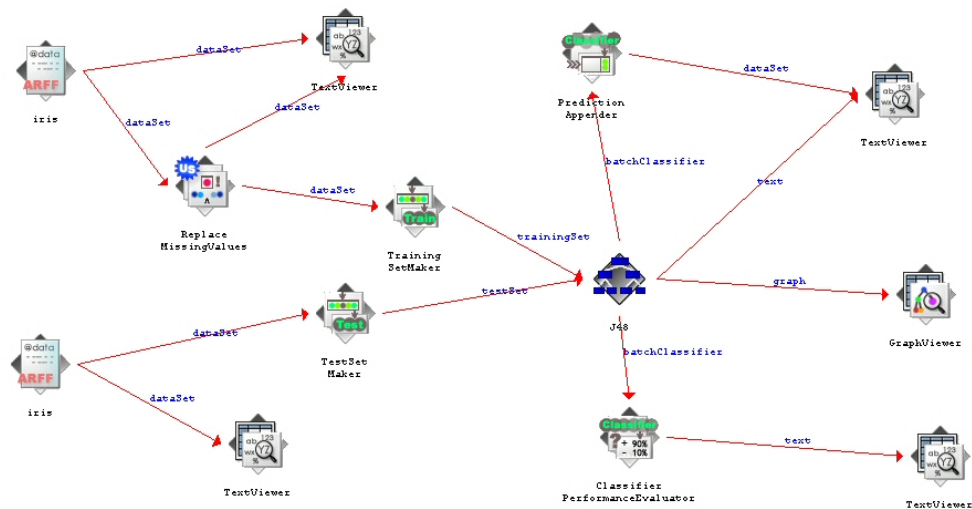


Figura 3.7 Flusso finale dei dati.

Abbiamo infine inserito tutti i nodi, per permettere la visualizzazione delle informazioni che riteniamo utili. Possiamo adesso mandare in esecuzione, andando a visualizzare ciò che ci interessa, come ad esempio l'albero di classificazione (Figura 3.8), oppure la classe predetta nel test set, o una serie di statistiche sull'accuratezza del modello estratto applicato al test set.

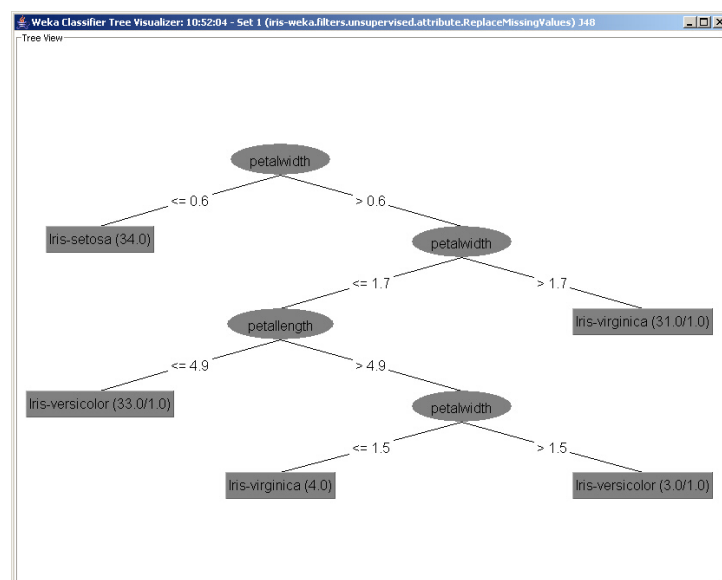


Figura 3.8 Albero di classificazione estratto.

3.3 YALE

Proseguiamo la nostra analisi, andando a considerare un altro sistema chiamato YALE (Yet Another Learning Environment) [YALE] nella versione 2.4.1. Il sistema, sviluppato dalla Artificial Intelligence Unit dell'Università di Dortmund, è free ed open source, ed è completamente implementato in JAVA, con l'utilizzo di JAVA Swing per la parte grafica.

YALE viene applicato per risolvere compiti di machine learning e knowledge discovery in una serie di domini che vanno dalla generazione alla selezione di caratteristiche fino alla gestione del *concept drift*. Oltre ai domini appena menzionati, altre possibili applicazioni del sistema includono anche il preprocessing, il clustering, il text processing e la classificazione. Esistono inoltre alcuni plug-in che forniscono operatori non presenti nella versione standard del sistema, che aggiungono funzionalità per specifici compiti di apprendimento.

I processi KDD sono spesso visti come una concatenazione sequenziale di operatori. In molte applicazioni, questa visione non è sufficiente a modellare il processo, e di conseguenza vi è la necessità di poter annidare le varie catene di operatori, che si presentano essenzialmente come alberi.

Questa osservazione è la base concettuale di YALE, il quale prevede che le foglie dell'albero di un processo KDD corrispondano a semplici passi nel processo modellato. I nodi interni invece corrispondono a passi più complessi o astratti del processo, mentre la radice dell'albero rappresenta l'intero processo KDD. Questi alberi in YALE si chiamano *esperimenti*, e sono costituiti da una catena di operatori in alcuni casi annidati, in cui ogni operatore consuma una serie di oggetti in input e produce alcuni oggetti in output.

Gli operatori definiscono gli input richiesti, gli output forniti e i parametri, sia obbligatori sia opzionali, permettendo in questo modo a YALE di controllare automaticamente l'annidamento ed i tipi di oggetti che vengono scambiati.

Una cosa importante da dire è che gli oggetti in input di un operatore possono essere utilizzati, o se non sono richiesti, passati direttamente agli operatori successivi. Questo aumenta la flessibilità del sistema, facilitando la corrispondenza

tra le interfacce di operatori consecutivi, e permettendo di passare oggetti da un operatore all'operatore target attraverso una serie di passaggi.

Generalmente gli oggetti che vengono passati tra gli operatori sono insiemi di dati, modelli di predizione, vettori di valutazione, etc. Gli operatori possono aggiungere informazioni agli oggetti in input, tipicamente etichette per oggetti precedentemente non etichettati, o nuove caratteristiche e produrre oggetti "estesi".

YALE utilizza XML, adatto per descrivere gli alberi di operatori che modellano processi KDD. Se usato attraverso la command-line, basterà invocare il sistema sottoponendogli un *esperimento*, che non è altro che un file XML, con cui, attraverso specifici tag, vengono definiti gli operatori (ed i loro parametri) e il loro annidamento.

Per una trattazione completa su come costruire il file XML, gli operatori o gli algoritmi presenti in YALE fare riferimento ai documenti presenti in [YALE].

A questo punto ci interessa come la GUI di YALE può essere utilizzata per la costruzione dell'albero degli operatori (*esperimento*), la rappresentazione che la stessa produce del processo KDD, e la possibilità di controllare e supervisionare interattivamente gli *esperimenti* in corso.

3.3.1 YALE GUI

Introdotta, anche in questo caso, solo nelle ultime versioni del sistema, la GUI (Figura 3.9) si prefigge il compito di facilitare l'uso di YALE, evitando all'utente di andare a manipolare direttamente i file XML.

Facendo partire l'applicazione, per prima cosa compare un "welcome screen", attraverso il quale l'utente può scegliere tra cinque possibilità:

- Aprire un *esperimento* esistente
- Aprire uno degli *esperimenti* recenti elencati
- Creare un nuovo *esperimento* vuoto
- Utilizzare il *Wizard*
- Far partire il tutorial on-line

Ai fini della nostra trattazione alcune delle voci elencate risultano irrilevanti, per questo motivo ci soffermeremo solo sulle ultime due voci presenti.

Il tutorial on-line è un utile mezzo, attraverso il quale l'utente che accede per la prima a YALE prende confidenza con i primi concetti basilari per l'uso della GUI, e con i diversi operatori forniti dal sistema.

Il *Wizard* guida l'utente durante il processo di creazione di un nuovo *esperimento*. Si inizia selezionando da una lista un *esperimento* “template”, che funge da schema generale su cui modellare il proprio *esperimento*.

Per essere del tutto generali, supponiamo di far partire la GUI con un *esperimento* vuoto. Ci troviamo di fronte ad una finestra principalmente divisa in due parti, e nella cui parte bassa è presente un *message box*, che registra l'attività del sistema. Questa finestra è utile, quando si avviano esperimenti complessi che richiedono, ad esempio, più volte l'applicazione di una serie di algoritmi, come la tecnica della cross-validation. Attraverso il box, infatti, è possibile seguire l'evoluzione dell'*esperimento*, osservando i messaggi in output visualizzati nel *message box*. Questo output può anche essere memorizzato in un file di *log*.

Gran parte della GUI è lasciata invece ad una serie di tab, grazie ai quali è possibile avere viste differenti della stessa query, vedere i risultati, sia intermedi che finali e monitorare l'attività del sistema e lo stato di avanzamento dell'*esperimento*, basandosi sull'occupazione della memoria e l'utilizzo della CPU.

Nella parte alta della finestra, per completezza, si segnala la presenza di bottoni, utili per aprire, salvare, eseguire, compilare un *esperimento*, e i classici menù a tendina, come *File*, che completano il sistema, fornendo l'accesso alle funzionalità più “nascoste”, ad esempio *Settings*, che permette di configurare vari aspetti del sistema.

Senza dilungarci troppo, andiamo ad esaminare le feature a nostra disposizione, per la costruzione e rappresentazione dell'*esperimento*, ovvero i seguenti tab:

- **Tree.** Yale parte dal concetto che la naturale rappresentazione di un *esperimento* KDD sia assimilabile ad un albero. Attraverso questa vista è possibile costruire tale albero, inserendo i vari operatori. Partendo dal nodo

Root, sempre presente, semplicemente cliccando sul nodo, si accede ad un menù a tendina che permette l'inserimento, la sostituzione, la cancellazione e la disabilitazione di un operatore. La costruzione dell'albero risulta quindi piuttosto intuitiva: si decide il nodo padre dell'operatore da inserire e si procede ad inserirlo, utilizzando la procedura precedentemente descritta. Nel caso dell'eliminazione, è necessario selezionare il nodo da eliminare e cancellarlo, prestando attenzione al fatto che l'eliminazione di un nodo comporta la necessaria eliminazione dell'intero sottoalbero ad esso associato. Lo stesso procedimento vale nel caso della disabilitazione dell'operatore, mentre in quello della sostituzione, il sottoalbero rimane immutato. Eventuali sostituzioni erronee impediranno comunque la validazione dell'*esperimento*. Un'altra caratteristica interessante dei vari operatori inseriti riguarda il fatto che, cliccandoci sopra con il tasto destro del mouse, compare alla destra della *treeview* una tabella con due colonne, *Key* e *Value*, grazie alle quali è possibile settare i vari parametri, tra i quali vengono evidenziati quelli obbligatori. Attraverso questa tabella, è inoltre possibile accedere anche ad un *Attribute Editor*, per modificare i vari insiemi di dati in input. Come ultima caratteristica, vorremmo sottolineare la possibilità di rinominare i nodi dell'albero a proprio piacimento, rendendo ancora più chiari i vari compiti associati ai nodi (l'algoritmo associato al nodo ovviamente resta immutato).

- **XML.** Come abbiamo visto prima, ogni *esperimento* in YALE è rappresentato da un file, opportunamente redatto in XML. Questa vista non fa altro che mostrare il file XML, che sarà quello effettivamente utilizzato dal sistema, relativo all'albero costruito nella *treeview*. Tuttavia, nessuno vieta all'utente di modificare o scrivere l'*esperimento*, direttamente maneggiando il file XML attraverso questa vista. È da notare, comunque, che le modifiche apportate al file si ripercuotono immediatamente anche sulle altre due viste presenti.

- **Experiment.** Attraverso questa finestra, che a differenza delle precedenti, è solo una finestra di output e non permette di modificare la query, è possibile vedere il *nesting* degli operatori presenti nel nostro *esperimento*. Il tutto è rappresentato attraverso una serie di box annidati uno dentro l'altro.

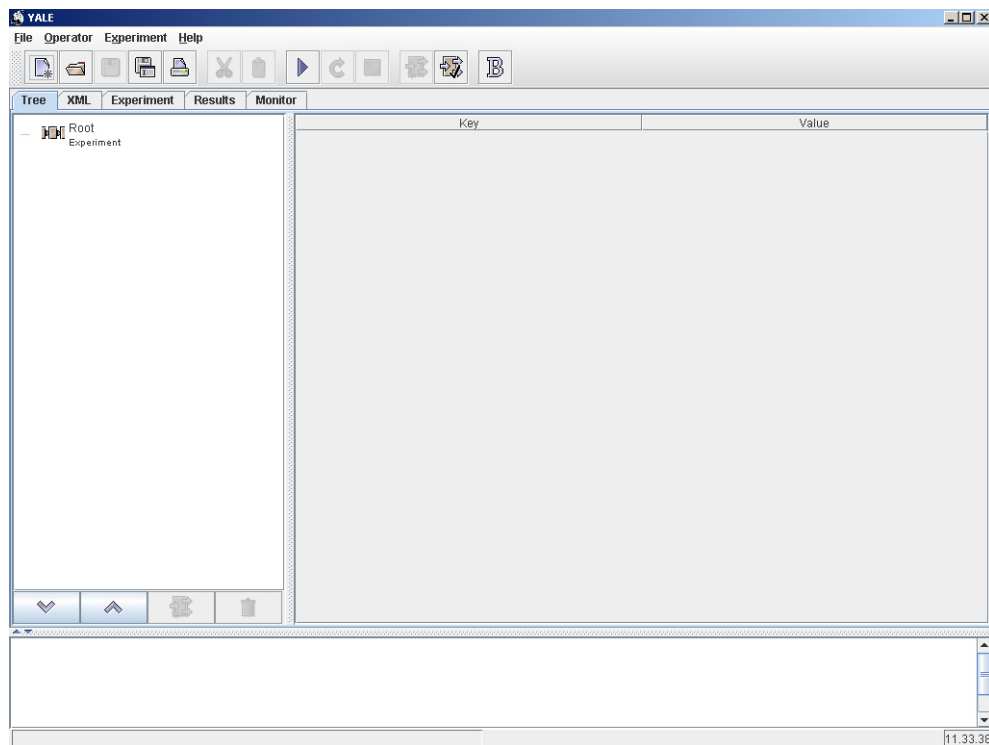


Figura 3.9 YALE GUI.

Una volta costruito il nostro esperimento è possibile:

- Validare l'*esperimento* (*Validate Experiment*). Questa funzione controlla se tutti gli operatori sono annidati correttamente ed hanno tutti i loro parametri corretti. In caso contrario, la presenza di eventuali errori viene segnalata nel *message box*. Questa possibilità di controllo statico risulta utile, perchè evita di eseguire l'*esperimento*, e ciò in caso di *esperimenti* complessi potrebbe comportare una notevole perdita di tempo. Bisogna precisare che tale controllo, essendo statico, garantisce solo la correttezza dell'*esperimento* relativamente ai tipi e la sintassi, ma non dà garanzie sulla semantica.

- Eseguire (*Run*). Tale funzione permette di eseguire l'*esperimento*. Bisogna notare, che anche in questo caso, il sistema effettua la fase di validazione anticipatamente all'esecuzione.

Quando un *esperimento* è finito, il suo risultato sarà presentato nella finestra **Results**. Attraverso di essa, è possibile visualizzare statistiche di performance, alberi di decisione, grafici a due e a tre dimensioni. Grazie all'utilizzo di break-point, inseribili prima o dopo qualsiasi operatore o catena di operatori, è possibile visualizzare anche i risultati intermedi delle varie fasi.

Per quanto riguarda la metafora grafica utilizzata, YALE ha una concezione molto simile a quella di KDDML, infatti il processo KDD, rappresentato attraverso il file XML, può essere visto come un albero di operatori. In realtà YALE estende tale concetto, aggiungendo anche la visualizzazione dell'annidamento degli operatori, infatti dal punto di vista prettamente grafico, l'albero degli operatori di YALE appare molto stilizzato. Un utente potrebbe perdere molto tempo, nel caso di alberi complessi, per riuscire a capire il grado di annidamento di un certo operatore. La scelta di YALE di affiancare alla *treeview* anche questa vista appare quindi ottima, mettendo in luce il fatto che forse l'albero da solo non è sufficiente, almeno fino a che l'utente non abbia raggiunto una certa confidenza con il sistema, a rappresentare la query ed il flusso di dati attraverso gli operatori presenti in essa.

Senza ripeterci troppo, YALE permette di salvare, modificare, caricare esperimenti, modelli e risultati. Tranne che per gli *esperimenti*, non si rileva l'utilizzo di file in formati di interscambio o leggibili anche al di fuori del sistema. Un caso interessante è che YALE al suo interno permette di utilizzare tutti gli algoritmi WEKA. Il modello generato da un algoritmo WEKA può essere anche salvato in YALE, ma in questo caso non può essere più ricaricato da WEKA (almeno non nell'ultima versione di quest'ultimo).

Per quanto riguarda il delicato problema dell'estensione del sistema, YALE supporta l'implementazione di operatori definiti dall'utente. Per implementare un operatore, l'utente deve semplicemente definire gli input richiesti, l'output generato,

i parametri obbligatori e opzionali ed il core funzionale dell'operatore. Il tutto ovviamente deve avvenire secondo regole ben precise, dettate dal tipo di operatore che si vuole inserire. Ad esempio, se l'operatore è generico, o prevede l'utilizzo di altri operatori interni, i metodi previsti dall'interfaccia di quella classe di operatori dovranno essere implementati dall'utente in JAVA.

Una caratteristica particolarmente interessante e importante è che, fornendo la descrizione dell'operatore inserito in un opportuno file XML, YALE creerà automaticamente tutti gli elementi grafici corrispondenti nella GUI, grazie ai quali sarà possibile inserire l'operatore appena definito nei propri esperimenti.

Questo rappresenta senza dubbio un punto di forza del sistema, in quanto, oltre ad avere una facile espandibilità a livello più basso, permette all'utente di personalizzare anche la GUI con i propri operatori. La facilità di espansione è testimoniata anche da una serie di plug-in disponibili sul web, che garantiscono a YALE una versatilità e duttilità non comuni.

Come è prevedibile, YALE permette inoltre di essere facilmente invocato anche attraverso altre applicazioni JAVA definite dall'utente. Per una trattazione completa e approfondita di questi argomenti, fare riferimento alla documentazione *API* (*Application Program Interface*) ed a YALE tutorial disponibile presso la home page di YALE [YALE].

Concludendo, possiamo affermare che la GUI di YALE, pur essendo un sistema molto complesso, grazie al tutorial on-line ed al *Wizard*, permette di essere sfruttata in breve tempo anche da chi non ha molta familiarità con questi sistemi, mentre un utente già esperto può manipolare direttamente il file XML ed avere al tempo stesso una serie di utili informazioni su ciò che sta costruendo. A questo proposito ricordiamo la presenza di una funzione di validazione dell'*esperimento*, che guida l'utente nella localizzazione di eventuali errori commessi nella stesura-costruzione dello stesso.

Pertanto, in questo caso siamo in presenza di una vera e propria metafora grafica, che non incrementa, né limita le funzionalità del sistema, ma rappresenta solo una visione diversa, ad un livello superiore, di ciò che è la costruzione e la visualizzazione di una query, o in questo caso un *esperimento* di knowledge discovery.

3.3.2 ESEMPIO DI UTILIZZO DI YALE

Forniamo ora un esempio del funzionamento di YALE. Anche in questo caso abbiamo scelto la costruzione di un albero di classificazione. Una volta avviato, come mostrato in Figura 3.10, YALE fornisce la possibilità di scegliere tra aprire un *esperimento* esistente, utilizzare il *Wizard*, per avere un'intelaiatura predefinita del nostro *esperimento*, oppure partire con un *esperimento* vuoto. Nel nostro caso optiamo per quest'ultima possibilità.

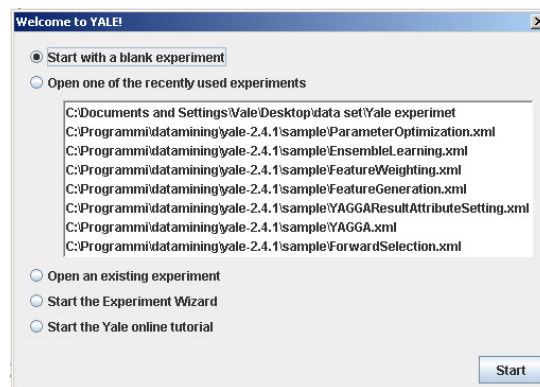


Figura 3.10 Welcome screen di YALE.

Il passo successivo è quello di importazione dei dati su cui estrarre il nostro modello. Tali dati, sia training set che test set, sono gli stessi visti nell'esempio di WEKA, quindi il training set presenta dei valori mancanti. A tal fine, aggiungiamo nell'albero degli operatori, che contiene per ora solo il nodo radice, un operatore chiamato *ExampleSource*, che possiamo rinominare a nostro piacimento. Tale operatore è raggiungibile attraverso un menù, che compare cliccando con il tasto destro del mouse, sul nodo padre di quello che si vuole inserire. In questo caso lo si seleziona accedendo, in ordine, alle voci *New Operator*, *IO* ed *Example*. Una volta completata questa operazione, è possibile configurare l'operatore appena inserito, attraverso una serie di parametri che compaiono sulla parte destra dell'applicazione. Attraverso questi parametri, dopo aver selezionato il file che contiene il data set, è possibile visualizzarlo e modificarlo tramite il pulsante *Edit* (Figura 3.11).

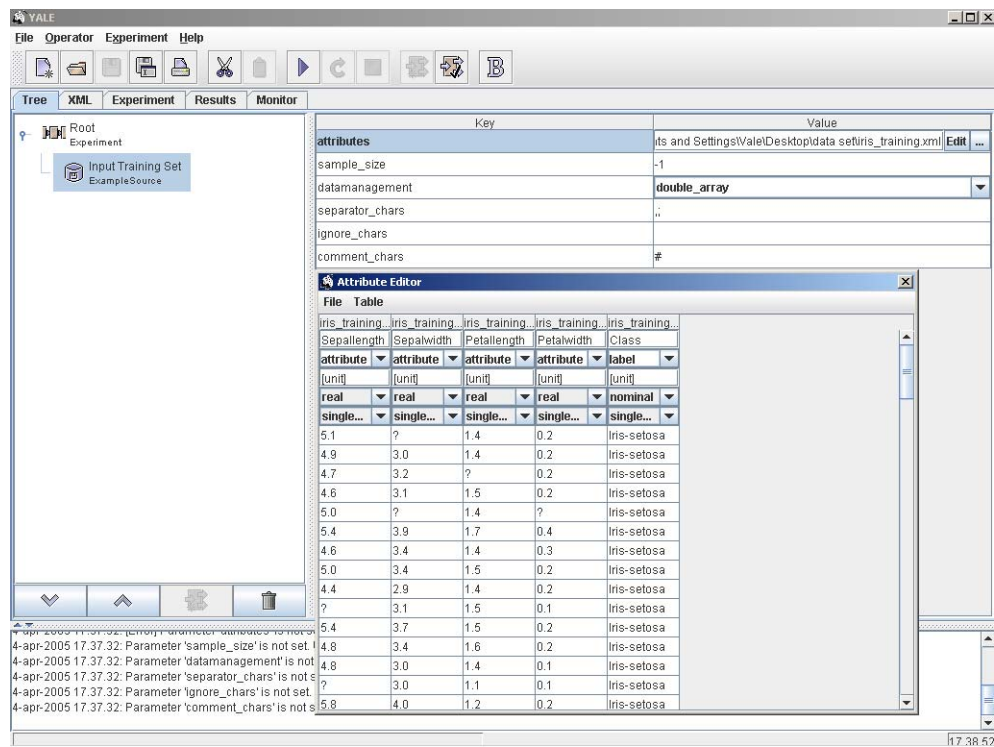


Figura 3.11 Caricamento e visualizzazione del training set.

A questo punto, inseriamo rispettivamente dalle categorie *Preprocessing* e *Learner*, gli operatori *MissingValueReplenishment*, per eliminare i valori mancanti, e *DecisionTreeLearner*, per costruire l'albero di classificazione, ottenendo l'albero degli operatori mostrato in Figura 3.12.

L'attributo *class* su cui costruire il modello predittivo è definito nei dati *iris_training.xml*, e non è necessario quindi l'inserimento di opportuni operatori per definirlo.

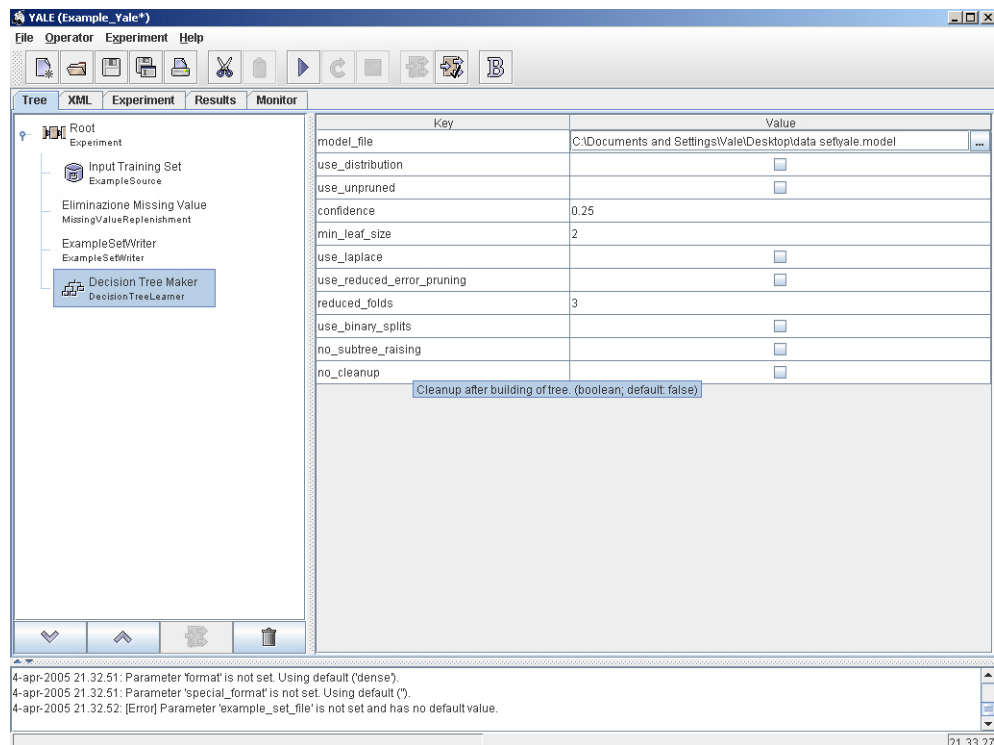


Figura 3.12 Inserimento di operatori per la costruzione dell'albero di classificazione.

È da notare in questo caso l'inserimento di un operatore per salvare il nuovo data set privo di valori mancanti in un file. Inoltre, al fine di riutilizzarlo al passo successivo, salviamo il modello estratto in un file specificato nella prima riga dei parametri del *DecisionTreeLearner*. Questo modello infatti verrà utilizzato al passo successivo, per attuare una classificazione su un test set definito esplicitamente. A tal fine, vengono inseriti gli operatori, per leggere il modello precedentemente estratto (*ModelLoader*), per caricare il test set e per applicare il modello (*ModelApplier*⁹).

Inoltre aggiungeremo un operatore per salvare il test set con la classe predetta in un file.

⁹ Per utilizzare tale modello, è necessario, prima di inserire gli operatori *ModelLoader* e *ModelApplier*, eseguire l'esperimento, al fine di permettere al nodo *DecisionTreeLearner* di generare il file contenente il modello estratto, che verrà successivamente utilizzato.

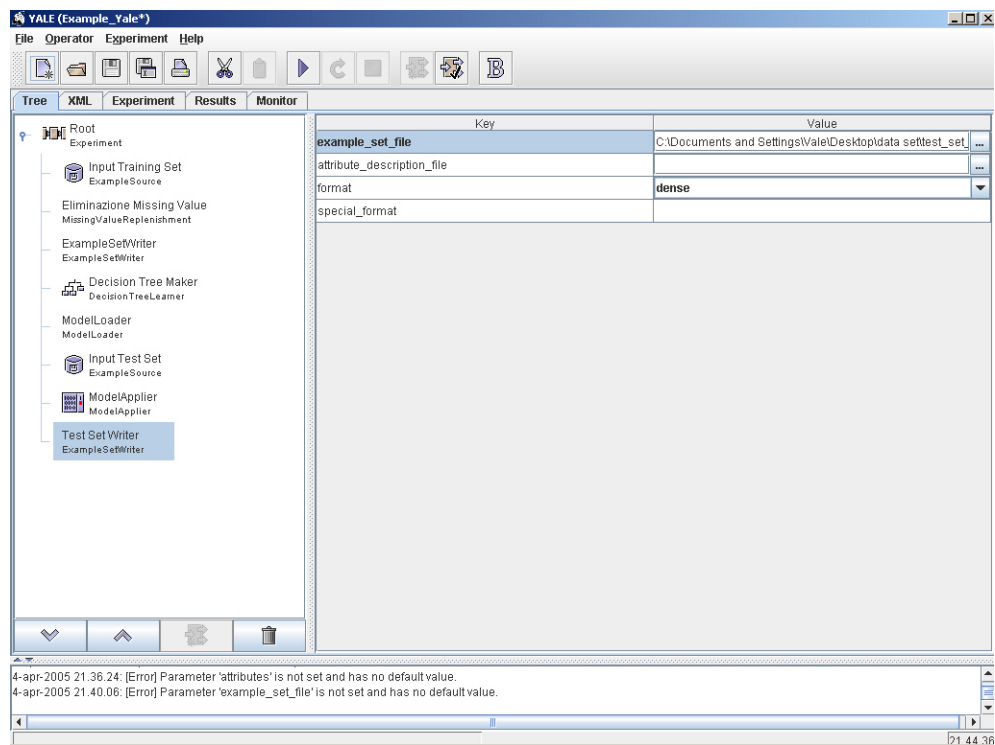


Figura 3.13 Vista finale del nostro *esperimento*.

YALE, oltre alla visualizzazione della catena di operatori (Figura 3.13), fornisce anche la possibilità di visualizzare ed editare a mano il file XML, che rappresenta la nostra query, ed offre inoltre informazioni sul *nesting* degli operatori (Figura 3.14).

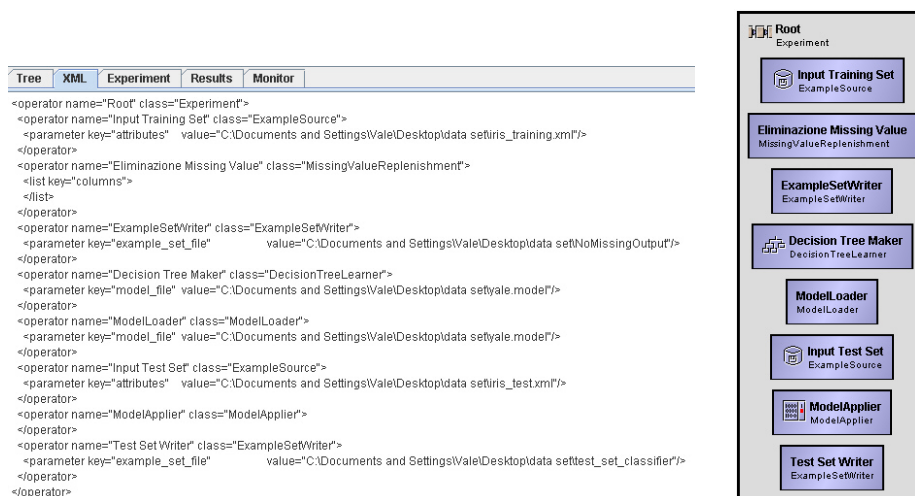


Figura 3.14 Versione XML del nostro *esperimento* e del diagramma di *nesting* degli operatori presenti.

Possiamo adesso eseguire l'*esperimento* e valutare gli output forniti dal sistema. Per fare ciò, è sufficiente andare su *Experiment* e scegliere *Run*, oppure utilizzare il bottone *Run* sulla barra degli strumenti. Se tutto è corretto, il sistema presenta gli output prodotti attraverso il tag *Results*.

Oltre all'albero di classificazione estratto, sono disponibili una serie di statistiche sul test set e sulla classe predetta per tale insieme (Figura 3.15). In più vi è la possibilità di accedere al file generato per visualizzare le caratteristiche delle ennuple erroneamente classificate.

SimpleExampleSet							
Number of examples: 45							
Number of attributes: 4							
Index	Name	Generated from	Unit	Type	Blocktype	Blocknr.	Values
Regular attributes							
0	Sepallength	Sepallength		real	single_value	-	range=4.4-7.6; avg=5.779999999999999 +/- 0.8590951311958734
1	Sepalwidth	Sepalwidth		real	single_value	-	range=2.3-3.9; avg=3.048888888888889 +/- 0.3636984234360536
2	Petalength	Petalength		real	single_value	-	range=1.3-6.6; avg=3.7533333333333334 +/- 1.7700721893627824
3	Petalwidth	Petalwidth		real	single_value	-	range=0.1-2.5; avg=1.2 +/- 0.7720103626247511
prediction							
5	prediction(Class)	prediction(Class)		nominal	single_value	-	Iris-setosa (15),Iris-versicolor (16),Iris-virginica (14)
label							
4	Class	Class		nominal	single_value	-	Iris-setosa (15),Iris-versicolor (15),Iris-virginica (15)

Figura 3.15 Statistiche sul test set.

3.4 ORANGE

Il sistema preso ora in esame si chiama ORANGE [ORANGE]. Sviluppato dall'AI Lab della Facoltà di Informatica, presso l'Università di Ljubljana in Slovenia, è realizzato sotto la General Programming License (GPL). Come tale è free, se usato in rispetto a tali termini.

Anche in questo caso, il sistema offre una serie di funzionalità che includono una larga varietà di algoritmi per il machine learning ed il DM, in più comprende una serie di routine per la manipolazione e l'input dei dati [DZ04].

La caratteristica che contraddistingue questo sistema da quelli esaminati fino ad ora è la sua filosofia costruttiva. ORANGE, qui analizzato in versione 0.9.5, infatti, a detta degli stessi autori, è stato attentamente concepito per garantire il miglior bilanciamento possibile tra velocità di esecuzione e velocità di sviluppo delle applicazioni. Questa considerazione ha portato gli sviluppatori ad implementare in

C++ [Str97] quelle componenti del sistema considerate “time-critical”, ed in Python [PYTHON] le restanti parti che svolgono un ruolo di connessione tra le suddette componenti.

Python è un linguaggio di programmazione, interpretato, interattivo e orientato agli oggetti, che viene spesso paragonato a linguaggi quali Tcl [TCL], Perl [PERL]. Combina in sé una notevole potenza con una sintassi molto chiara, ed è inoltre portabile su molte delle piattaforme più diffuse. Per una descrizione dettagliata del linguaggio fare riferimento al sito web [PYTHON].

Possiamo quindi affermare che, grazie alla sua strutturazione, ORANGE può essere visto come uno “scriptable environment” per una veloce prototipizzazione di nuovi algoritmi e soprattutto schemi di test. Esso, infatti, permette all’utente di scrivere ed attuare un esperimento attraverso dei programmi (script) Python, mentre successivamente sarà l’interprete Python che si occuperà di invocare i metodi ORANGE previsti da tale applicazione. Tutto questo evita all’utente di dover scrivere programmi in C++, che presenta sintassi e caratteristiche notevolmente più complesse di Python, facilitandolo quindi nello sviluppo delle proprie applicazioni e rendendo, di fatto, lo stesso ORANGE più estendibile.

Altra caratteristica costruttiva di ORANGE è rappresentata dal fatto che i *data model* sono oggetti ORANGE. Grazie a questa caratteristica, un albero di classificazione può ovviamente essere visualizzato, ma ha una struttura tale che permette di iterarci sopra, ad esempio contando il numero di nodi che soddisfano alcuni criteri, escogitando metodi per cambiare la visualizzazione o ispezionandolo per trovare coppie di attributi correlati.

ORANGE è anche rappresentato da un insieme di widget grafici che fanno uso dei metodi della core library del sistema. I widget supportano la comunicazione *signal-based*, e possono essere assemblati insieme in un’applicazione, grazie ad uno strumento di programmazione visuale chiamato ORANGE Canvas [ZLDC04].

Il prossimo paragrafo si concentrerà su tale tool, che non è altro che la GUI attraverso la quale è possibile scrivere esperimenti, query di DM in ORANGE.

3.4.1 ORANGE CANVAS

Prima di entrare in dettaglio sulle principali caratteristiche dell'interfaccia, occorre soffermarci brevemente sui “mattoni” con cui essa è costruita, ovvero gli ORANGE widget [ZLDC04]. Attraverso di essi il sistema fornisce un'interfaccia grafica ai metodi di DM e machine learning di ORANGE. Nella GUI sono infatti presenti, tra gli altri, widget per

- Data entry, preprocessing.
- Data visualization.
- Classificazione, regressione, costruzione di regole di associazione, clustering.
- Modelli di valutazione

I widget, che non sono altro che *wrapper* basati su codici di analisi, comunicano e si passano token attraverso canali di comunicazione per interagire con altri widget. Generalmente supportano un numero di segnali standardizzati che possono essere creativamente combinati, per costruire l'applicazione desiderata.

L'utente può scrivere tale applicazione a mano, utilizzando script Python, o utilizzare ORANGE Canvas per costruire incrementalmente lo schema. Sia gli ORANGE widget che ORANGE Canvas sono implementati in Python, usando Qt [TQT] come libreria grafica.

L'ambiente ORANGE Canvas (Figura 3.16) è molto simile a quello visto per **Knowledge Flow** di WEKA, ed anche la filosofia di rappresentazione è la stessa. Andiamo ora a darne una breve descrizione.

ORANGE Canvas è costituito principalmente da una finestra, su cui costruire lo schema del nostro esperimento di DM o machine learning. Lo schema, infatti, non è altro che un grafo, i cui nodi sono ORANGE Widget, e gli archi sono i canali di comunicazione tra i nodi.

Nella parte alta dell'applicazione è presente la lista di tali widget divisi per categoria.

- **Data:** contiene widget per selezionare file in input, visualizzare attributi e statistiche.
- **Classify:** contiene widget per inserire classificatori nello schema, ed anche per visualizzare il risultato prodotto dal classificatore, come l'albero di classificazione estratto.
- **Associate:** attraverso questo tab è possibile inserire widget per generare regole di associazione, per visualizzarle e salvarle.
- **Visualize:** in questo tab sono presenti una serie di visualizzatori generici che, solo per citarne alcuni, comprendono tra gli altri *Scatterplot*, attraverso il quale è possibile generare grafici, e *Attribute Statistics*, per visualizzare graficamente statistiche su un insieme di dati.
- **Evaluate:** comprende una serie di strumenti per riuscire a valutare le performance predittive di un learner su un insieme di dati.

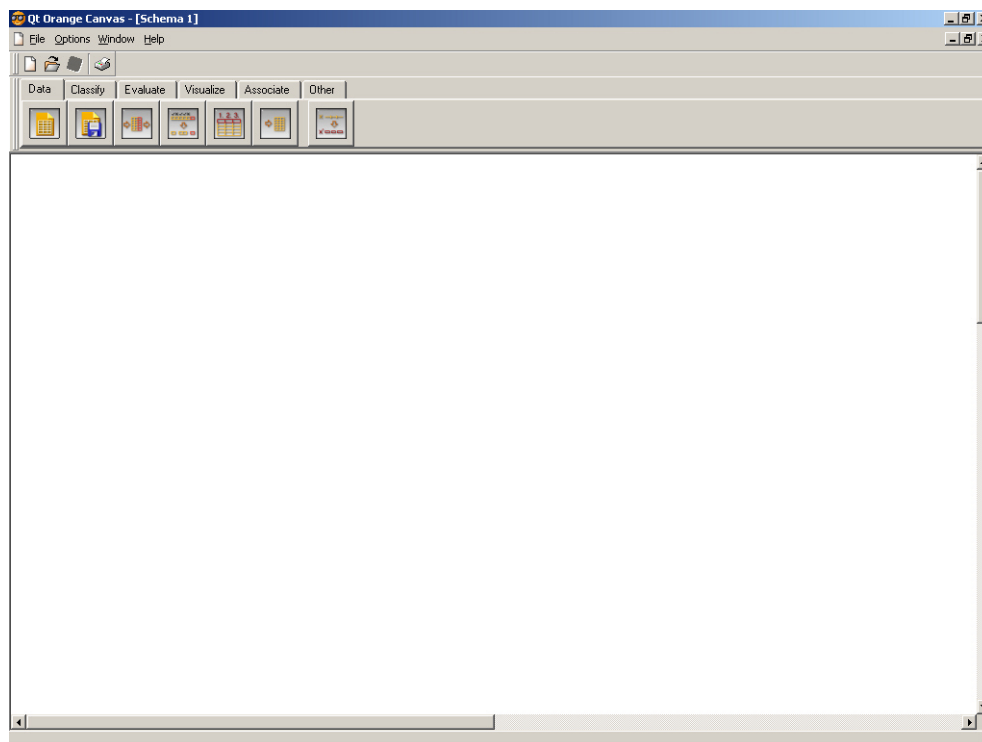


Figura 3.16 Ambiente ORANGE Canvas.

La costruzione dello schema segue la **Knowledge Flow GUI** di WEKA; si clicca sull'operatore da inserire, ed il sistema lo fa comparire nello schema. Una volta presente, lo si può trascinare ovunque e lo si può collegare ad altri widget eventualmente presenti.

Rispetto a WEKA, è inoltre possibile rinominare l'operatore, facilitando la lettura dello schema stesso, facendo attenzione a non utilizzare un nome già presente. A questo scopo, risulta utile anche la presenza di un widget chiamato *Information*, che permette di inserire utili commenti esplicativi dello schema.

La connessione dei widget risulta quindi il passo più delicato del procedimento, in quanto le connessioni non sono altro che dei canali di comunicazione, attraverso i quali i componenti si scambiano oggetti. Il tutto è complicato dal fatto che ogni ORANGE Widget molto spesso supporta diversi input e fornisce diversi output, quindi, in accordo con questa caratteristica, è in grado di ricevere e spedire token di tipi differenti. Pertanto, al fine di distinguerli, è stato necessario tipare i canali input-output previsti per ciascun operatore-widget. Per facilitare la comprensione e la costruzione dello schema, cliccando su un collegamento, o quando lo si crea, appare una finestra (Figura 3.17), in cui vengono elencati i tipi esportati dal widget di origine e quelli previsti in input per quello di destinazione.

Grazie anche a questa informazione, non solo otteniamo il flusso di dati, ma anche il tipo di oggetti, ad esempio un *Learner* (ORANGE.Learner) o un *Classification tree* (ORANGE.TreeClassifier), che passa attraverso il grafo. Degno di nota è inoltre il fatto che il sistema prevede comunque un controllo che impedisce all'utente di connettere due widget incompatibili tra loro.

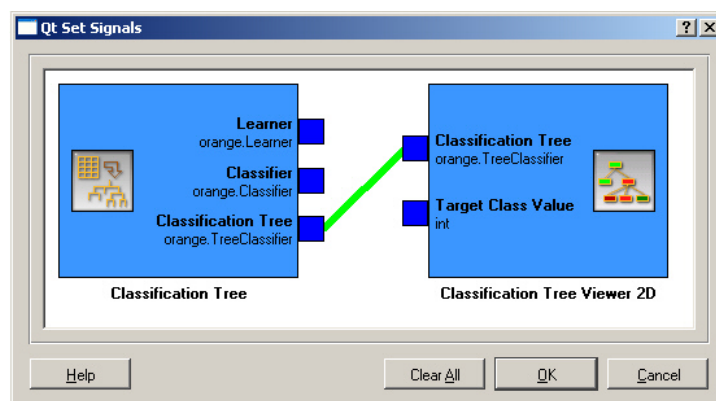


Figura 3.17 Finestra di visualizzazione del collegamento attivo tra due widget.

Per quel che riguarda l'esecuzione dello schema, il sistema lo esegue progressivamente, mentre vengono inseriti i collegamenti e sono forniti i parametri necessari ai vari widget.

Per completezza, evidenziamo anche la presenza di una finestra di output, attraverso la quale l'utente viene informato di eventuali eccezioni, che vengono sollevate dal sistema durante l'esecuzione dello schema.

Una volta costruiti gli schemi, è possibile salvarli, ed a questo punto occorre soffermarci brevemente, perché in questo caso ORANGE Canvas presenta delle caratteristiche non ancora riscontrate fino ad ora. Oltre al salvataggio classico dello schema stesso attraverso la generazione di un file XML, che racchiude i widget presenti ed i loro canali di comunicazione, abbiamo la possibilità di un altro tipo di salvataggio. Con la funzione *save as application button* (o *tabs*), il sistema genera dallo schema appena costruito una vera e propria applicazione, tramite la quale un utente interagisce, grazie alla presenza di bottoni, con le feature previste dallo schema generante. L'applicazione generata in realtà è un file Python, legato alle classi base di ORANGE e Qt, per questo la sua esecuzione è legata alla presenza nel sistema dei suddetti componenti.

Grazie a questa caratteristica, ORANGE Canvas può essere considerato uno strumento di *visual programming*, che permette anche a chi non ha conoscenze specifiche di programmazione, di costruirsi, seppur in modo intuitivo, una propria applicazione di DM, senza aver bisogno di utilizzare neppure costrutti di loop o condizionali.

Riguardo ai modelli, ORANGE Canvas mette a disposizione diversi strumenti per il loro salvataggio. Questi strumenti sono basati sul tipo di conoscenza estratta - ad esempio nel caso di regole di associazione è possibile salvarle in un file di testo -, mentre non si rileva la possibilità di salvataggio in formato di interscambio PMML, cosa invece prevista invocando i metodi ORANGE direttamente. Molto probabilmente a causa della sua gioventù, questa possibilità non è stata ancora inclusa nella GUI.

Una feature interessante fornita dalla GUI è rappresentata dalla possibilità di salvare *naive bayes* in formato XML, per attuare quella che in gergo viene chiamata

Decisions-at-Hand¹⁰ [ZDKO+01].

Tornando allo scopo principale della nostra trattazione, bisogna dire che la metafora grafica appare anche in questo caso azzeccata, in quanto attraverso di essa, non solo possiamo osservare il flusso dei dati, ma anche il tipo di dati che attraversano un determinato arco. La filosofia costruttiva permette di costruire gli schemi in maniera chiara e leggibile, vincolando la posizione dei widget all'interno del form di disegno, e ben evidenziando i canali di comunicazione, sia attivi che non attivi. Il suo utilizzo è piuttosto semplice, ed anche la presenza di un *help* sia in locale che in remoto aiuta l'utente nel prendere confidenza con il tool. In futuro è previsto l'inserimento di un *wizard*, attualmente presente solo come voce nel menù file, ma non ancora implementato.

Una piccola nota dolente riguarda l'esecuzione della query. Questa viene eseguita di pari passo con la costruzione dello schema, in maniera trasparente all'utente, che quindi non ha il totale controllo su ciò che il sistema sta effettivamente eseguendo. Una volta raggiunto un buon grado di familiarità con il sistema, si riesce comunque attraverso la disabilitazione dei canali, anche ad avere un'esecuzione parziale, come in YALE.

Un vantaggio di questo approccio risiede nel fatto che durante la composizione dello schema, il sistema, attraverso la finestra di output, informa l'utente di eventuali errori presenti. Questi errori non riguardano la sintassi, ma rappresentano errori rilevabili solo a run-time, riguardanti ad esempio caratteristiche richieste da un determinato operatore su un file di training set in input.

A livello di espandibilità, il sistema appare piuttosto rigido. La sua versione base, senza ORANGE Canvas, prevede l'utilizzo del compilatore C++ per creare le *DLL* (*Dynamic Link Library*), contenenti la propria versione di un algoritmo da utilizzare poi nel codice Python. Tale strada risulta a nostro avviso piuttosto complessa, e sicuramente più di difficile attuazione di quelle previste dai sistemi precedentemente

¹⁰ Con l'espressione Decisions-at-hand ci si riferisce ad un metodo che permette all'utente di applicare un modello codificato in XML tramite il web, o in alternativa attraverso un supporto palmare. Lo schema Decisions-at-hand attualmente supporta modelli *naive Bayes* e di *logistic regression*, ed è implementato attraverso un'applicazione di supporto di decisioni basata su un server abilitato in rete, o tramite un software attivo su un computer palmare.

analizzati basati su JAVA. L'utente deve avere notevoli nozioni di C++ e Python per ottenere un certo livello di estendibilità.

Per quanto riguarda ORANGE Canvas, purtroppo la GUI risulta statica, in quanto non è possibile, almeno nella versione attuale, estenderla inserendo dei widget personalizzati. Questo è testimoniato, ad esempio, dal fatto che è previsto un widget per l'utilizzo dell'algoritmo classificatore C4.5, ma le *DLL* per eseguirlo sono presenti solo in un plug-in a parte. La presenza del widget senza il codice mostra che gli sviluppatori non hanno ancora previsto un metodo per inserirlo direttamente attraverso l'installazione del plug-in.

Concludendo, ORANGE Canvas sembra concepito più come uno strumento per riuscire a programmare applicazioni DM ORANGE in maniera visuale, piuttosto che come una semplice metafora grafica per costruire e visualizzare risultati di query KDD. ORANGE Canvas utilizza quindi il concetto di flusso di dati sia come astrazione, metafora del processo KDD, che come linguaggio nella costruzione di applicazioni KDD.

3.4.2 ESEMPIO DI UTILIZZO DI ORANGE CANVAS

Cominciamo ad illustrare il nostro esempio dalla selezione del file in input (Figura 3.18), che costituirà il nostro training set per la costruzione dell'albero di classificazione. In realtà, tale scelta può essere anche posticipata, mentre l'importante è l'inserimento del nodo addetto a tal fine. In questo caso, il sistema non prevede metodi automatizzati per la gestione di valori mancanti.

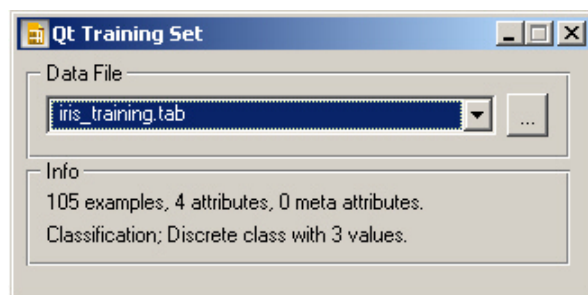


Figura 3.18 Dialog Box per la scelta del file contenente il training set.

Abbiamo quindi i nodi per la definizione di training e test set, selezionando l'operatore *File* dal gruppo *Data*. I file caricati sono gli stessi visti per WEKA, senza i valori mancanti. Si ricorda che ORANGE offre la possibilità di rinominare i nodi.

Inseriamo e colleghiamo ora i nodi per visualizzare i data set e costruire l'albero. Per fare ciò, ci serviamo degli operatori *Data Table* dal tab *Data* e *Classification Tree*¹¹ da *Classify*. Di quest'ultimo lasciamo i parametri di default, impostabili da un menù che compare facendo doppio click sul nodo, come mostrato in Figura 3.19.

Da notare, sempre dalla Figura 3.19, che lasciando il puntatore del mouse su un operatore, compare un *tooltip* che informa l'utente dei possibili segnali in input e output per quel determinato operatore. Tale informazione ci aiuta nel collegare i vari nodi presenti.

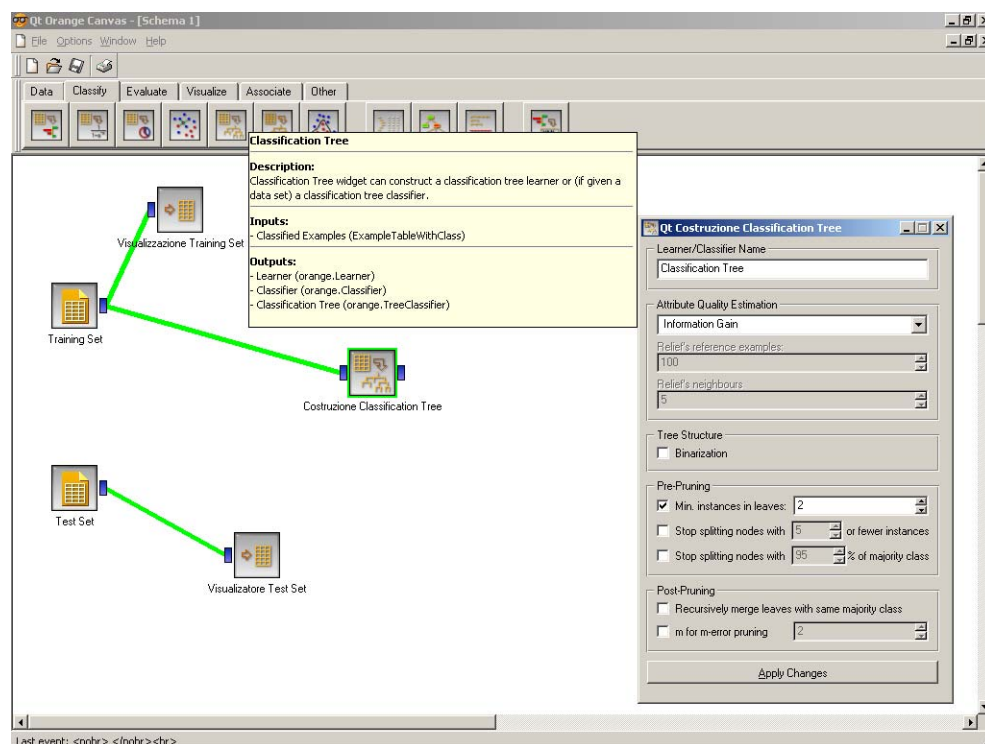


Figura 3.19 Inserimento del nodo per la costruzione dell'albero di classificazione.

Per creare un collegamento, ricordiamo che è sufficiente cliccare su uno dei rettangoli blu presenti ai lati di un operatore. Quello a sinistra del box rappresenta

¹¹ Anche in questo caso l'inserimento di operatori per effettuare la selezione della classe su cui costruire il modello non è necessaria, poiché la definizione dell'attributo da predire è fornita insieme al file data set.

l'input, mentre quello a destra l'output. Una connessione tra due nodi si ottiene cliccando col mouse sul rettangolo in output di un operatore e, tenendo il tasto premuto, trascinando il puntatore del mouse sul rettangolo in input del nodo di destinazione o viceversa.

L'ultimo passo di costruzione riguarda l'inserimento ed il collegamento dei nodi, per consentire la visualizzazione della conoscenza estratta, la sua applicazione ed una valutazione della bontà del modello estratto. A tal fine, inseriamo l'elemento per visualizzare l'albero in 2D (*Classification Viewer 2D* dal tab *Classify*), e da *Evaluate* gli operatori *Test Learner* e *Classification*, rispettivamente per valutare il modello estratto e per ottenere una classificazione sul test set. La Figura 3.20 mostra lo schema finale.

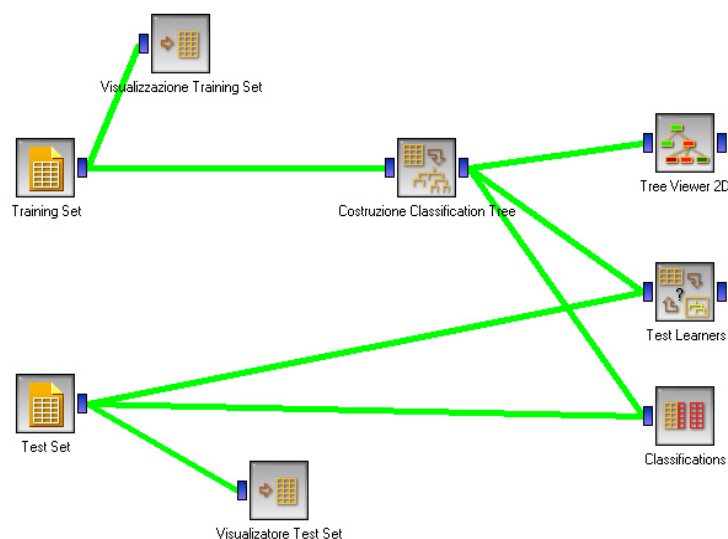


Figura 3.20 Schema finale per la costruzione di un albero di classificazione.

In ORANGE l'esecuzione dell'esperimento avviene progressivamente, mentre si collegano i vari nodi e si forniscono loro i parametri necessari. Ad esempio, nel caso di data set di grandi dimensioni, conviene fornire al sistema il file contenente i dati solo alla fine della costruzione dello schema, evitando così attese ogni volta che inseriamo un nodo che compie operazioni su tale data set.

Per visualizzare la conoscenza estratta, basta cliccare sul nodo che presenta la visualizzazione dell'informazione che ci interessa. La Figura 3.21 mostra la finestra fornita dal nodo *Tree Viewer 2D*.

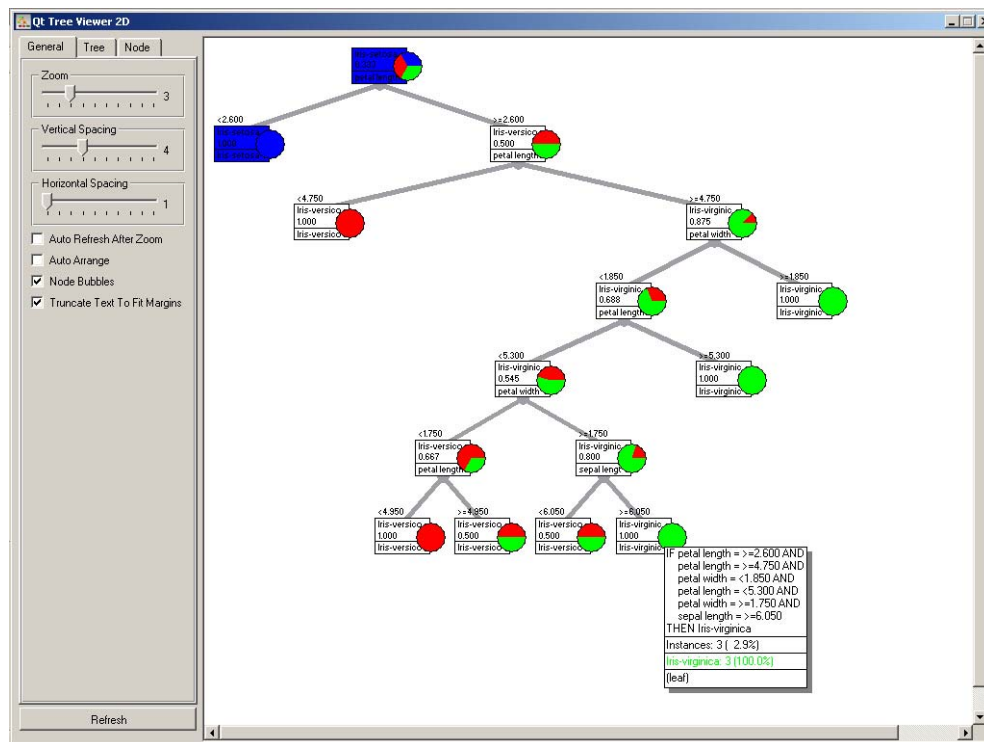


Figura 3.21 Visualizzazione dell'albero estratto.

La caratteristica sicuramente più interessante di ORANGE è quella di permettere la costruzione di una vera e propria applicazione grafica a partire dallo schema appena redatto. Per fare ciò, è sufficiente accedere al menù *File*, cliccare su *save application as (Tabs)*, e scegliere gli elementi che vogliamo rendere accessibili nell'applicazione¹².

¹² In questo caso abbiamo scelto di includere tutti i widget presenti nello schema.

	iris	Classification Tree
1	Iris-setosa	1.000000 : 0.000000 : 0.000000 -> Iris-setosa
2	Iris-setosa	1.000000 : 0.000000 : 0.000000 -> Iris-setosa
3	Iris-setosa	1.000000 : 0.000000 : 0.000000 -> Iris-setosa
4	Iris-setosa	1.000000 : 0.000000 : 0.000000 -> Iris-setosa
5	Iris-setosa	1.000000 : 0.000000 : 0.000000 -> Iris-setosa
6	Iris-setosa	1.000000 : 0.000000 : 0.000000 -> Iris-setosa
7	Iris-setosa	1.000000 : 0.000000 : 0.000000 -> Iris-setosa
8	Iris-setosa	1.000000 : 0.000000 : 0.000000 -> Iris-setosa
9	Iris-setosa	1.000000 : 0.000000 : 0.000000 -> Iris-setosa
10	Iris-setosa	1.000000 : 0.000000 : 0.000000 -> Iris-setosa
11	Iris-setosa	1.000000 : 0.000000 : 0.000000 -> Iris-setosa
12	Iris-setosa	1.000000 : 0.000000 : 0.000000 -> Iris-setosa
13	Iris-setosa	1.000000 : 0.000000 : 0.000000 -> Iris-setosa
14	Iris-setosa	1.000000 : 0.000000 : 0.000000 -> Iris-setosa
15	Iris-setosa	1.000000 : 0.000000 : 0.000000 -> Iris-setosa
16	Iris-versicolor	0.000000 : 1.000000 : 0.000000 -> Iris-versicolor
17	Iris-versicolor	0.000000 : 1.000000 : 0.000000 -> Iris-versicolor
18	Iris-versicolor	0.000000 : 1.000000 : 0.000000 -> Iris-versicolor
19	Iris-versicolor	0.000000 : 1.000000 : 0.000000 -> Iris-versicolor
20	Iris-versicolor	0.000000 : 1.000000 : 0.000000 -> Iris-versicolor
21	Iris-versicolor	0.000000 : 1.000000 : 0.000000 -> Iris-versicolor
22	Iris-versicolor	0.000000 : 1.000000 : 0.000000 -> Iris-versicolor
23	Iris-versicolor	0.000000 : 1.000000 : 0.000000 -> Iris-versicolor
24	Iris-versicolor	0.000000 : 1.000000 : 0.000000 -> Iris-versicolor
25	Iris-versicolor	0.000000 : 1.000000 : 0.000000 -> Iris-versicolor
26	Iris-versicolor	0.000000 : 1.000000 : 0.000000 -> Iris-versicolor
27	Iris-versicolor	0.000000 : 1.000000 : 0.000000 -> Iris-versicolor
28	Iris-versicolor	0.000000 : 1.000000 : 0.000000 -> Iris-versicolor
29	Iris-versicolor	0.000000 : 1.000000 : 0.000000 -> Iris-versicolor
30	Iris-versicolor	0.000000 : 1.000000 : 0.000000 -> Iris-versicolor
31	Iris-virginica	0.000000 : 0.000000 : 1.000000 -> Iris-virginica
32	Iris-virginica	0.000000 : 0.000000 : 1.000000 -> Iris-virginica
33	Iris-virginica	0.000000 : 0.000000 : 1.000000 -> Iris-virginica

Figura 3.22 Visualizzazione dell'applicazione prodotta dallo schema di Figura 3.20.

La Figura 3.22 mostra il risultato prodotto dalla modalità *Tabs*, in cui tutti gli elementi presenti nello schema sono accessibili selezionando i tab nella parte alta. Come possiamo vedere, per ogni nodo è stato generato un apposito tab, attraverso il quale è possibile accedere e modificare i vari parametri previsti da tale nodo. Ad esempio, accedendo al tab *Test Set*, è possibile cambiare il file utilizzato come test set. L'altra modalità di salvataggio, *save application as (Buttons)*, è del tutto simile e crea un'applicazione, in cui sono presenti un gruppo di bottoni, grazie ai quali è possibile accedere agli stessi menù appena visti.

3.5 GHOSTMINER

Proseguiamo la nostra rassegna, andando ad esaminare un software chiamato GhostMiner 3.0 [FGH], sviluppato da FQS Poland con il supporto di Fujitsu. A differenza dei sistemi visti finora, questo è un tool per uso commerciale, quindi non è gratis, e non è disponibile il suo codice sorgente.

In realtà, anche questo sistema è nato per scopi di ricerca, e non sarebbe stato possibile utilizzarlo al di fuori di tale ambito, se non con l'introduzione di una GUI.

La creazione della GUI, disponibile solo per sistemi *Microsoft Windows*, ha richiesto un notevole sforzo che, a detta degli stessi autori, sarebbe stato irrealizzabile senza un supporto commerciale. Questo fatto comunque non invalida le nostre analisi, anzi a nostro avviso fornisce dei metri di paragone aggiuntivi, per valutare il grado di evoluzione dei sistemi open source, tipicamente sviluppati in ambito accademico, rispetto a quelli costruiti a fini commerciali, quindi presumibilmente con budget più ricchi.

Un'altra caratteristica che contraddistingue GhostMiner rispetto ai tool esaminati fino ad ora, è rappresentata dal fatto che non si basa su un sistema preesistente a riga di comando come WEKA, ma è utilizzabile esclusivamente attraverso la sua interfaccia grafica.

Questo pacchetto per il DM può essere utilizzato da esperti in molti progetti per l'analisi dei dati, ma i risultati finali dovrebbero essere il più semplice possibile da usare. L'utente finale dovrebbe accedere soltanto alle opzioni *made-to-order* utili per il particolare tipo di dati con cui ha a che fare. Questo richiede quindi un approccio a due fasi, che portano il sistema ad essere costituito da due programmi:

- **GHOSTMINER Developer:** è il programma principale, ed è progettato per fornire un'analisi esplorativa dei dati, e per la costruzione ed il testing di modelli predittivi. Vedremo più avanti in dettaglio alcune sue caratteristiche.
- **GHOSTMINER Analyzer:** è progettato per permettere all'utente la valutazione di nuovi dati, usando modelli precedentemente creati con GhostMiner Developer. Questo programma non verrà qui esaminato, poiché non è rilevante per la costruzione di una query KDD.

La struttura utilizzata ad alto livello da entrambi i programmi è chiamata *progetto*. Un *progetto* GhostMiner specifica i dati che vengono analizzati, i modelli ed il loro uso per riconoscere ed interpretare nuovi dati. Senza entrare troppo in dettaglio,

diamo ora una descrizione generale delle classi dei metodi disponibili in GhostMiner 3.0:

- **Initial data exploration.** Include l'abilità di visualizzare i dati in modi diversi, standardizzarli e normalizzarli, controllare le statistiche fondamentali per ogni attributo, scoprire vettori (tuple) insoliti, osservando le rappresentazioni 2D e 3D di caratteristiche sotto forma di punti, o utilizzare *scatterplot* bidimensionali di punti.
- **Visualization.** La visualizzazione di caratteristiche di singoli dati o le proiezioni in due dimensioni non ci forniscono informazioni rilevanti riguardo ai cluster nei dati o alle relazioni tra i singoli elementi. Un modo semplice per visualizzare le relazioni tra i dati è individuare due dimensioni secondo le quali i dati cambiano maggiormente. A tal fine, abbiamo a disposizione due metodi di visualizzazione chiamati rispettivamente PCA e MDS. Per una completa descrizione di tali metodi, fare riferimento a [FGM].
- **Feature selection.** I modelli di *feature selection* sono molto utili, specialmente quando i vettori di dati sono descritti tramite una vasta quantità di caratteristiche, facilitano la selezione di un sottoinsieme di dati, per cui il processo di classificazione è più semplice. A tal fine, GhostMiner presenta diversi modelli per il *feature selection*, che possono essere utilizzati come una singola operazione, o associati a classificatori.
- **Classification/Cluster.** In questa classe sono presenti diversi algoritmi di classificazione e clusterizzazione, come ad esempio:
 - **SSV** (decision tree)
 - **FSM** (neurofuzzy system)
 - **kNN** (weighted nearest neighbor method)
 - **IncNet** (constructive neural network algorithm)

Inoltre, per aumentare l'accuratezza dei modelli predittivi, è possibile combinare i risultati di diversi classificatori. GhostMiner permette la creazione di gruppi di classificatori dalla complessità arbitraria. Tali gruppi, infatti, possono contenere classificatori dello stesso tipo, o di vari tipi, contenenti a loro volta sottogruppi.

Grazie a queste caratteristiche, GhostMiner trova applicazione in vari domini, che vanno dalla finanza, alle telecomunicazioni, al commercio elettronico, fino all'astronomia, alla biologia, alla medicina ed alla bioinformatica.

Entriamo adesso più in dettaglio, andando ad esaminare quali caratteristiche ci mette a disposizione GhostMiner Developer.

3.5.1 GHOSTMINER DEVELOPER

GhostMiner Developer è uno strumento per costruire progetti di DM. Esso infatti permette di:

- manipolare i dati, rendendoli compatibili con i metodi disponibili
- controllare il processo di creazione dei modelli
- valutare la bontà dei modelli appena scoperti
- fornire metodi per la completa analisi dei dati e modelli costruiti su tali dati.

GhostMiner è stato costruito come un sistema orientato agli oggetti, e la manipolazione dei progetti è parte di questa costruzione. Tutte le azioni in GhostMiner sono “context-dependent”. Per esempio, se un modello è l'elemento attivo in un *progetto*, le operazioni per configurare, allenare e resettare il modello, che si trovano nel menù principale o nella barra degli strumenti, sono disponibili e si riferiranno al modello attivo.

Il *progetto* è quindi organizzato e rappresentato come un albero di oggetti di diversi tipi. La metafora è simile a quella riscontrata in YALE, ed in questo caso un

operatore non è altro che un oggetto visto alla luce di ciò che produce, e come tale ha degli attributi visualizzabili.

L'interfaccia grafica (Figura 3.23) risulta piuttosto semplice ed immediata, ed è essenzialmente divisa in due parti:

- **Project tree window.** Posta a sinistra del “*main context*”, rappresenta l'albero del *progetto*, ed è il centro di utilizzo principale di tutto GhostMiner Developer. Essa, infatti, mostra la struttura dei modelli applicati al data set, e permette all'utente di aggiungere, rimuovere, configurare ed allenare modelli in maniera semplice. In questo contesto, un modello è un algoritmo (metodo) con specifici parametri, allenato su un insieme di dati.
- La sezione a destra è lasciata vuota e permette, selezionando un modello presente nel **Project tree**, di accedere ad una notevole quantità di informazioni. Ad esempio, una volta caricato un data set, compare una finestra chiamata **Dataset Information**, che fornisce informazioni statistiche sui dati appena caricati, come il numero di istanze, il numero di attributi etc. Le differenti feature disponibili sono mostrate selezionando i vari tab presenti nella parte alta della finestra. Da evidenziare è la presenza dei tab **N-Dots** e **2D**, che forniscono una comparazione visuale della distribuzione dei dati tra le varie classi e per ogni attributo presente nei dati.

Questa caratteristica risulta importante, in quanto queste informazioni possono aiutare l'utente nella costruzione e validazione di particolari modelli. Basti pensare ad un training set in cui la distribuzione delle classi sugli attributi è netta. Non c'è da meravigliarsi, se un modello allenato su tali dati darà un'accuratezza del 100%. Ciò però può spingere l'utente ad eseguire altri test, in quanto tale accuratezza, ad esempio di un albero di classificazione su tale training set, può essere dovuta alla fortunata distribuzione dei dati su tali classi, e non all'effettiva bontà del modello predittivo scoperto.

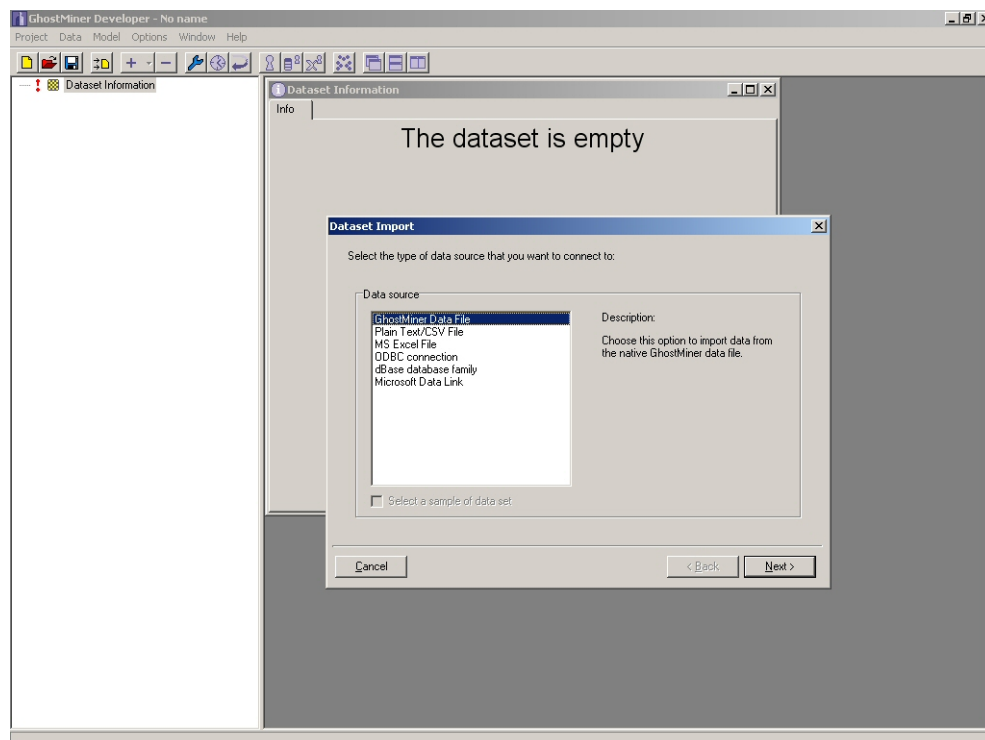


Figura 3.23 GhostMiner Developer.

La costruzione di un *progetto* avviene in maniera incrementale. Un *progetto* può avere solo un training set di dati che viene selezionato all'inizio della sua costruzione. Successivamente si inseriscono e si configurano, attraverso apposite finestre, i vari operatori, in questo caso chiamati modelli. In altre parole, un *progetto* consiste in un data set ed in un insieme di modelli che utilizzano questi dati. Inutile dire che il sistema permette di importare dati da vari formati, proprietari e non, e da basi di dati preesistenti.

Per quel che riguarda l'esecuzione del *progetto*, il sistema permette di eseguirlo di pari passo all'inserimento dei modelli, o quando si ritiene più opportuno, rendendo inoltre possibile l'esecuzione di singoli sottoalberi, purché l'informazione necessaria sia disponibile al nodo radice del sottoalbero. I risultati dei modelli addestrati sono mostrati in una finestra chiamata **Train Results**, che integra i tab già presenti in **Dataset Information**, con altri specifici per la rappresentazione della conoscenza estratta. Bisogna inoltre evidenziare che ogni modello allenato ha una propria **Train Results window**, e che ad ogni istante è attivo solo un item del *progetto* a cui è riferita tale finestra. Una caratteristica interessante è la possibilità, nel caso sia

presente nel *progetto* un *Cross Validation Training (CVT)*, di andare ad analizzare gli oggetti creati dal CVT per generare il modello finale, visualizzando per ogni oggetto varie statistiche, come l'accuratezza sia sul training set che sul test set.

L'esecuzione effettiva dei modelli può avvenire in maniera parallela ed indipendentemente dagli altri, poiché ad ogni modello è associato un thread, caratteristica che porta notevoli vantaggi solo nel caso di architetture multiprocessore.

Per quanto riguarda il salvataggio della conoscenza estratta, si rileva un buon numero di opzioni (oltre il formato proprietario), che vanno da vari formati immagine, utilizzabili per salvare grafici o alberi di decisione, a formati di markup standardizzati come HTML, XML o semplicemente testo. Non si rileva la possibilità di salvare in formato PMML.

Andando ad analizzare la metafora scelta, possiamo affermare che essa sembra buona, ma presenta qualche limitazione. Su di essa infatti si basa tutto il sistema, e questo porta ad inserirvi sia gli operatori che gli oggetti generati da tali operatori, in quanto l'unico modo di selezionarli è attraverso il **Project Tree**. Questo a nostro avviso, nel caso di progetti complessi, conduce al sovraffollamento di tale finestra, reso ancora più importante dagli attributi visualizzabili per ogni oggetto creato. Al fine di aiutare l'utente nella "lettura" di un *progetto*, sono presenti una serie di caratteristiche, come la possibilità di non visualizzare completamente tutti i sottoalberi, l'icona per l'item attivo ed informazioni sui modelli addestrati e non.

Nel caso degli algoritmi implementati, bisogna evidenziare il fatto che il sistema non fornisce una vasta serie di algoritmi, soprattutto se paragonato a WEKA e YALE, mentre riesce comunque a coprire un'ampia serie di problematiche che comprendono, oltre al machine learning, anche i sistemi fuzzy e le reti neurali.

La costruzione della query (in questo caso si parla di *progetto*), risulta abbastanza intuitiva, anche se non vi è la presenza di *wizard*. L'utente è comunque vincolato dal sistema a seguire una serie di passi per poterla costruire, primo fra tutti la scelta del data set, senza il quale il sistema non permette di inserire alcun oggetto nel *progetto*. Nel complesso, la GUI risulta piuttosto curata, e sicuramente è da elogiare la presenza di bottoni, sotto le voci del menù principale, che semplificano la stesura del

progetto stesso. Infatti, attraverso di essi, l'utente accede in maniera semplice e diretta alla maggior parte delle funzionalità del sistema.

A nostro avviso, bisogna comunque sottolineare che il sistema sembra essere stato concepito in maniera tale da dare più enfasi alla visualizzazione dei dati e dei risultati, piuttosto che alla costruzione e rappresentazione di un esperimento di DM. Attraverso il **Project tree** non si riesce infatti a capire immediatamente con chiarezza il flusso dei dati attraverso i vari oggetti presenti. Nella finestra, infatti, sono presenti sia modelli, algoritmi inseriti esplicitamente dall'utente, che sottomodelli generati automaticamente dal sistema, al fine di eseguire il modello principale. Questo, almeno fino a che non si raggiunge una certa familiarità con il sistema, può confondere l'utente su qual è l'operatore importante da considerare.

Bisogna comunque affermare che la documentazione fornita, insieme al software, risulta piuttosto completa ed aiuta non poco l'utente ad acquisire in breve tempo confidenza sia con l'utilizzo dell'interfaccia grafica, che con il sistema di rappresentazione usato per il *progetto*, e quindi con la navigazione delle caratteristiche della finestra **Project tree**, e le potenzialità di visualizzazione offerte.

La nota dolente è data dal fatto che il sistema non è espandibile, caratteristica comunque tipica dei sistemi sviluppati a fini commerciali. Questa mancanza però lo pone sicuramente un gradino sotto i sistemi incontrati finora, in quanto reputiamo la possibilità di espansione una delle caratteristiche fondamentali di qualsiasi DM tool.

Concludendo, tralasciando il fatto che il sistema non è espandibile ed a pagamento, possiamo affermare che è molto curato dal punto di vista della visualizzazione e rappresentazione dei risultati, mentre appare meno definito nella parte che concerne la rappresentazione e la costruzione della query.

3.5.2 ESEMPIO DI UTILIZZO DI GHOSTMINER DEVELOPER

Come per gli esempi precedentemente esaminati, il primo passo riguarda l'importazione del data set su cui estrarre la conoscenza. Questa prima fase è supportata da un *wizard* (Figura 3.24), che compare al momento della creazione di un nuovo *progetto*. Come possiamo osservare in Figura 3.24, è possibile importare i dati in vari formati, o attraverso una connessione a database. Nel nostro caso,

scegliamo di importare il file *Iris* già visto negli esempi di WEKA e YALE, ma in questo caso privo di valori mancanti, attraverso la voce *Plain Text/CSV File*.

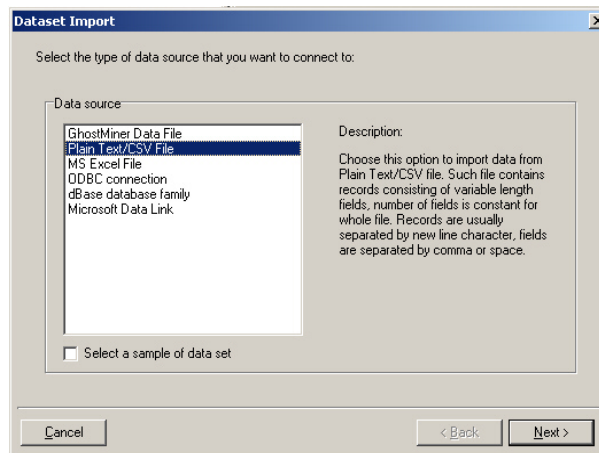


Figura 3.24 Import *wizard* di GhostMiner Developer.

Una volta importato il file, possiamo andare ad esplorare una serie di statistiche presenti nel data set, oppure visualizzare il data set stesso, sia in formato testuale che attraverso grafici.

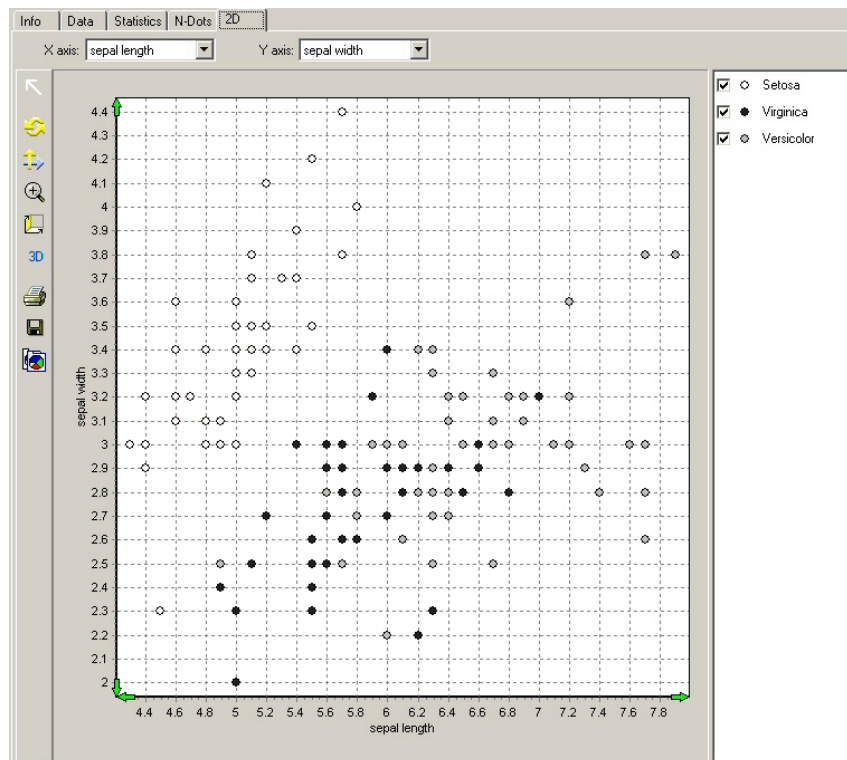


Figura 3.25 Grafico rappresentante la distribuzione dei dati in input.

Osservando la Figura 3.25, possiamo ad esempio notare, considerando la *sepal*¹³ *length* e la *sepal width*, che la classe *setosa* è nettamente separata dalle altre.

In questo caso, per variare leggermente rispetto agli esempi illustrati precedentemente, decidiamo di costruire l'albero solo su un sottogruppo di attributi. A tal fine, selezioniamo dal menù *Model* la voce *Data Transformation*, ed inseriamo l'operatore *Manual Feature Selection*. Successivamente, premendo sul tasto *Configure*, scegliamo gli attributi che riteniamo rilevanti (in questo caso *petal length* e *petal width*).

Il passo successivo è quello di inserire il classificatore che genererà l'albero. Selezioniamo quindi il nodo padre di tale operatore, accediamo al menù *Model*, scegliamo la voce *Add Classifier*, ed alla finestra successiva scegliamo *SSV Tree*¹⁴. Attraverso il bottone *Configure*, possiamo scegliere le varie impostazioni previste per tale operatore, ma in questo caso optiamo per lasciare quelle di default (Figura 3.26).

GhostMiner Developer prevede che per la costruzione di un albero sia necessaria una cross-validation, che per default viene eseguita 10 volte. Una volta inserito l'operatore, il sistema chiede di eseguirlo.

¹³ Sepalo: parte di un fiore, solitamente verde, che circonda e protegge il fiore nel germoglio. I sepali derivano da foglie modificate, e sono di solito indicate con il nome collettivo di *calice*.

¹⁴ Come nel caso di YALE e ORANGE, la classe su cui costruire il modello è definita all'interno dei dati.

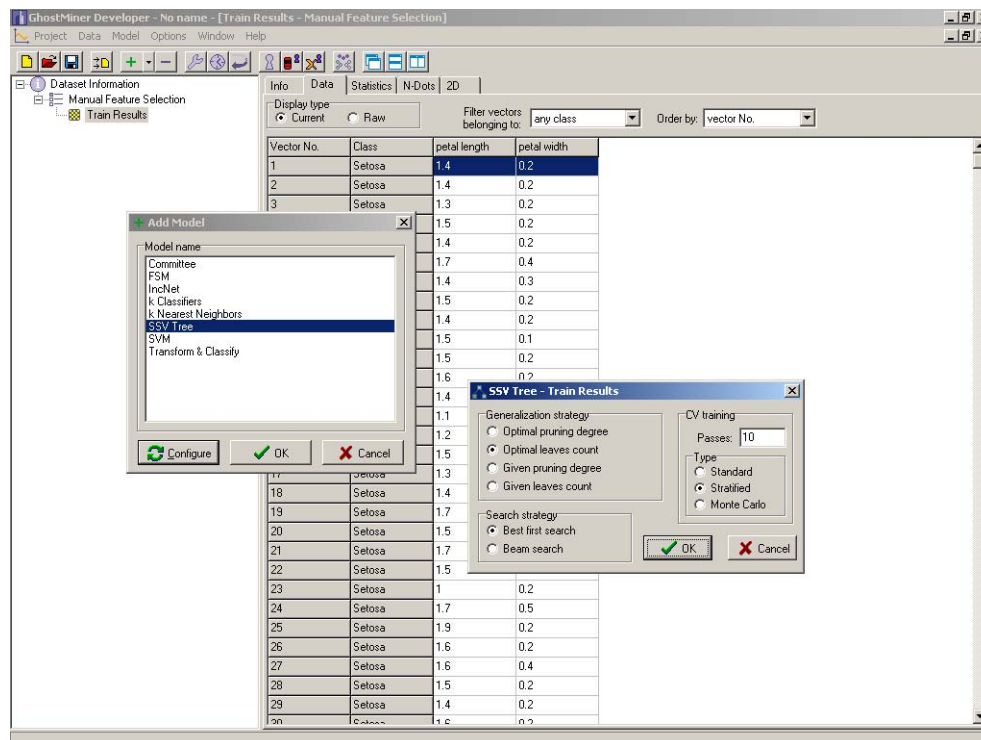


Figura 3.26 Inserimento e configurazione dell'operatore *SSV Tree*.

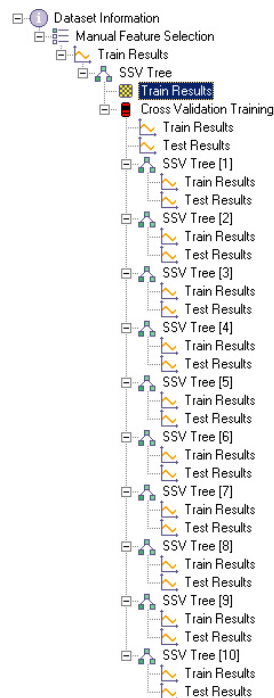


Figura 3.27 Treeview del sistema per la generazione dell'albero di classificazione.

Come si può notare dalla *treeview* di Figura 3.27, dopo l'esecuzione del *progetto*, il sistema ha costruito un sottoalbero per ogni passo di validazione, e per ogni sottoalbero sono disponibili varie statistiche, tra cui la parte del data set originario utilizzata come training set, quella utilizzata come test set, l'accuratezza ad ogni passo, l'albero costruito all'i-esimo passo etc.

Il sottonodo *Train Results* di *SSV Tree*, oltre alle varie statistiche appena menzionate, permette la visualizzazione dell'albero finale estratto, basato su 10 passi di validazione. In questo caso presentiamo quella con gli *Info Boxes*.

Come possiamo notare dalla Figura 3.28, ad ogni nodo dell'albero è associata una condizione, ed una serie di statistiche che informano sull'accuratezza del modello.

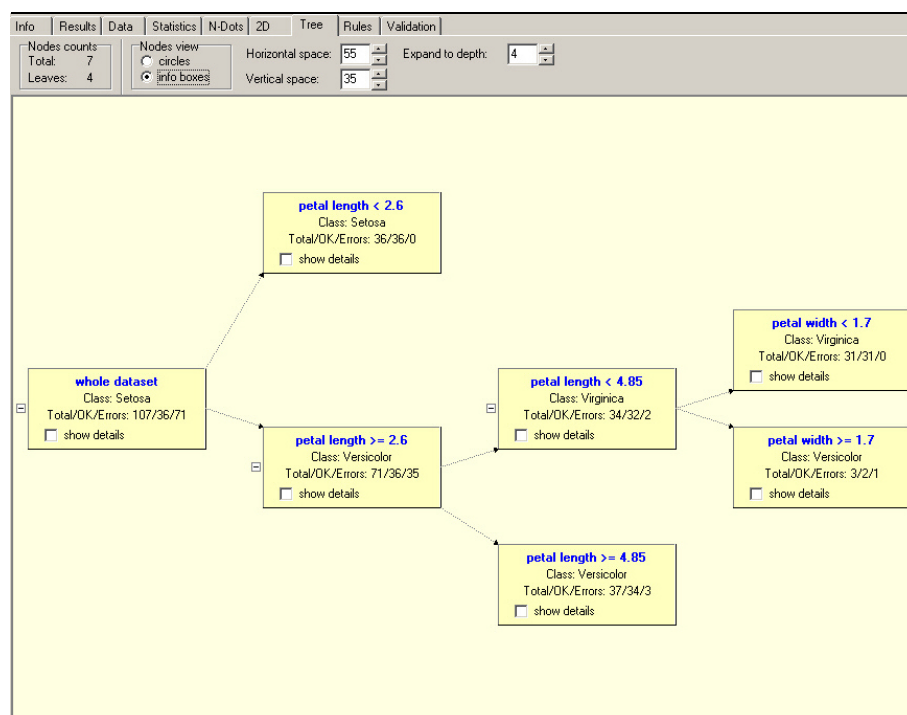


Figura 3.28 Albero di classificazione estratto.

Per completezza, dobbiamo sottolineare il fatto che GhostMiner Developer è concepito per essere utilizzato in parallelo a GhostMiner Analyzer, infatti il modello appena costruito può essere applicato con l'utilizzo di GhostMiner Analyzer. Possiamo applicare adesso il passo di testing, selezionando il nodo *SSV Tree*, andando nel menù *Model*, e scegliendo la voce *Test*. Una volta inserito tale

operatore, compare il *wizard* come quello in Figura 3.24, per importare il test set. Anche in questo caso, l'insieme è lo stesso visto negli esempi precedenti.

La Figura 3.29 mostra la matrice di confusione derivata dal passo di testing e l'albero finale del *progetto*.

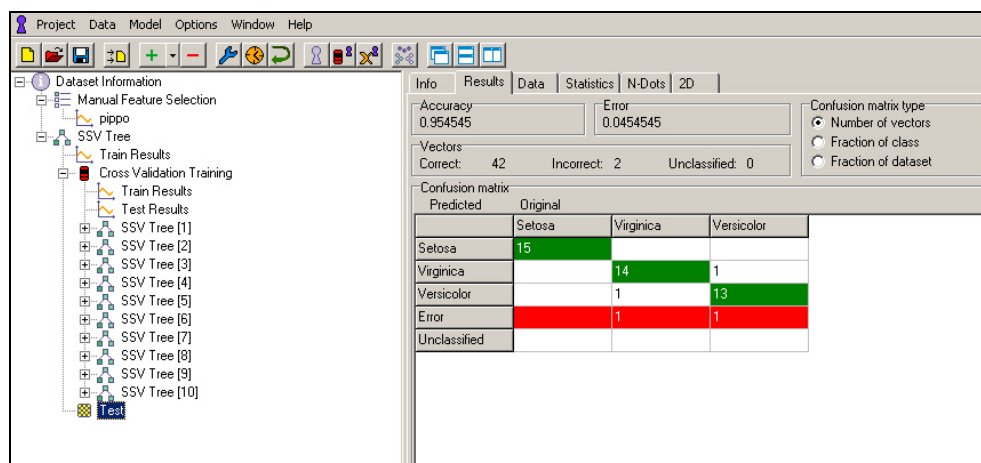


Figura 3.29 Albero finale del *progetto* e matrice di confusione sul test set.

3.6 TANAGRA

Continuiamo, introducendo un nuovo sistema scritto in Delphi 6 [DELPHI], chiamato TANAGRA [TAN]. TANAGRA è un software free ed open source per il DM, sviluppato in Francia, più precisamente all'Università di Lione 2, da Ricco Rakotomalala con scopi accademici e di ricerca. A tal fine, propone diversi metodi per l'esplorazione e l'analisi dei dati, machine learning e statistical learning. Per il suo utilizzo fare riferimento alla licenza presente nel sito [TAN].

Lo scopo principale del progetto TANAGRA, qui esaminato nella versione 1.1.4, è quello di fornire a ricercatori e studenti un software di DM facile da usare e conforme alle attuali norme per lo sviluppo del software in questo dominio, specialmente riguardo alla costruzione e l'utilizzo dell'interfaccia grafica. Infatti, TANAGRA si comporta come una piattaforma sperimentale, che permette agli utenti di accedere direttamente alla parte essenziale del loro lavoro, dispensandoli dalla parte più noiosa che riguarda la programmazione di questo tipo di strumenti, ovvero la gestione dei dati.

Il secondo obiettivo è quello di fornire ai ricercatori un'architettura che permetta loro di estendere il sistema con propri algoritmi di DM, al fine di comparare le varie performance ottenute, mentre il terzo ed ultimo scopo di TANAGRA va nella direzione degli sviluppatori, e consiste nel diffondere una possibile metodologia per la costruzione di questo tipo di software.

In questo caso l'utente, avvantaggiandosi della fruibilità del codice sorgente, può osservare come tale software è stato implementato, quali sono i problemi da evitare, i maggiori passi del progetto, e quali strumenti e librerie possono essere utilizzate per i singoli scopi. In questo senso, possiamo essenzialmente affermare che TANAGRA funziona da strumento pedagogico per l'apprendimento delle tecniche di programmazione, rivolte alla produzione di software per il DM.

TANAGRA viene definito dal suo stesso autore come un software di nuova generazione che associa all'interno dello stesso framework la maggior parte delle categorie dei metodi per il DM, in quanto oltre ai metodi per il supervised learning, sono stati integrati metodi di clustering, di feature selection etc., in modo che possano cooperare. Una caratteristica, questa, che può risultare particolarmente interessante in un processo di feature construction.

Anche in questa occasione, il sistema è totalmente integrato nella sua GUI, e quindi il suo utilizzo è possibile solo attraverso di essa.

Al momento TANAGRA non include quello che costituisce il punto di forza dei software commerciali in questo ambito, ovvero una vasta gamma di data source, accesso diretto ai database e al data warehouse, data cleaning ed utilizzo interattivo.

L'intera interfaccia utente di TANAGRA è basata su un paradigma a diagramma di flusso. All'interno di tale paradigma, l'utente costruisce un grafico, specificando il data source e le operazioni su di essi. TANAGRA in realtà semplifica questo paradigma, riducendo il grafico ad albero, in modo da poter avere un unico padre per ogni nodo, e quindi un'unica fonte di dati per ogni operazione.

Andiamo adesso ad esaminare più in dettaglio le caratteristiche salienti di TANAGRA.

3.6.1 TANAGRA GUI

L'interfaccia grafica di TANAGRA (Figura 3.30) è essenzialmente costituita da tre parti; a sinistra abbiamo la descrizione del già citato diagramma di flusso (*stream diagram*), ovvero la *treeview* del nostro progetto. La metafora utilizzata è quindi pressoché identica a quella vista per GhostMiner Developer. Nella parte bassa, abbiamo l'insieme di nodi, in realtà operatori, componenti inseribili nell'albero, divisi ciascuno nella propria categoria. La parte rimanente è riservata alla visualizzazione dei risultati, che sono in formato HTML.

Abbiamo appena detto che gli operatori di TANAGRA sono divisi in categorie. Andiamo adesso a dare una breve descrizione di tali categorie:

- **Data Visualization.** In questa sezione sono presenti diversi operatori sia per visualizzare i dati, in maniera testuale o attraverso grafici, che per esportarli in formato testuale.
- **Descriptive Stats.** In questo gruppo troviamo operatori grazie ai quali è possibile accedere ad una serie di statistiche insite nei dati in ingresso.
- **Instance Selection.** Attraverso i componenti qui presenti è possibile ridurre l'insieme di esempi attivi, prima di effettuare un'analisi. In TANAGRA, infatti, gli esempi, ovvero i termini dei dati in ingresso, sono suddivisi in attivi e non attivi. Nella costruzione di un particolare modello parteciperanno solo i termini attivi. Se non diversamente specificato, tutti i termini presenti in un data set sono considerati attivi.
- **Feature construction/Feature selection.** Attraverso gli operatori di questi due gruppi è possibile rispettivamente produrre nuovi attributi, sia automaticamente che non, da attributi disponibili, o modificare sempre automaticamente o manualmente lo stato di alcuni attributi. Quest'ultima caratteristica viene frequentemente usata per ridurre l'insieme di attributi in input.

- **Regression.** Qui è presente un solo componente, grazie al quale è possibile ottenere un'analisi di regressione, predire valori di un attributo continuo (endogeno) da altri attributi (esogeni).
- **Factorial analysis.** Attraverso queste componenti è possibile produrre una descrizione degli esempi forniti in input in una ridotta dimensionalità.
- **Clustering.** In questa sezione sono presenti componenti per la costruzione di cluster di dati. Tali componenti sono conosciuti anche come metodi unsupervised.
- **Spv Learning.** I componenti presenti in questa sezione implementano una serie di algoritmi supervised. Qui sono presenti i tipici algoritmi per la costruzione di alberi di classificazione, naive-bayes, etc.
- **Meta-Spv Learning.** Attraverso le componenti di questa sezione è possibile combinare ed utilizzare i diversi algoritmi supervised presenti nella sezione precedente.
- **Spv Learning Assessment.** Queste componenti permettono di valutare le performance di algoritmi supervised. Questa sezione comprende, ad esempio, la cross-validation.
- **Association.** In questa sezione sono presenti una serie di operatori per la costruzione di regole di associazione.

Dalla lista abbiamo volutamente escluso la componente **Data Source**, in quanto essa oltre ad essere necessariamente posta all'inizio dello *stream diagram*, è inserita automaticamente da un *wizard*, e non è visibile nella parte dove sono presenti gli altri operatori.

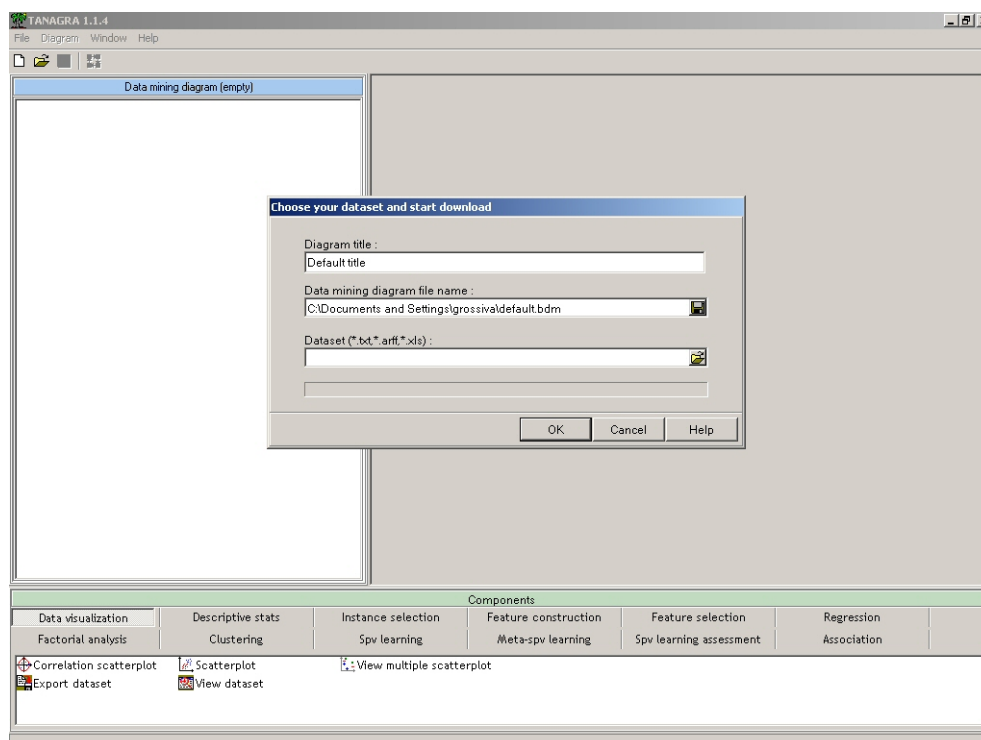


Figura 3.30 TANAGRA GUI.

La costruzione del progetto parte quindi inserendo l'operatore di **Data Source**, grazie al quale è possibile selezionare la sorgente da cui prendere i dati. TANAGRA, sotto questo punto di vista, non offre molta scelta, infatti è possibile importare data set solo in formato testuale ma con una precisa codifica del file, oppure in formato *arff* o *xls*, tipica di *Microsoft Excel*. Una caratteristica importante da evidenziare è la possibilità di importare per ogni progetto un solo data set.

Questo, a detta dell'autore, presenta due vantaggi, da un lato facilita la lettura del progetto all'utente, dall'altro semplifica il lavoro di controllo di integrità dei dati da parte dello sviluppatore. La costruzione dello *stream diagram*, una volta scelto l'input, avviene in maniera molto semplice. Attraverso l'insieme di operatori, scegliamo di volta in volta quello da inserire, e lo trasciniamo sopra al nodo presente nella *treeview*. Tale nodo diventerà il padre dell'operatore appena inserito. Ogni operatore ha parametri individuali, che è possibile settare attraverso un menù, che compare cliccando con il tasto destro del mouse sull'operatore inserito. Grazie a tale menù, è inoltre possibile eseguire l'operatore, e nel caso siano presenti delle incompatibilità tra i vari componenti inseriti, lo fa presente notificando dei messaggi

di errore. Questi messaggi però risultano poco dettagliati e raramente aiutano nella scoperta del punto in cui è presente l'errore. Da sottolineare è inoltre il fatto che ad ogni operatore è associata anche una finestra di output, accessibile sempre dal menù sopra citato (o con doppio click sull'operatore stesso), grazie alla quale è possibile conoscere il valore dei parametri settati, e/o il risultato fornito. Questa caratteristica permette quindi di accedere anche a risultati intermedi della nostra analisi.

Come è lecito aspettarsi, il sistema permette di salvare/caricare lo *stream diagram* prodotto, fornendo all'utente la possibilità di includere o meno il file utilizzato come data set nel file salvato. Più precisamente, il formato chiamato *bdm* (binary description of the stream diagram) include i dati importati nel file salvato. Il file così ottenuto è compatibile solo con TANAGRA. Il vantaggio principale di questo formato è che i dati sono importati, e quindi trasformati nel formato interno soltanto una volta, una caratteristica che rende il caricamento in memoria del data set molto veloce ad ogni successiva esecuzione del progetto. Tuttavia, il principale svantaggio è rappresentato dal fatto che tutte le analisi effettuate vengono eseguite su una copia del data set originale, quindi nel caso vi siano modifiche al data set precedentemente utilizzato, esso deve essere importato nuovamente, ed il diagramma ridefinito.

L'altro formato, *tdm* (text description of the stream diagram) non salva direttamente i dati al suo interno, bensì conserva semplicemente il riferimento al file contenente il data set preso in esame. Salvando tale riferimento, ogni modifica effettuata sul file si ripercuote sulla conoscenza estratta. In questo caso, il file generato può essere aperto ed esaminato anche attraverso un semplice text editor. Il file *tdm* può essere visto anche come una sorta di script che descrive le operazioni da fare, rispetta la specifica *INI* di *Microsoft Windows*, e permette di modificare o definire nuovi diagrammi senza utilizzare TANAGRA.

La rappresentazione scelta ricalca quella già vista in altri sistemi, ed allo stato attuale risulta sicuramente quella di maggior successo nei tool di DM. Nella versione TANAGRA di tale rappresentazione non vi è la possibilità di rinominare i vari operatori. La costruzione dello *stream diagram* purtroppo risulta poco guidata dal sistema, infatti non tutti gli operatori sono applicabili a tutti i nodi dell'albero.

Ma TANAGRA non prevede alcun metodo per indicare ciò, e quindi è possibile inserire anche un operatore inappropriato ad un certo punto dell'albero. La presenza

di un operatore inappropriato o con parametri sbagliati o incompleti porta al fallimento dell'esecuzione dell'analisi. Purtroppo, come già detto precedentemente, le informazioni fornite dal sistema per individuare e correggere le incongruenze presenti sono piuttosto scarse, e nella maggior parte dei casi inutili. Solo dopo aver preso una certa confidenza con il sistema e con il suo "modo di ragionare", si riesce ad individuare in breve tempo l'errore commesso.

Andando ad esaminare ciò che il sistema produce, c'è da sottolineare la mancanza della possibilità di salvataggio dei modelli estratti in formato di interscambio PMML, ed inoltre il sistema non presenta nemmeno, rispetto ai sistemi analizzati finora, una vasta possibilità di rappresentazione della conoscenza estratta. Vi è comunque la possibilità, attraverso il tasto *Create Report*, di salvare in formato HTML sia l'albero prodotto che tutte le informazioni fornite dai vari operatori sulla conoscenza estratta.

Bisogna inoltre evidenziare che la documentazione fornita è un po' scarsa, e che non vi è la presenza di *wizard*, - tranne che per il passo di *import* del data set -, che fornisca una base di partenza nella costruzione dello *stream diagram*. Sono presenti, comunque, dei tutorial [TAN] che aiutano ad acquisire le prime conoscenze di base sulla GUI.

Per quel che riguarda la possibilità di espansione, il sistema non prevede alcun metodo semplificato per permettere all'utente di aggiungere nuove funzionalità proprie. Questo probabilmente è dovuto al fatto che il sistema, concepito soprattutto per ambienti di ricerca e con scopi didattici, prevede che la sua espansione venga effettuata da persone con esperienza in ambito di programmazione. Tuttavia, il sistema è espandibile e personalizzabile, poiché open source. Infatti, avendo a disposizione il codice sorgente, vi è la possibilità di estenderlo e di personalizzarlo con propri algoritmi.

Questa mancanza, come per altri sistemi analizzati in precedenza, rappresenta un punto debole di TANAGRA. Non è pensabile che chi utilizza un sistema di DM debba avere necessariamente ottime conoscenze anche di programmazione, o comunque debba immergersi nel codice sorgente per capirne il funzionamento. Il sistema inoltre, non risulta portabile e funziona solo su piattaforma *Microsoft Windows*.

Concludendo, anche tralasciando caratteristiche come la possibilità di importare dati da database, o la povertà di strumenti per la visualizzazione dei risultati, in cui il sistema è sicuramente inferiore a quelli visti in precedenza, possiamo affermare che TANAGRA non aggiunge molto anche per quel che riguarda la rappresentazione utilizzata per il processo di DM. Tuttavia, amplia le possibilità di effettiva costruzione di tale processo, introducendo la modalità chiamata “drag and drop”, vista finora solo in quei sistemi che prevedevano il concetto più generico, rispetto all’albero, di flusso visto attraverso un grafo (knowledge-flow).

3.6.2 ESEMPIO DI UTILIZZO DI TANAGRA

Presentiamo ora un esempio sulla costruzione di un progetto di DM in TANAGRA. Cercheremo per coerenza di fornire un esempio del tutto simile a quelli visti per gli altri programmi. Bisogna comunque evidenziare il fatto che la funzionalità richiesta per tali esempi, ovvero la classificazione o il testing su file esplicitamente definiti dall’utente, esula dai compiti di TANAGRA, il quale si pone come scopo quello di valutare e comparare algoritmi di DM.

Per raggiungere il nostro scopo, bisogna quindi modificare il data set in input, inserendovi un nuovo attributo, al fine di discriminare le tuple viste come training set, e quelle utilizzate come test set da classificare. Questo passo è obbligato, in quanto TANAGRA può manipolare solo un data source a progetto.

sep_length	sep_width	pet_length	pet_width	type	status
5,4	3,7	1,5	0,2	Iris-setosa	learning
4,8	3,4	1,6	0,2	Iris-setosa	learning
4,8	3	1,4	0,1	Iris-setosa	learning
4,3	3	1,1	0,1	Iris-setosa	learning
5,8	4	1,2	0,2	Iris-setosa	learning
5,7	4,4	1,5	0,4	Iris-setosa	learning
5,4	3,9	1,3	0,4	Iris-setosa	learning
5,1	3,5	1,4	0,3	Iris-setosa	learning
5,7	3,8	1,7	0,3	Iris-setosa	learning
5,1	3,8	1,5	0,3	Iris-setosa	learning
5,4	3,4	1,7	0,2	Iris-setosa	learning

Figura 3.31 Introduzione dell'attributo *status* nel data set.

Nella Figura 3.31 è mostrato il data set che è lo stesso visto per gli altri esempi. È da notare, comunque, l'introduzione di un attributo *status* che ha come valori *learning* e *to_classify*, per discriminare la parte del file usato come training set da quella utilizzata per la valutazione.

Costruito il data set, dobbiamo importarlo nel progetto. TANAGRA prevede un *Dialog Box* che viene eseguito non appena si clicca sul bottone *New* (Figura 3.32). Bisogna sottolineare che questo passo è obbligatorio nella creazione di un nuovo progetto, in quanto senza il data set su cui lavorare, TANAGRA non permette l'inserimento di nessun operatore.

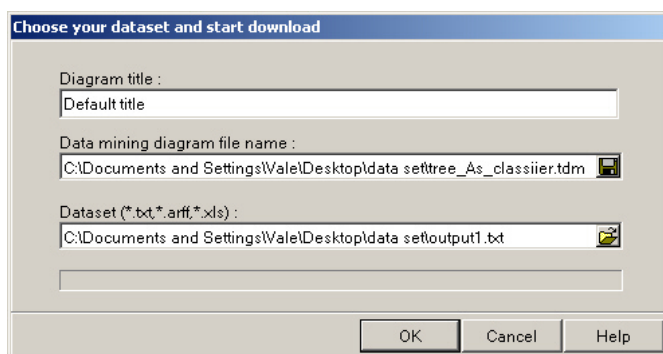


Figura 3.32 Dialog Box per la scelta del data set, titolo e path per il salvataggio del diagramma.

Una volta caricato il data set, il programma fornisce all'utente, attraverso un finestra posta sulla destra, una sintesi delle caratteristiche dei dati contenuti nel data set, come il numero di attributi e di esempi (tuple). Caricato il file, possiamo inserire un operatore per visualizzarne il contenuto. Per fare ciò accediamo al gruppo *Data Visualization*, selezioniamo l'operatore *View Dataset* e lo trasciniamo sull'unico nodo per ora presente nel progetto (*Dataset*).

Il passo successivo è quello di selezionare la parte del data set che ha funzione di training set, e definire quali sono gli attributi rilevanti, in input e target¹⁵, sui quali costruire il nostro albero di classificazione. Questo si ottiene con gli operatori *Select Example* e *Define Status*, rispettivamente dall'insieme *Instance Selection* e *Feature Selection*. La Figura 3.33 mostra sia l'albero degli operatori presenti, che i *dialog box* per la selezione di un sottoinsieme di ennuple, e quello per la selezione dei vari

¹⁵ L'attributo target rappresenta la classe da predire attraverso il classificatore.

attributi da utilizzare come input e target nella nostra costruzione di un classificatore.

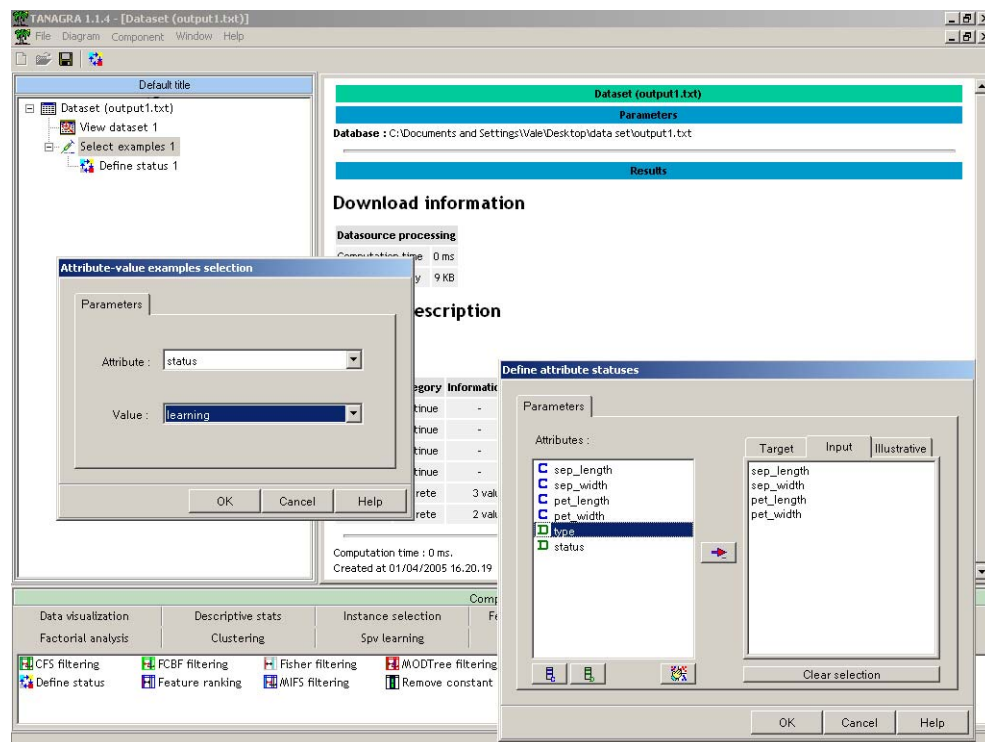


Figura 3.33 Dialog Box per la configurazione del nodo *Select Example* e *Define Status*.

Ogni operatore di classificazione in TANAGRA deve essere inserito in un meta-operatore. In questo caso scegliamo l'operatore *C-RT*, e lo inseriamo nel meta-operatore *Supervised Learning*. Non andiamo a modificare i parametri, che in questo caso rimangono quelli di default.

Inserito l'operatore, possiamo eseguire il progetto fin qui costruito, e visualizzare la conoscenza estratta (Figura 3.34), facendo doppio click sul nodo per la costruzione dell'albero di classificazione.

Values prediction			Confusion matrix				
Value	Recall	1-Precision		Iris-setosa	Iris-versicolor	Iris-virginica	Sum
Iris-setosa	1,0000	0,0000	Iris-setosa	35	0	0	35
Iris-versicolor	0,8571	0,0000	Iris-versicolor	0	30	5	35
Iris-virginica	1,0000	0,1250	Iris-virginica	0	0	35	35
			Sum	35	30	40	105

Classifier characteristics

Data partition

Growing set	70
Pruning set	35

Trees sequence (# 3)

N°	# Leaves	Err (growing set)	Err (pruning set)
3	1	0,6571	0,6857
2	2	0,3143	0,3714
1	3	0,0429	0,0571

Tree description

Number of nodes	5
Number of leaves	3

Decision tree

- pet_length < 2,6000 then type = **Iris-setosa** (100,00 % of 24 examples)
- pet_length ≥ 2,6000
 - pet_length < 4,7500 then type = **Iris-versicolor** (100,00 % of 21 examples)
 - pet_length ≥ 4,7500 then type = **Iris-virginica** (88,00 % of 25 examples)

Figura 3.34 Insieme di statistiche ed albero di classificazione prodotti dal sistema.

L'ultimo passo riguarda la classificazione, ovvero l'applicazione della conoscenza estratta. Ciò si ottiene facilmente, inserendo l'operatore *Recover Examples* da *Instance Selection*, che effettua la vera e propria classificazione, e l'operatore *View Dataset*, per vedere la classe predetta, inserita come ultima colonna del data set. In Figura 3.35 è mostrata la *treeview* finale del progetto.

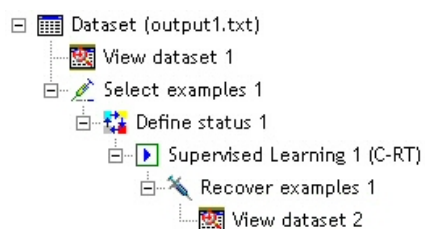


Figura 3.35 Treeview degli operatori del nostro progetto.

Possiamo eseguire nuovamente il progetto, e cliccando sull'elemento *View Dataset*, visualizzare la classe predetta per ogni ennupla (Figura 3.36).

	sep_len	sep_wid	pet_len	pet_wid	type	pred_SpvInstance_1
106	5,1	3,5	1,4	0,2	Iris-setosa	Iris-setosa
107	4,9	3	1,4	0,2	Iris-setosa	Iris-setosa
108	4,7	3,2	1,3	0,2	Iris-setosa	Iris-setosa
109	4,6	3,1	1,5	0,2	Iris-setosa	Iris-setosa
110	5	3,6	1,4	0,2	Iris-setosa	Iris-setosa
111	5,4	3,9	1,7	0,4	Iris-setosa	Iris-setosa
112	4,6	3,4	1,4	0,3	Iris-setosa	Iris-setosa
113	5	3,4	1,5	0,2	Iris-setosa	Iris-setosa
114	4,4	2,9	1,4	0,2	Iris-setosa	Iris-setosa
115	4,9	3,1	1,5	0,1	Iris-setosa	Iris-setosa
116	4,8	3	1,4	0,3	Iris-setosa	Iris-setosa
117	5,1	3,8	1,6	0,2	Iris-setosa	Iris-setosa
118	4,6	3,2	1,4	0,2	Iris-setosa	Iris-setosa
119	5,3	3,7	1,5	0,2	Iris-setosa	Iris-setosa
120	5	3,3	1,4	0,2	Iris-setosa	Iris-setosa
121	7	3,2	4,7	1,4	Iris-versicolor	Iris-versicolor
122	6,4	3,2	4,5	1,5	Iris-versicolor	Iris-versicolor
123	6,9	3,1	4,9	1,5	Iris-versicolor	Iris-virginica
124	5,5	2,3	4	1,3	Iris-versicolor	Iris-versicolor
125	6,5	2,8	4,6	1,5	Iris-versicolor	Iris-versicolor
126	5,7	2,8	4,5	1,3	Iris-versicolor	Iris-versicolor
127	6,3	3,3	4,7	1,6	Iris-versicolor	Iris-versicolor
128	4,9	2,4	3,3	1	Iris-versicolor	Iris-versicolor
129	6,6	2,9	4,6	1,3	Iris-versicolor	Iris-versicolor
130	5,2	2,7	3,9	1,4	Iris-versicolor	Iris-versicolor
131	5,7	3	4,2	1,2	Iris-versicolor	Iris-versicolor
132	5,7	2,9	4,2	1,3	Iris-versicolor	Iris-versicolor
133	6,2	2,9	4,3	1,3	Iris-versicolor	Iris-versicolor
134	6,5	3,5	4,9	1,4	Iris-versicolor	Iris-versicolor

Figura 3.36 Risultato della classificazione. Nell'ultima colonna è presente la classe predetta dal classificatore.

Nel prossimo capitolo proporremo una comparazione dei vari tool presentati, esaminandoli sotto vari punti di vista, che comprendono sia il linguaggio grafico con cui vengono presentati i processi KDD, sia considerazioni di carattere generale. Forniremo ulteriori esempi a sostegno della nostra analisi.

Capitolo 4

COMPARAZIONE

4.1 INTRODUZIONE

In questo capitolo metteremo a confronto i vari tool per KD precedentemente introdotti. Il nostro obiettivo è quello di fornire una valutazione di quelli che sono i requisiti che una metafora grafica, e più in generale un'interfaccia grafica, dovrebbe soddisfare in questo settore. Tali requisiti costituiranno una serie di linee guida per lo sviluppo di una metafora grafica per KDDML.

Prima di entrare nella discussione, premettiamo che i vari tool analizzati non coprono tutti esattamente la stessa area di problematiche di DM, che peraltro non presenta nemmeno confini ben definiti e precisi (par. 1.4). Pertanto, non terremo in considerazione il numero di algoritmi presenti nelle varie interfacce, anche se, per completezza segnaliamo che WEKA e YALE¹⁶ affrontano un numero elevato di problematiche, fornendo una vasta gamma di algoritmi per la loro risoluzione.

Al contrario, in questo capitolo ci concentreremo nell'identificare un nucleo comune di funzionalità, previste o ritenute importanti, grazie alle quali riuscire a confrontare i vari programmi, talvolta facendo riferimento a problematiche specifiche.

Le funzionalità a cui faremo riferimento nel seguito del capitolo riguardano principalmente:

¹⁶ YALE in parte utilizza anche algoritmi WEKA

- **la modalità di accesso ai dati (par. 4.2)**, in riferimento alla caratteristica di un tool di poter accedere a dati memorizzati in un database o di utilizzare più data set contemporaneamente.
- **la costruzione e la rappresentazione grafica di una query (par. 4.3)**, in cui si analizzano vari fattori che rendono la costruzione e la rappresentazione di un processo KDD più intuitiva possibile.
- **l'esecuzione e la validazione della query (par. 4.4)**, in cui si cerca di evidenziare gli strumenti offerti dai vari software per favorire l'utente nella stesura di query corrette, e la successiva esecuzione delle stesse.
- **la visualizzazione dei dati e dei risultati (par. 4.5)**, in cui si analizzano gli strumenti di output offerti dai vari tool.
- **l'estendibilità dei sistemi (par. 4.6)**, in cui trattiamo le proprietà dei vari tool per favorire l'espansione del sistema, con l'inserimento di nuove caratteristiche non previste in origine.
- **il supporto all'utente (par. 4.7)**, ovvero la quantità ed il tipo di documentazione fornita a corredo del sistema.

Il capitolo si concluderà con l'esposizione di alcune considerazioni finali, attraverso le quali indicheremo una serie di **requisiti** e suggerimenti.

4.2 MODALITÀ DI ACCESSO A DATI E MODELLI

Gli esempi illustrati nel capitolo 3 ci forniscono lo spunto per iniziare la nostra trattazione parlando delle modalità d'importazione dei data set nelle varie GUI.

Tralasciando il numero di formati supportati, su due punti riteniamo necessario porre la nostra attenzione:

1. Possibilità di interfacciamento a database.
2. Possibilità di utilizzare più data set contemporaneamente nei nostri esperimenti.

Per quanto riguarda il primo punto, bisogna dire che esula dalla nostra trattazione sulla metafora grafica, ma non da una valutazione complessiva del tool, in quanto in alcuni casi potremmo avere la necessità di accedere a dati non localmente presenti, e scaricare solo le ennuple che ci interessano.

Brevemente diciamo che non tutti i sistemi analizzati prevedono la possibilità di interfacciarsi ad un database, infatti, almeno per il momento sia TANAGRA che ORANGE ne sono sprovviste. La mancanza di tale caratteristica, rispetto ad una visione complessiva del sistema, e non relativa alla sola interfaccia, potrebbe in taluni casi rappresentare una notevole limitazione.

Il secondo punto, pur non essendo strettamente legato alla sola metafora grafica, richiede un maggiore approfondimento. Dagli esempi è emerso che non tutti i tool possono lavorare con più data set diversi contemporaneamente. Come abbiamo visto in 3.6, TANAGRA permette di gestire solo un data set per volta, quindi anche un'operazione piuttosto semplice come una validazione, o una classificazione su un data set definito esplicitamente dall'utente, e non scelto dal sistema come sub set dell'originale, richiede modifiche al data set. Tale strada può essere percorribile in caso di piccoli data set, ma se siamo in presenza di migliaia di ennuple, presenta maggiori difficoltà.

Anche GhostMiner Developer prevede che per ogni progetto sia presente un solo nodo, denominato *Dataset Information*, che fornisce informazioni sull'unico training data set utilizzabile per l'estrazione della conoscenza.

In entrambe i casi, il nodo d'importazione del data set rappresenta la radice del progetto, ed è inserito automaticamente dal sistema come passo iniziale. GhostMiner Developer permette, comunque, al contrario di TANAGRA, di fornire esplicitamente al sistema un data file su cui eseguire test, per verificare l'accuratezza

del modello estratto. Quindi, in entrambe i casi abbiamo una metafora ad albero, che ha origine sui dati che vogliamo utilizzare per estrarre il nostro modello¹⁷, ed entrambe permettono di costruire modelli solo su un data set a progetto.

Tale scelta non viene condivisa da YALE, il quale pur utilizzando la metafora ad albero per la visualizzazione dell'esperimento, non vincola l'utente ad iniziare il proprio progetto scegliendo il data set su cui lavorare, e comunque permette, come mostrato dall'esempio nel capitolo precedente (par. 3.3.2), di assegnare al sistema diversi data set su cui effettuare le varie operazioni.

Dal nostro punto di vista, questa sembra la strada migliore, in quanto non è detto che un utente voglia necessariamente estrarre della conoscenza, ma potrebbe anche voler costruire un esperimento, semplicemente importando modelli già costruiti ed applicarli a dati propri.

La Figura 4.1 mette in evidenza i diversi criteri adottati dai tool appena menzionati. In Figura 4.1.b e Figura 4.1.c sono rappresentati gli alberi creati rispettivamente con TANAGRA e GhostMiner Developer.

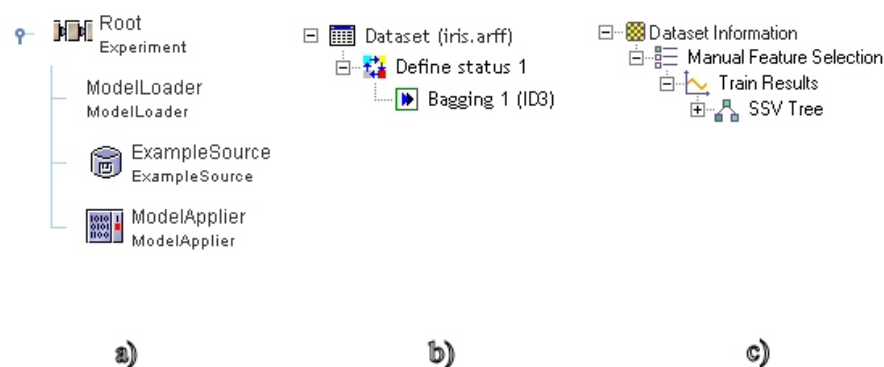


Figura 4.1 Treeview rispettivamente di YALE (1.a), TANAGRA (1.b) e GhostMiner Developer (1.c).

Come si può notare, entrambe hanno l'import dei dati come radice dell'albero, ed estraggono i modelli richiesti da un unico data set. La Figura 4.1.a mostra l'albero di

¹⁷ Bisogna comunque dire che, anche se entrambe lavorano con un unico data set a progetto, GhostMiner Developer permette di applicare la conoscenza precedentemente estratta grazie a GhostMiner Analyzer.

YALE, in questo caso un semplice esempio di applicazione di un modello precedentemente estratto. Come possiamo notare, YALE lascia maggior libertà all'utente sulla costruzione dell'esperimento. È da evidenziare, inoltre, che attraverso il nodo denominato *Root* è possibile settare alcuni parametri generali dell'intero progetto.

WEKA e ORANGE non hanno alcuna limitazione a priori, sul numero di data set che l'utente può utilizzare, ma prevedono comunque che in ogni flusso vi sia almeno un nodo di input.

Riguardo ai modelli, purtroppo, tranne YALE, nessun altro sistema presenta la possibilità di caricare un modello precedentemente estratto. Questo rappresenta una grossa limitazione, in quanto non è pensabile dover estrarre ex-novo un modello, ogni volta che lo si deve utilizzare. L'estrazione del modello potrebbe richiedere una notevole quantità di tempo, e per questo dovrebbe essere possibile salvare ed importare i modelli che vengono estratti nel corso del processo KDD.

4.3 COSTRUZIONE E RAPPRESENTAZIONE DI UNA QUERY

Continuiamo il nostro confronto tra le varie interfacce, parlando di quella che chiameremo costruzione della query¹⁸, per utilizzare un termine generale, applicabile a tutte le interfacce.

La costruzione della query è sicuramente il punto più delicato da affrontare nella concezione di un'interfaccia grafica. Tutte le interfacce analizzate si pongono l'obiettivo di rendere tale processo il più intuitivo e semplice possibile, al fine di permettere, anche a chi non ha conoscenze specifiche di programmazione, di sfruttare le potenzialità offerte dal sistema.

La prima osservazione che dobbiamo fare è la mancanza di uno standard ben definito per tale processo. Ciò è dovuto a molteplici fattori, tra cui assume particolare rilevanza la mancanza di un linguaggio standardizzato per il DM, come può essere SQL nel mondo dei database, sul quale concepire e costruire intere applicazioni grafiche.

¹⁸ In realtà non tutti i tool analizzati possiedono un vero e proprio query language. Quindi, per questi ultimi il termine potrebbe risultare improprio.

Successivamente ci concentreremo sull'analisi di alcune caratteristiche specifiche, come la presenza di un *wizard*, la rappresentazione del flusso dei dati e l'immissione dei parametri, ed avvalendoci di alcuni esempi, cercheremo di approfondire alcune problematiche che potrebbero emergere. La conclusione del paragrafo sarà dedicata ad una valutazione dell'espressività delle metafore grafiche prese in esame.

4.3.1 RAPPRESENTAZIONE DEL FLUSSO DI DATI: ALBERO vs GRAFO

La metodologia di rappresentazione influisce sul metodo di costruzione della query. A questo proposito è utile fornire una distinzione tra vari metodi di rappresentazione del flusso dei dati. Le interfacce analizzate, sulla base di questo criterio, possono essere classificate in due famiglie:

- quelle che prevedono un grafo.
- quelle che prevedono un albero.

GRAFO

Di questa categoria fanno parte WEKA Knowledge Flow e ORANGE, che presentano la stessa modalità di costruzione. In questo caso, l'utente si trova di fronte alla necessità di costruire un grafo orientato. Per quel che riguarda la costruzione di tale grafo, WEKA Knowledge Flow non offre molto aiuto all'utente, il quale, almeno in una fase iniziale, e fino a che non ha raggiunto una certa confidenza col sistema, deve agire per tentativi, nel cercare quali sono i giusti operatori da concatenare¹⁹.

ORANGE, al contrario, favorisce l'utente nella stesura della query, visualizzando un *tooltip*, in cui sono presenti i segnali di input ed output gestiti da quel determinato nodo.

La caratteristica oggettiva che riguarda questo tipo di costruzione è la maggiore complessità rispetto a quella ad albero. In questo caso, l'utente non solo deve

¹⁹ WEKA Knowledge Flow fornisce comunque informazioni sull'utilizzo dei vari parametri previsti per l'operatore appena inserito.

inserire gli operatori, ma deve anche connetterli. Pertanto, dal punto di vista della costruzione, vi è la necessità di un passo ulteriore rispetto alla metafora ad albero. È chiaro che questa maggiore difficoltà può influire sull'usabilità e sull'immediatezza della GUI stessa.

ALBERO

GhostMiner Developer, YALE e TANAGRA presentano una visione di tipo ad albero, che introduce anche una gerarchia tra operatori. Ad esempio, nel caso di TANAGRA e GhostMiner Developer, l'operatore di input è sicuramente quello di grado maggiore, poiché in assenza di esso, non è possibile introdurre nessun altro operatore, mentre YALE prevede che alcuni operatori (nodi dell'albero) non possano avere figli.

La costruzione della query in questi tool richiede quindi all'utente solo l'inserimento dell'operatore, e non necessita dell'introduzione di connessioni tra gli operatori. La posizione di quest'ultimo all'interno dell'albero mostra il tipo di dato che lo attraversa, e la gerarchia prevista tra questi evita l'inserimento di operatori incompatibili in alcuni punti dell'albero²⁰.

Questa visione conduce ad una costruzione più ordinata, ed a nostro avviso più intuitiva, rispetto a quella a grafo, in cui un utente può, in una prima fase introdurre tutti gli operatori, e solo alla fine connetterli²¹.

Inoltre è da evidenziare il fatto che i due tool che utilizzano il grafo prevedono l'inserimento (e la successiva connessione) in modo esplicito da parte dell'utente, di nodi per la visualizzazione delle informazioni ritenute importanti. Questo fatto può rappresentare un'ulteriore difficoltà, sia per capire l'operatore più adatto al tipo di informazione da visualizzare, sia per la maggiore complessità del grafo risultante, mentre col tempo garantisce comunque maggiore libertà e permette all'utente di scegliere solo le informazioni che ritiene realmente rilevanti.

²⁰ All'utente rimane quindi la scelta del punto nell'albero dove inserire l'operatore. La scelta di tale punto è comunque "guidata" dal sistema, che evita di annidare operatori incompatibili.

²¹ Nel caso di una visualizzazione ad albero, la connessione è implicita e creata automaticamente dal sistema nel momento in cui si inserisce l'operatore.

GhostMiner Developer fornisce in modo automatico tutte le informazioni, sia sul data set utilizzato che sulla conoscenza estratta, come risultato dell'esecuzione della query, senza l'utilizzo esplicito di opportuni operatori. Questa è una filosofia completamente opposta alla precedente, e bisogna comunque sottolineare che, mentre dal punto di vista dell'immediatezza e della facilità d'uso rappresenta un'ottima caratteristica, in questo caso vi può essere un'eccessiva offerta di informazione, che può distogliere l'attenzione dell'utente dalle informazioni veramente rilevanti.

YALE e TANAGRA rappresentano una soluzione intermedia, e secondo noi quella più consona, in quanto introducono automaticamente alcune informazioni fondamentali, sia sul data set utilizzato, sia sulla conoscenza estratta, mentre lasciano all'utente la possibilità di inserire operatori per visualizzare specifiche caratteristiche. Presentano quindi vantaggi sia rispetto alla prima filosofia, poiché un utente che si avvicina per la prima volta al software ha un riscontro su ciò che sta producendo, sia sulla seconda, in quanto evitano di fornire informazioni non richieste, che possono risultare irrilevanti.

4.3.2 PRESENZA DI UN WIZARD

Una prima osservazione su cui porre la nostra attenzione riguarda la presenza, prevista dai vari software, di strumenti per agevolare l'utente nella stesura della query. A questo proposito, possiamo affermare che l'unico tool che fornisce un *wizard* per costruire la query, e non solo per l'import del data set²², è YALE (Figura 4.2). Grazie a tale meccanismo, è possibile generare una query, semplicemente settando alcuni parametri richiesti dal *wizard* nelle varie schermate. Inoltre, tale *wizard* risulta facilmente espandibile, poiché, attraverso alcuni semplici meccanismi della GUI, è possibile personalizzare ed espandere a proprio piacimento le varie tipologie di *template* fornite.

²² Come TANAGRA e GhostMiner Developer.

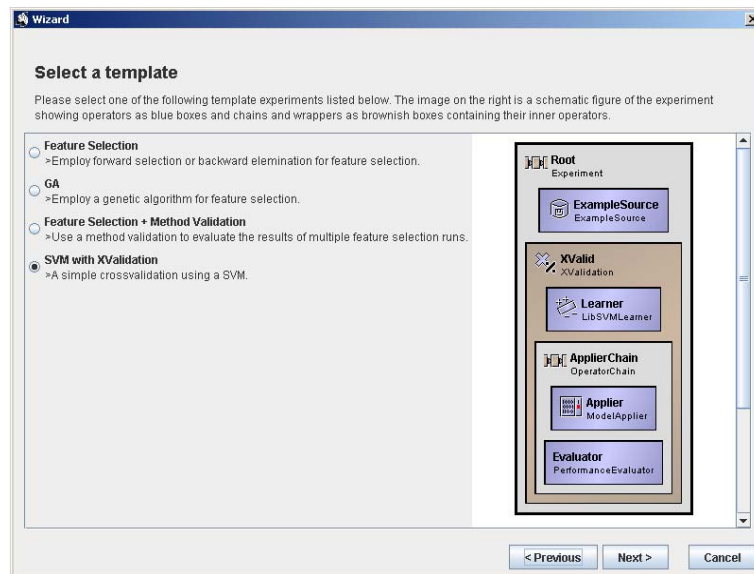


Figura 4.2 Wizard di YALE.

Questo rappresenta indubbiamente un vantaggio, in quanto avere un meccanismo che fornisca un'intelaiatura di quelle che possono essere le tipologie di query di DM più diffuse, è utile in qualsiasi interfaccia, e può facilitare notevolmente l'utente nell'utilizzo iniziale del sistema. La costruzione di un buon *wizard* è comunque un passo successivo alla concezione di un buon linguaggio grafico e di un buon metodo di costruzione.

4.3.3 IMMISSIONE DEI PARAMETRI

La filosofia di visualizzazione scelta sembra non influenzare le applicazioni, per quel che riguarda l'immissione dei parametri funzionali dei vari operatori. Da questo punto di vista, infatti non si registrano notevoli differenze, in quanto WEKA Knowledge Flow, ORANGE, TANAGRA e GhostMiner Developer adottano scelte affini. In taluni casi cliccando sull'operatore, in altri utilizzando l'opportuno bottone, compare un menù, attraverso il quale è possibile settare i parametri previsti per tale operatore.

YALE invece utilizza per ogni operatore la parte destra della *treeview*, e semplicemente selezionando un nodo, visualizza in tale area gli eventuali parametri riguardanti il nodo prescelto, includendo sia una lista parziale di parametri, sia una lista completa per utenti più esperti. Tale politica risulta quindi leggermente diversa

rispetto a quella presentata dagli altri software, in quanto permette di visualizzare i parametri automaticamente, favorendone una più rapida consultazione, migliorando a nostro avviso anche l'usabilità dello strumento stesso, che presenta una forma più ordinata e meno complessa, rispetto ad applicazioni come WEKA Knowledge Flow e ORANGE. Questi ultimi, infatti, permettono di aprire più dialog box di gestione dei parametri contemporaneamente, con il rischio di confondere l'utente tra i vari menù riguardanti operatori diversi.

4.3.4 LIMITAZIONI E POTENZIALITÀ DEI VARI TOOL

Riprendendo quanto è stato detto nei paragrafi precedenti, possiamo affermare che, per quanto riguarda la sola costruzione della query, l'albero rappresenta la metafora grafica più semplice ed immediata. Tuttavia, sappiamo che l'espressività di un albero è minore rispetto a quella di un grafo, essendo quest'ultimo una generalizzazione del concetto più restrittivo di albero²³.

Quello che ci proponiamo di chiarire adesso è se questa limitazione si riflette anche sulle potenzialità che un tool offre, oppure ne identifica solo una diversa rappresentazione.

Ad una prima analisi superficiale, basata solo sugli esempi visti nelle sezioni precedenti, considerando solo TANAGRA e GhostMiner Developer, potremmo dire che quello che si verifica è la prima ipotesi. Infatti, se ci limitiamo a quei tool che lavorano su un solo data set, sicuramente siamo indotti a pensare che anche la potenzialità del tool ricalca fedelmente la struttura scelta per visualizzare e costruire la nostra query.

Ampliando la nostra analisi anche a YALE, la questione diventa sicuramente più complessa, poiché quest'ultimo, come abbiamo osservato in precedenza, pur adottando una metafora ad albero, non presenta le stesse limitazioni dei due software precedenti.

²³ In realtà nei tool analizzati siamo in presenza di grafi orientati ed aciclici, e la sostanziale differenza è rappresentata dal fatto che nel caso dell'albero, ogni nodo, tranne la radice, ha al massimo un padre, mentre nel caso del grafo ciò non si verifica.

Il nostro compito diventa quindi quello di valutare se tale limitazione è dovuta semplicemente ad una politica di implementazione - la maggiore semplicità dell'albero e la gerarchia più rigida portano sicuramente ad una più semplice implementazione - o alla vera e propria limitazione introdotta dall'applicazione in senso stretto²⁴ della struttura utilizzata per creare e visualizzare la query.

Purtroppo, al fine della nostra trattazione, dobbiamo constatare che YALE è l'unico tra i tool analizzati ad avere alla sua base un vero e proprio *data mining query language*. Attraverso questa caratteristica è possibile effettuare un'analisi più approfondita, distinguendo quella che è la sola rappresentazione dalla vera struttura a basso livello della query, ed osservando come i due concetti sono legati. Cerchiamo adesso di mostrare le differenti interpretazioni di YALE rispetto a TANAGRA e GhostMiner, illustrando una serie di esempi.

4.3.5 RAPPRESENTAZIONE DELLA QUERY: ALCUNI ESEMPI

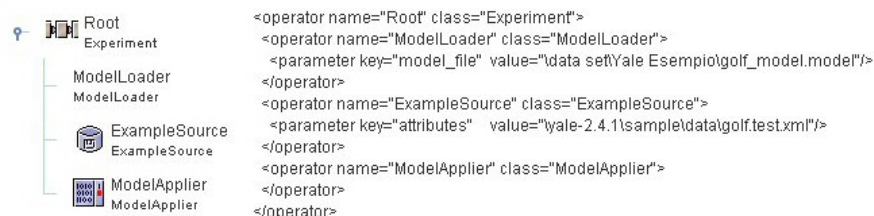


Figura 4.3 Catena di operatori YALE per la lettura e l'applicazione di un modello e rispettiva versione in XML.

Iniziamo il nostro esempio, riproponendo una figura in parte già vista all'inizio del paragrafo 4.2. La Figura 4.3 mostra la costruzione di una query in YALE, che applica semplicemente la conoscenza, ottenuta attraverso un modello precedentemente estratto. Questa possibilità rende YALE unico rispetto agli altri software presi in considerazione. Il nodo *ModelApplier* richiede che tra i suoi immediati predecessori nella catena, vi siano gli operatori per leggere il modello da applicare (*ModelLoader*) ed il file contenente i dati su cui attuare la predizione

²⁴ YALE, pur utilizzando una metafora ad albero, potrebbe, a nostro avviso, adottare tecniche meno restrittive per risolvere alcune problematiche.

(*ExampleSource*). In caso contrario, il sistema avvisa l'utente di tale mancanza attraverso la finestra di output, nel momento in cui si appresta a validare la query.

Questo esempio mostra, quindi, che YALE utilizza la struttura ad albero solo per visualizzare l'esperimento. Da un punto di vista logico infatti, l'operatore *ModelApplier* ha in realtà due nodi in input, e non soltanto uno, come la struttura ad albero richiede. Inoltre, la query XML risultante appare più come una catena di operatori allo stesso livello, che come una serie di operatori annidati, in cui l'input di uno diventa l'output dell'altro.

Tale caratteristica viene messa in risalto anche dal seguente esempio, in cui cerchiamo di estrarre due modelli dalla stessa sorgente.

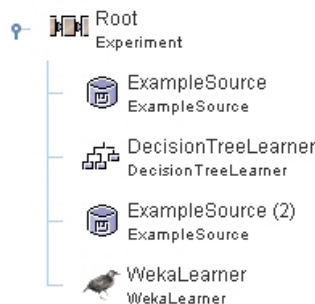


Figura 4.4 Albero degli operatori per l'estrazione di due modelli in YALE.

Come si può notare dalla Figura 4.4, YALE richiede che, oltre alla presenza dei nodi riguardanti il tipo di modello che vogliamo estrarre, vi siano anche due nodi per leggere l'input. Ogni learner, infatti consuma l'input ottenuto dall'operatore *ExampleSource* che lo precede nella catena. Anche se vogliamo estrarre la conoscenza dal medesimo data set, dobbiamo inserire due operatori che forniscano l'input ai learner.

Vediamo adesso invece come WEKA, che non condivide con YALE la struttura ad albero, e TANAGRA attuano lo stesso procedimento.

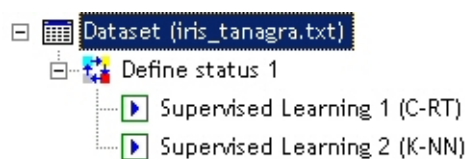
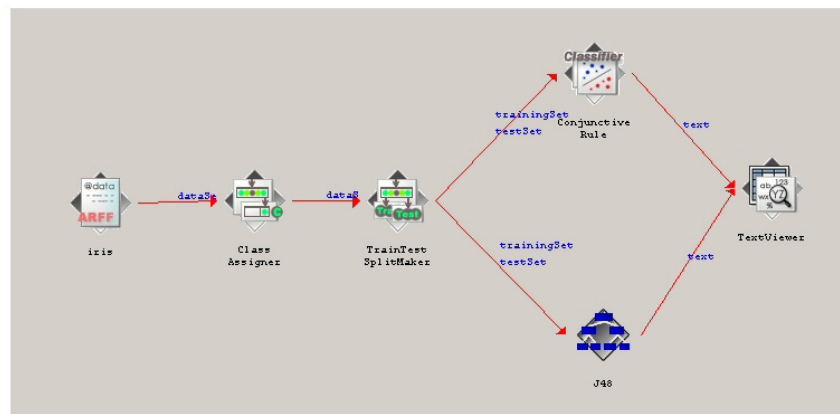


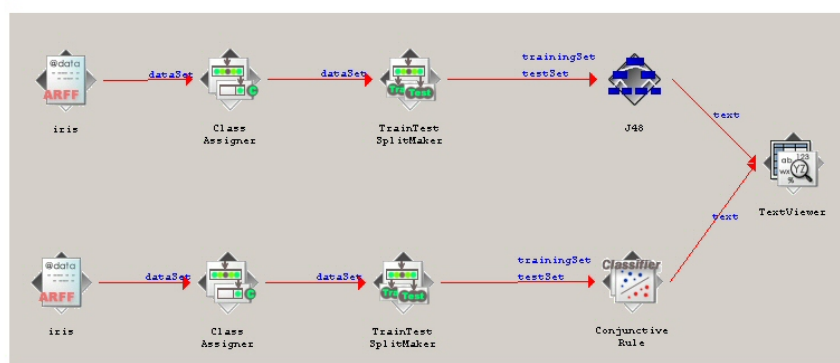
Figura 4.5 Albero degli operatori per l'estrazione di due modelli in TANAGRA.

Come mostrato in Figura 4.5, TANAGRA adotta anche in questo caso una metafora ad albero pura. I due learner estraggono la conoscenza dall'unico data set, ed entrambe utilizzano i dati ereditati dal nodo padre, mettendo anche in luce una chiara limitazione di tale concezione, ovvero l'impossibilità di estrarre la conoscenza da due data set distinti in un unico progetto.

In Figura 4.6 è rappresentata in WEKA la stessa query vista per TANAGRA e YALE. La parte a) mostra il flusso rappresentante la query attuabile in TANAGRA, mentre la parte b) quella relativa a YALE.



a)



b)

**Figura 4.6. a) Estrazione di due modelli dallo stesso data set.
b) Estrazione di due modelli da data set indipendenti in WEKA.**

Il confronto con WEKA ci permette di poter concludere che, anche se graficamente YALE rappresenta le sue query con una struttura molto simile a quella di un albero, la sua concezione logica, vista anche attraverso il file XML, è molto più simile ad un flusso caratterizzato da catene di operatori tutti allo stesso livello di annidamento.

L'esempio fornito, infatti, mette in evidenza la diversa concezione logica di TANAGRA e YALE. Quest'ultimo rappresenta la query semplicemente attraverso due flussi separati, che a differenza di TANAGRA, possono estrarre contemporaneamente la conoscenza anche da dati diversi. Una limitazione dell'approccio di YALE riguarda la possibilità che l'input ai due learner non sia un data set presente già sul disco, ma un data set ottenuto dopo una lunga fase di preprocessing. In questo caso l'utente potrebbe trovarsi di fronte al dover costruire due processi separati, il primo solo per la fase di preprocessing, che permetta di salvare il data set su disco, il secondo simile a quello appena visto, per permettere l'estrazione della conoscenza da tali dati preprocessati.

Purtroppo, queste osservazioni non sono sufficienti ad offrire una visione netta e definitiva delle differenti strategie di interpretazione dell'albero di YALE rispetto alle altre. Come vedremo tra poco, infatti, si verificano alcuni casi in cui la struttura tipica ad albero viene fuori anche in YALE, invalidando in parte la conclusione a cui potremmo facilmente arrivare, affermando che YALE utilizza l'albero solo per avere una strutturazione più ordinata dell'esperimento, che non è altro che un flusso molto simile a quello di WEKA o ORANGE.

A tal fine, nel prossimo paragrafo introdurremo meccanismi più complessi, tipici del processo KDD, come la validazione di classificatori e la costruzione e rappresentazione di meta-classificatori.

4.3.6 RAPPRESENTAZIONE DI COSTRUTTI COMPLESSI

Inseriamo adesso tale discorso, legato alla valutazione delle differenti strategie di interpretazione dell'albero di YALE, in uno più ampio, andando ad osservare come i vari tool affrontano la costruzione e la rappresentazione grafica di metodologie

tipiche più complesse di quelle esaminate negli esempi sinora trattati, in modo da poter valutare ed incrementare l'accuratezza dei modelli estratti.

Tali metodologie risultano interessanti dal punto di vista grafico, poiché richiedono l'esecuzione ripetuta di una serie di operatori. Cerchiamo quindi di capire come i vari software espongono concetti come K-cross-validation (1.4.2), ed in modo più generale la meta-classificazione, ovvero l'utilizzo ripetuto di un classificatore (o più classificatori), per ottenere un grado di accuratezza più elevato rispetto a quello ottenibile dall'utilizzo di un singolo classificatore.

Senza entrare troppo in dettaglio, possiamo affermare che, per quanto riguarda la costruzione di tali processi, GhostMiner Developer presenta le caratteristiche più interessanti. Ricordiamo che, nella costruzione di tali processi, è fondamentale una corretta interazione con l'utente, il quale, soprattutto nel caso della costruzione di meta-classificatori, deve scegliere tra innumerevoli parametri. Nel caso di quest'ultimo software, la costruzione di tali processi è completamente guidata dal sistema, che indirizza l'utente alla corretta immissione dei parametri richiesti attraverso alcuni menù.

Per quanto riguarda la rappresentazione, GhostMiner Developer raffigura la K-cross-validation attraverso un nodo inserito nel progetto. Grazie ai figli di questo nodo, è possibile accedere ai dati forniti dalla validazione, come l'accuratezza sui training set e sui test set. GhostMiner Developer fornisce inoltre il modello estratto ed utilizzato nella validazione²⁵. Non è possibile però fornire al sistema un modello precedentemente estratto da validare, in quanto ogni volta che si attua una K-cross-validation, GhostMiner richiede anche il tipo di classificatore da utilizzare.

Nel caso di meta-classificatori²⁶, il sistema, oltre a fornire il risultato finale, genera per ogni classificatore richiesto dall'utente un sottoalbero, attraverso il quale è possibile accedere ai singoli modelli estratti che hanno permesso la costruzione del meta-modello finale, come raffigurato in Figura 4.7.

²⁵ Alcuni algoritmi, come abbiamo visto dall'esempio fornito nel paragrafo 3.5.2 del capitolo precedente, dedicata a GhostMiner, prevedono che l'estrazione del modello da attuare nella validazione sia basata su una *K-cross-validation*. Questi permettono anche di accedere ad ogni singolo passo di *validation training*.

²⁶ GhostMiner presenta a questo proposito due metodi, *Committee* e *K-Classifier*.

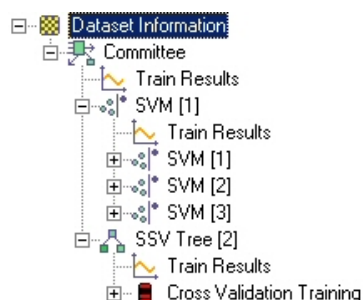


Figura 4.7 Meta-classificatore costruito con GhostMiner Developer.

L'altro software che mostra ottime potenzialità a questo proposito è sicuramente WEKA. Quest'ultimo ha a disposizione una vasta gamma di meta-classificatori, ognuno dei quali viene rappresentato da un nodo. Rispetto a GhostMiner Developer, non vi è in questo caso la possibilità di visualizzare i singoli modelli estratti, ma viene presentato solo il risultato finale. Dal punto di vista visivo, non è garantita quindi molta espressività, in quanto un meta-classificatore è semplicemente un nodo, ma non vengono rappresentati i classificatori che concretizzano il singolo meta-classificatore. Purtroppo la costruzione, se comparata a GhostMiner Developer, risulta di gran lunga più complessa, a causa della scarsa immediatezza dei vari menù per l'immissione dei parametri richiesti.

Per quanto riguarda la K-cross-validation, le informazioni fornite sono in gran parte simili a quelle garantite da GhostMiner Developer. Anche in questo caso, la K-cross-validation è rappresentata semplicemente da un nodo del grafo.

Al di là delle metodologie offerte, che non si discostano molto da quelle previste dai software precedentemente menzionati, YALE non fornisce, a nostro avviso, un adeguato supporto nella fase di costruzione dei meta-modelli e delle validazioni. Purtroppo il sistema risulta poco intuitivo, ed ha una complessità maggiore rispetto agli altri tool. Bisogna sottolineare il fatto che nel caso di validazioni, meta-classificazioni, o operazioni iterate, emerge la vera natura ad albero dell'esperimento YALE. Come possiamo vedere nella Figura 4.8, al contrario degli esempi illustrati fino ad ora, in cui era presente solo una catena di operatori, qui siamo in presenza di un vero e proprio albero. Ciò si riflette anche sul file XML risultante, attraverso il quale si può notare che anche la rappresentazione testuale prevede un annidamento

delle operazioni uguale alla rappresentazione grafica. Per completezza, mostriamo anche lo schema riguardante l'annidamento dei vari operatori previsti nella query.



Figura 4.8 Comparazione tra metafora grafica, file XML risultante e disegno del *nesting* degli operatori in YALE.

Anche TANAGRA offre buone caratteristiche di costruzione, certamente più intuitive rispetto a YALE, mentre per quanto riguarda le metodologie attuabili, non si discosta molto da quelle dei software precedentemente descritti.

ORANGE, al contrario, non offre grossi mezzi per risolvere le problematiche suddette, in quanto non presenta la possibilità di costruire meta-classificatori, e per quel che riguarda la K-cross-validation, sia nella costruzione che nella rappresentazione, ricalca quella vista per WEKA.

Continuiamo, quindi, dicendo che in base agli esempi appena illustrati ed alle problematiche affrontate, possiamo affermare che, utilizzando l'albero in maniera troppo restrittiva, si possono avere delle limitazioni nelle potenzialità che un tool offre, rispetto alle possibilità offerte da quelli che utilizzano il grafo.

Sotto questo punto di vista merita particolare attenzione YALE, il quale, pur condividendo una struttura ad albero, sfrutta in maniera più proficua i vantaggi di tale struttura nella stesura della query e nella rappresentazione di esperimenti

complessi, mentre evita le limitazioni tipiche di tale costruzione, proponendo una concezione mista sia di visualizzazione che di costruzione reale della query.

4.3.7 ESPRESSIVITÀ ED IMMEDIATEZZA DELLE RAPPRESENTAZIONI

Fino ad ora ci siamo posti il problema di valutare la metafora grafica solo dal punto di vista della costruzione degli esperimenti, ma tale discorso deve essere necessariamente integrato con quello che riguarda l'espressività generale di tutta la rappresentazione, intesa anche come immediatezza e facilità di lettura di ciò che viene prodotto, o ci troviamo ad analizzare.

Sotto questo punto di vista, la situazione si complica, in quanto, se da una parte l'albero risulta più semplice e più intuitivo, dall'altra il grafo sembra fornire maggiori caratteristiche di espressività ed immediatezza. Per quel che riguarda la sola rappresentazione della query, quindi slegata dalla sua costruzione, il grafo fornisce oggettivamente una metafora più potente e di facile lettura, poiché ad esempio, partendo da uno dei nodi in input, permette di capire quali sono i nodi che vengono attraversati, e come viene trasformata la conoscenza iniziale, semplicemente seguendo gli archi.

Nell'albero, in caso di esperimenti complessi, con un elevato *nesting* degli operatori, risulta più difficile, o per meglio dire, meno immediato²⁷ capire che tipo di informazione attraversa i vari operatori. Purtroppo, la facilità di lettura del grafo dipende anche da fattori di tipo soggettivo, in quanto la costruzione di un grafo è un procedimento che permette maggiori libertà individuali da parte dell'utente, rispetto alla costruzione di un albero, nel quale l'inserimento degli oggetti grafici avviene in maniera ordinata e completamente assistita dal sistema.

Quando costruiamo il grafo, il sistema non ci vincola minimamente rispetto alla disposizione dei nodi nello spazio. Risulta evidente che un grafo può venir costruito in maniera leggibile, distribuendo bene i nodi ed evitando intersezioni tra le connessioni, ma anche in maniera disordinata, senza una logica all'apparenza ben

²⁷ Bisogna dire, a questo proposito, che parte della difficoltà può essere legata anche alla stilizzazione dell'albero scelta.

precisa, che limita considerevolmente la facilità di lettura. Al fine di migliorarne la lettura, a nostro avviso è utile la possibilità, offerta da ORANGE, di rinominare i vari nodi presenti e la possibilità di inserire un nodo di informazione, che mostri i dettagli sullo schema costruito.

Concludendo questa parte, dedicata alla rappresentazione ed alla costruzione della query, possiamo dire che il tipo di metafora scelta, oltre ad influire sull'aspetto grafico dell'applicazione, influisce particolarmente sulla costruzione della query, e di conseguenza sull'usabilità e le potenzialità del software stesso. Sotto certi punti di vista, la struttura ad albero risulta sicuramente quella che offre il miglior connubio tra espressività e facilità d'uso, mentre sotto altri, tale struttura, se intesa in maniera troppo rigida, come in TANAGRA o in GhostMiner Developer, può limitare eccessivamente sia il sistema, pregiudicando le possibili problematiche effettivamente affrontabili, sia l'utente, che può trovarsi ad essere "schiacciato" in schemi troppo rigidi e poco espressivi.

4.4 ESECUZIONE E VALIDAZIONE

Un altro punto chiave, che andremo ad analizzare, riguarda le metodologie fornite dai tool per mostrare eventuali errori presenti nella query, e per l'esecuzione di quest'ultima. Tutti i tool, come abbiamo osservato in precedenza, presentano mezzi per evitare la costruzione di query incoerenti, direttamente durante la costruzione, come nel caso di WEKA Knowledge Flow o ORANGE, impedendo così il collegamento di operatori incompatibili. Dato che tali regole da sole non sono in grado di garantire la correttezza della query, vi è la necessità di controlli più approfonditi, come ad esempio su eventuali parametri mancanti e su catene di operatori incompleti. Vediamo quindi ora le caratteristiche che i vari software forniscono all'utente per la validazione completa della query.

Partendo da YALE, osserviamo che quest'ultimo permette di validare un esperimento senza necessariamente eseguirlo, e tale validazione riguarda principalmente aspetti sintattici, come il controllo sull'avvenuta immissione dei parametri obbligatori, o sulla corretta sequenza di concatenazione degli operatori.

Ad esempio, il sistema richiede che l'operatore di costruzione di un albero di decisione abbia come operatore a monte quello di lettura di un data set. Se ciò non si verifica, il sistema non permette di eseguire la query e presenta l'errore in una finestra di output posta nella parte bassa della *treeview*. Tale validazione viene eseguita su esplicita richiesta dell'utente, tramite l'apposito bottone posto nella parte alta dell'applicazione, o attraverso le opzioni del menù *Experiment*.

```
23-mag-2005 11.19.28: Checking properties...  
23-mag-2005 11.19.28: Properties are ok  
23-mag-2005 11.19.28: Checking experimental setup...  
23-mag-2005 11.19.28: Inner operators are ok  
23-mag-2005 11.19.28: Checking i/o classes...  
23-mag-2005 11.19.28: i/o classes are ok  
23-mag-2005 11.19.28: Experiment ok.
```

Figura 4.9 Validazione di una query in YALE.

Se tutto è corretto, come mostrato in Figura 4.9, il sistema permette l'esecuzione dell'esperimento sempre attraverso le modalità sopradescritte²⁸. Poiché i controlli riguardano aspetti soprattutto sintattici, e non prevedono alcun controllo sui tipi, eventuali eccezioni sollevate a run-time vengono presentate attraverso la comparsa di una finestra, che mostra la causa e il nodo che ha generato l'eccezione.

WEKA, seppur in maniera diversa e con un grado di complessità maggiore rispetto a YALE, offre anch'esso informazioni su eventuali incompatibilità presenti nel processo appena costruito. Attraverso una finestra di *log*, mostrata in Figura 4.10, visualizzabile solo su esplicita richiesta dell'utente, il sistema registra eventuali errori commessi nella stesura ed immissione dei parametri, presentando in maniera chiara e completa il tipo di errore commesso ed il punto in cui è localizzato. Purtroppo, tale finestra non viene presentata in maniera automatica dal sistema ogniqualevolta si cerca di eseguire un flusso non corretto, ma è necessario accedervi, quando il sistema non presenta alcun output o genera un'eccezione. Risulta poco intuitiva e scomoda anche l'esecuzione della query stessa, infatti, il software prevede di selezionare la voce *Start loading* da tutti i nodi *DataSources* presenti nel

²⁸ Il passo di validazione viene sempre effettuato, anche se non esplicitamente richiesto, quando si esegue per la prima volta l'esperimento.

nostro flusso. Tale scelta è poco funzionale, basti pensare al caso in cui nel nostro flusso siano presenti diversi nodi di tale classe.

```
11:25:06: Weka Knowledge Flow was written by Mark Hall
11:25:06: Weka Knowledge Flow
11:25:06: (c) 2002-2004 Mark Hall
11:25:06: web: http://www.cs.waikato.ac.nz/~ml/
11:25:06: lunedì, 23 maggio 2005
11:25:48: ERROR : weka.classifiers.meta.AdditiveRegression has not been batch trained;
              can't process instance events.
11:25:48: Warning : weka.classifiers.meta.AdditiveRegression is not an updateable classifier.
              This classifier will only be evaluated on incoming instance events and not trained on them.
```

Figura 4.10 Esempio di finestra di Log di WEKA.

GhostMiner Developer richiede l'esecuzione dell'esperimento ogniqualvolta l'utente inserisce un nuovo operatore. Tale richiesta avviene attraverso un dialog box che lascia all'operatore la libertà di eseguire o meno la query in quel momento. Per quel che riguarda la rilevazione di errori, bisogna dire che GhostMiner Developer prevede una costruzione del progetto totalmente guidata dal sistema, sia nell'inserimento degli operatori, sia nell'immissione dei parametri, quindi il sistema riesce a garantire una costruzione priva di errori.

L'esecuzione del progetto in TANAGRA è delegata all'esplicita richiesta dell'utente, il quale può utilizzare la voce *Execute* presente nel menù *Diagram*, oppure semplicemente fare doppio click sull'operatore di cui interessa l'output.

Riguardo alla ricerca degli errori eventualmente commessi nella stesura della query, TANAGRA non offre praticamente alcun supporto, informa l'utente della presenza dell'errore, ma le informazioni fornite per la sua localizzazione e risoluzione sono alquanto scarse.

Un discorso a parte merita ORANGE. Quest'ultimo esegue la query, man mano che la si costruisce e si forniscono al sistema gli input necessari, ma purtroppo non offre all'utente alcun mezzo esplicito per gestire tale esecuzione, una scelta, quest'ultima dettata dalla necessità di rilevare eventuali errori riscontrabili solo a run-time.

ORANGE infatti si propone essenzialmente come uno strumento di *visual programming* puro, piuttosto che come un *tool* di DM. Il suo scopo principale è la costruzione di un'applicazione *stand alone* a partire dallo schema rappresentante il

flusso di dati appena costruito. Da questo punto di vista, ORANGE è l'unico software tra quelli analizzati che permette di fare ciò. Purtroppo in caso di errori, il sistema risulta poco intuitivo nel guidare l'utente alla loro localizzazione.

La politica di esecuzione appare quindi poco governabile; infatti, un utente che costruisce il proprio flusso in maniera ordinata, partendo dai dati in input, e collegando di volta in volta i vari operatori, si trova ogni volta ad inserire un nuovo operatore ed a dover attendere l'esecuzione di quest'ultimo. Tale esecuzione, nel caso di file di grosse dimensioni, può richiedere parecchio tempo, periodo nel quale il sistema non permette l'immissione di ulteriori operatori. Sfortunatamente tale politica è piuttosto invasiva, e sarebbe auspicabile, dato il fine che il software si pone, la presenza di un ottimo supporto per il *debugging* o per un'esecuzione guidata dell'applicazione. Riguardo all'applicazione ottenuta, possiamo affermare che appare robusta e presenta un ottimo livello di interattività.

Sebbene purtroppo nella versione attuale presenti alcune limitazioni, che ne circoscrivono ancora il reale utilizzo in ambiti accademici, possiamo affermare che ORANGE rappresenta sicuramente uno sguardo verso il futuro nella costruzione di applicazioni di DM.

Evidenziamo, per correttezza, che gli altri tool analizzati in realtà non si pongono come obiettivo quello di creare un vero e proprio strumento di *visual programming* per costruire applicazioni di DM, ma un'interfaccia grafica che permetta di facilitare l'utilizzo di metodologie di DM.

4.5 VISUALIZZAZIONE DEI DATI E DEI RISULTATI

Altre caratteristiche fondamentali, che un tool di DM deve fornire, riguardano la rappresentazione dei dati utilizzati in input e la rappresentazione della conoscenza estratta.

Per i primi è auspicabile che il sistema fornisca rappresentazioni sintetiche, attraverso statistiche e grafici delle ennuple presenti nel data set, mentre per i

secondi ci si aspettano forme di visualizzazione che rendano più chiara e comprensibile l'informazione che è stata ottenuta²⁹.

Nei tool da noi analizzati, sotto questo punto di vista spicca nettamente GhostMiner Developer. Quest'ultimo offre innumerevoli opzioni di visualizzazione di qualità superiore rispetto agli altri, sia da un punto di vista visivo, che per quantità di informazioni fornite, e per il numero di opzioni presenti.

Per quanto riguarda gli altri software, ad eccezione di TANAGRA, presentano sostanzialmente tutti un buon livello di visualizzazione dei modelli estratti (offrendo anche rappresentazioni grafiche) e dei dati utilizzati in input. In taluni casi, come abbiamo rilevato in precedenza, è l'utente stesso a dover inserire opportuni operatori per avere informazioni aggiuntive. TANAGRA, invece, sotto questo punto di vista, risulta meno curato, poiché non possiede rilevanti strumenti di visualizzazione né per i dati in input, né per la conoscenza estratta, mentre offre principalmente visualizzazioni di tipo testuale. Presenta tuttavia la possibilità di esportare tutti gli elementi presenti in una query in HTML, rendendo di fatto visualizzabile la conoscenza estratta anche al di fuori del tool stesso, attraverso un semplice browser web.

Quest'ultima considerazione ci offre lo spunto per parlare dell'utilizzo dei modelli estratti. Oltre alla semplice visualizzazione ed all'utilizzo all'interno del software che li ha generati, vale infatti la pena soffermarci anche sulla possibilità di utilizzare tale conoscenza indipendentemente dal software generante.

Sarebbe infatti auspicabile che i modelli estratti fossero indipendenti dallo specifico tool, e quindi portabili. Purtroppo, anche in questo caso, la già citata mancanza di uno standard preciso ha portato gli sviluppatori ad utilizzare rappresentazioni proprietarie dei modelli estratti.

Per quanto riguarda la portabilità di modelli, negli anni è stato definito uno standard basato su XML, chiamato PMML (par. 2.3), ma nessuno dei tool da noi analizzati prevede ancora la possibilità di esportare, o di utilizzare i modelli in tale formato. Per alcuni strumenti open source sono disponibili delle versioni

²⁹ Non tutti i modelli estraibili, per la loro stessa natura, possono avere facile leggibilità.

modificate³⁰, che introducono tale caratteristica. Questo è il caso di WEKA 3-3-5 PMML, che estende WEKA 3-3-5, per permettere l'esportazione e la visualizzazione di modelli in formato PMML per alcuni operatori. Per maggiori informazioni su tale versione fare riferimento al sito [WPV].

GhostMiner Developer, a differenza di WEKA, non supporta PMML, ma permette comunque di salvare i modelli estratti in file XML.

Tutti gli strumenti analizzati presentano una buona interattività, sia a livello di costruzione della query, all'interno della quale è possibile di volta in volta cambiare i parametri di un *task*, o sostituire il *task* stesso, sia per quel che riguarda la visualizzazione dei risultati, in cui nella maggior parte dei casi, sono presenti numerose opzioni.

4.6 ESTENDIBILITÀ DEI SISTEMI

Pur non tenendo conto del numero di algoritmi forniti con il sistema, e delle aree di DM coperte, possiamo affermare che un punto chiave che può far preferire un sistema ad un altro riguarda l'espandibilità. Un utente potrebbe aver bisogno di utilizzare algoritmi propri o più specifici rispetto a quelli già presenti, estendendo di fatto il sistema con nuove funzionalità non previste in origine.

Tralasciando GhostMiner, che essendo sviluppato a fini commerciali e protetto da copyright, non prevede alcuna possibilità di espansione, né fornisce la possibilità di accedere, almeno in parte, al codice sorgente, possiamo affermare che i tool basati su JAVA, ovvero WEKA Knowledge Flow e YALE, hanno una maggiore facilità di espansione rispetto agli altri. Purtroppo l'introduzione dell'interfaccia grafica complica ulteriormente le cose.

WEKA³¹ offre all'utente la possibilità di espandere a proprio piacimento il sistema di base, mentre non presenta alcuna possibilità di integrare nella Knowledge Flow GUI gli operatori aggiunti, creando opportuni bottoni e menù relativi all'immissione dei parametri.

³⁰ Nella maggior parte dei casi, tali strumenti non vengono forniti direttamente dal team di sviluppo "ufficiale".

³¹ Non specificatamente il Knowledge Flow.

Per quel che riguarda YALE, l'espandibilità del sistema di base, quindi privo dell'interfaccia grafica, avviene in maniera analoga a WEKA. In questo caso, il sistema prevede anche dei meccanismi per integrare elementi grafici riguardanti i nuovi algoritmi nella GUI. Quest'ultima caratteristica è possibile grazie alla filosofia di YALE, che utilizza file XML, per configurare le varie componenti dell'interfaccia³². La completa integrazione in YALE prevede quindi anche la redazione da parte dell'utente di un apposito file XML, grazie al quale il sistema non solo crea gli opportuni oggetti grafici, ma riesce ad effettuare i dovuti controlli sui tipi e sui parametri eventualmente obbligatori e non ancora inseriti.

Gli altri due tool presi in esame non offrono meccanismi agevolati di espansione, e basano questa caratteristica sul fatto che il sistema è open source. Nel caso di ORANGE, sfruttando il meccanismo dell'ereditarietà, è possibile ridefinire i metodi già presenti, o estendere la gerarchia delle classi con altre, ma anche in questo caso l'espansione riguarda solo il sistema di base e non comprende la GUI.

Purtroppo l'introduzione della GUI, come abbiamo visto, compromette in maniera considerevole l'espansione di questi sistemi. A questo proposito, possiamo pertanto concludere che un metodo intuitivo e semplice per espandere il sistema, agendo direttamente attraverso l'interfaccia grafica, è tuttora impensabile.

Nella maggior parte dei casi, l'interfaccia grafica rappresenta semplicemente un livello "appoggiato" ad un sistema preesistente e funzionante indipendentemente dalla presenza di quest'ultima. Questa filosofia porta a diversificare i modelli di espansione, e mentre da una parte troviamo quelli dedicati all'inserimento nel sistema base degli algoritmi veri e propri, dall'altra abbiamo quelli che si occupano dell'inserimento nell'astrazione grafica di tutte quelle componenti che permettono di utilizzare le nuove funzionalità appena inserite.

4.7 SUPPORTO ALL'UTENTE

Prima di giungere alle conclusioni, occorre soffermarci brevemente anche sulla documentazione offerta dai vari tool. Da questo punto di vista, il software che

³² Ricordiamo che la query stessa è un file XML.

presenta maggiori svantaggi rispetto agli altri è sicuramente WEKA. Per la sua Knowledge Flow GUI non è ancora previsto alcun supporto rivolto agli utenti, né sotto forma di manuale, né di *tutorial* introduttivi.

Gli altri tool forniscono invece documentazione almeno sufficiente, e soprattutto opportuni *tutorial* che permettono, attraverso semplici esempi, di acquisire i primi rudimenti delle varie caratteristiche disponibili. Da evidenziare in questo caso la politica adottata da YALE, che è l'unico software ad integrare un *tutorial* direttamente nell'interfaccia.

Come abbiamo potuto constatare, il mondo dei tool di DM è piuttosto variegato e non vi è ancora uno standard che si stia imponendo in maniera consistente sugli altri. Questo non ci permette di evidenziare in modo così netto e definitivo delle qualità specifiche ben marcate su cui poggiare le nostre considerazioni. Possiamo comunque delineare delle caratteristiche che ben si adattano al nostro software, o alcune particolarità, che reputiamo importanti, presenti in alcuni software.

Il nostro compito finale, in sintesi, riguarda la ricerca di linee guida per la definizione di queste caratteristiche, che andranno a costituire il nucleo chiave su cui sarà basata l'intera interfaccia per KDDML.

4.8 REQUISITI

Concludiamo questo capitolo avviandoci a delineare delle scelte sugli aspetti che secondo noi dovrebbero essere soddisfatti da un tool di DM, inserendo talvolta anche considerazioni personali o suggerimenti per migliorare la funzionalità dei componenti presi in esame. Ci riserviamo poi nel prossimo capitolo di valutare se e in che misura questi aspetti si adattano ad essere inseriti ed utilizzati nella specifica realtà di KDDML.

Forniamo quindi una serie di punti chiave, riguardanti i principali fattori sui quali porre la nostra attenzione nella costruzione di un'interfaccia grafica. Tali punti verranno delineati attraverso **requisiti**, rappresentati da una lettera “R” e da **suggerimenti**, denotati da una “S”.

Tali regole non si pongono l'obiettivo di esser completamente esaustive, o fornire soluzioni specifiche. Ad esempio, rimane aperta la questione se fornire o meno un *wizard* per semplificare ulteriormente il processo di KDD, così come la modalità di presentazione di tale caratteristica, oppure se la query deve essere redatta in modo leggibile, anche al di fuori del sistema stesso.

Queste regole rappresentano semplicemente dei punti chiave, dei requisiti su cui costruire e basare i nostri studi futuri.

4.8.1 ACCESSO AI DATI

Come visto nel paragrafo 4.2, è importante, indipendentemente dal livello GUI, che un tool fornisca la possibilità di interfacciarsi a database, in quanto non è detto che i dati siano sempre presenti localmente. Inoltre i dati che ci interessano possono essere distribuiti tra diversi database.

Un altro punto chiave riguarda la possibilità di utilizzare più data set contemporaneamente per estrarre più modelli da dati diversi, attraverso una stessa query.

R. 1.0: Devono essere forniti strumenti per l'accesso a database.

R. 1.1: Deve essere possibile utilizzare più data set in una stessa query, per permettere l'estrazione contemporanea di più modelli.

4.8.2 LA METAFORA GRAFICA

Come visto nel paragrafo 4.3, la metafora grafica rappresenta il nucleo su cui costruire l'intera interfaccia, e dovrebbe garantire ottima espressività e facilità d'uso. A tal fine, riteniamo che il grafo presenti un livello di espressività e potenzialità maggiore rispetto all'albero, ma deve essere integrato con meccanismi che limitano la parte soggettiva nella costruzione del flusso.

Una possibile soluzione può essere rappresentata, a questo proposito, dalla divisione dell'area di costruzione della query in fasi. Essa, ad esempio, potrebbe

essere suddivisa in categorie dipendenti dal task di DM da effettuare, come input, preprocessing, learning, validation, e visualization, con la possibilità di inserire in ogni area solo gli operatori che fanno parte di una specifica classe. Ciò metterebbe in evidenza anche il processo a fasi tipico del KDD.

Numerose problematiche andrebbero comunque risolte, poiché non tutti gli utenti potrebbero aver bisogno delle medesime fasi. Inoltre l'output di una fase può rappresentare l'input di una fase precedente, dato che il processo KDD è iterativo (par. 1.3), vi è quindi la necessità di gestire anche cicli tra le fasi.

R. 2.0: Utilizzo del grafo come metafora grafica per rappresentare la query.

R. 2.1: Suddivisione della costruzione del flusso, seguendo le varie fasi del processo KDD.

R. 2.2: Controllo degli operatori inseribili nelle varie fasi del processo.

S. 2.0: Possibile presenza di un wizard, che aiuti nella stesura della query.

4.8.3 SEMPLICE E COMPLETA ESPANSIONE DEL SISTEMA

Riteniamo l'espandibilità del sistema una delle caratteristiche fondamentali di qualsiasi software di DM. Purtroppo, come osservato in precedenza (par. 4.6), l'introduzione di un'interfaccia grafica può vanificare molti degli sforzi rivolti alla semplificazione di tale meccanismo. Un requisito che reputiamo importante è la necessità di non dover andare a manipolare l'interfaccia grafica ogniqualvolta si estenda il sistema. L'estensione può riguardare l'introduzione di nuovi algoritmi, l'utilizzo di nuovi modelli e nuovi formati di sorgenti/destinazioni.

Una caratteristica su cui porre particolare attenzione riguarda la generazione dell'*help on line* dei vari operatori. Quest'ultimo deve avvenire in maniera automatica, ogniqualvolta si inserisca un nuovo operatore, senza alcuna modifica alla GUI.

L'obiettivo è quindi quello di costruire un'interfaccia che non sia realizzata staticamente, ma che svincolata dal nucleo funzionale del software, riesca ad aggiornarsi ogni volta che avviene un cambiamento nel sistema.

Potrebbe essere utile anche lo studio di qualche meccanismo per permettere l'espansione del sistema attraverso una procedura guidata a livello GUI. Grazie a questa, l'utente viene guidato dal sistema nella corretta immissione di tutte le parti richieste per l'integrazione di nuove funzionalità. Questa caratteristica è indubbiamente legata all'architettura specifica di un determinato software e richiede metodi specifici, non generalizzabili, per risolvere le varie problematiche ad essa associate.

R. 3.0: Il kernel del sistema deve essere facilmente espandibile.

R. 3.1: Non deve essere necessario modificare il livello GUI ogni volta che si estende il sistema.

S. 3.0: Possibile procedura guidata, a livello GUI, per l'estensione del sistema.

4.8.4 ESECUZIONE E META-ESECUZIONE

Non è pensabile dover necessariamente eseguire la query per scoprire eventuali errori presenti, in quanto tali errori potrebbero presentarsi anche dopo lunghe attese, dovute a fasi corrette di preprocessing su enormi quantità di dati. Come visto nel paragrafo 4.4, è desiderabile, quindi, che il sistema fornisca strumenti per verificare la correttezza della query senza eseguirla, cercando di mettere in luce tutti i possibili errori rilevabili attraverso un'analisi statica.

Sarebbe inoltre auspicabile riuscire a creare un meccanismo che incrementi le potenzialità offerte dalla sola analisi statica. A tal fine, si possono creare strumenti di meta-esecuzione.

La meta-esecuzione rappresenta la possibilità di concretizzare verifiche avanzate sulla correttezza della query. Attraverso questo meccanismo, è possibile attuare una

serie di controlli ottenibili tipicamente a run-time, senza dover mandare in esecuzione il processo, o per meglio dire senza manipolare i dati fisici.

Tali controlli devono quindi sfruttare una serie di “meta-informazioni” fornite dal sistema, sullo schema logico dei dati fisici (meta-dati) e dei modelli (meta-modelli), riguardanti la struttura dei dati.

Per chiarire meglio questi concetti, illustriamo dei semplici esempi di utilizzo della meta-esecuzione. Innanzitutto, partendo dai meta-dati, in Figura 4.11 è rappresentato un esempio di schema logico previsto nel file di tipo *arff*.

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

Figura 4.11 Esempio di meta-dati tipici dei file *arff* utilizzati da WEKA.

Attraverso questo schema è possibile avere delle indicazioni sul numero e sul tipo degli attributi che appaiono nel seguito del file, e che rappresentano i dati fisici. Un esempio tipico di errore rilevabile attraverso la meta-esecuzione, riguarda l'utilizzo dell'algoritmo ID3 (par. 1.4.2). Quest'ultimo lavora solo con attributi nominali, e quindi, se utilizziamo come training set un insieme che prevede attributi di tipo numerico, il sistema solleva un'eccezione a run-time, quando si trova ad eseguire tale algoritmo. Risulta evidente che, accedendo semplicemente ai meta-dati del training set, è possibile rilevare tale errore e presentarlo all'utente.

Un altro possibile esempio può riguardare l'utilizzo di tecniche per incrementare l'accuratezza di un classificatore, come il *bagging* (par. 1.4.2), che prevede l'utilizzo di k classificatori, che noi potremmo fornire al sistema, attraverso k modelli già precedentemente estratti. Questi modelli, seppur estratti da diversi dati fisici, devono condividere lo stesso schema logico, ed essere estratti da training set che hanno gli stessi meta-dati. In caso contrario, alcuni dei classificatori utilizzati potrebbero fare riferimento ad attributi non presenti nel file da classificare, invalidando l'intero processo.

Un controllo su tale caratteristica può essere effettuato, se il sistema prevede anche la presenza di schemi logici per i modelli. La proprietà da evidenziare ulteriormente è che, in entrambi i casi, noi non andiamo ad eseguire la query e quindi a manipolare i dati fisici, cosa che è *time-consuming*, ma i controlli avvengono accedendo solo alle “meta-informazioni”, che devono comunque essere previste dal sistema.

Come è lecito attendersi, non tutti gli errori sono rilevabili attraverso la meta-esecuzione, basti pensare al caso in cui i meta-dati non corrispondono ai dati fisici, perchè intenzionalmente modificati. Altre procedure devono quindi essere garantite a tempo di esecuzione, per permettere il corretto svolgimento della query KDD. Inoltre, i controlli a run-time riguardano in generale tutte quelle situazioni in cui le informazioni note staticamente, non sono sufficienti a garantire la correttezza della query.

Da quanto detto sopra, possiamo avere una stratificazione dei controlli attuabili sulla query, che in parte dipendono dalle informazioni che il sistema ci mette a disposizione:

1. **Analisi statica:** riguarda essenzialmente i controlli di correttezza sintattica e dei tipi utilizzati dagli operatori presenti nella query.
2. **Meta-esecuzione:** permette di effettuare controlli più approfonditi, grazie allo schema logico di dati e modelli.
3. **Run-time:** riguarda tutti i controlli sulla query possibili solo a tempo di esecuzione, poiché staticamente non abbiamo informazioni sufficienti per attuarli.

Dal punto di vista grafico, il sistema deve inoltre fornire una finestra di output in cui presentare eventuali errori commessi, e guidare l'utente nella loro correzione. Inoltre, i comandi di meta-esecuzione e di esecuzione devono essere ben visibili e di semplice accesso, e l'esecuzione deve avvenire solo su esplicita richiesta dell'utente.

R. 4.0: Evitare l'esecuzione di query non valide.

R. 4.1: Utilizzare meta-dati e meta-modelli per attuare la meta-esecuzione.

R. 4.2: Fornire la GUI di semplici strumenti per il reperimento, la guida nella correzione degli errori eventualmente commessi e per attuare la meta-esecuzione e l'esecuzione vera e propria.

S. 4.0: Aggiunta di strumenti per l'esecuzione parziale della query.

4.8.5 IMMISSIONE DEI PARAMETRI

Anche questa è un'altra caratteristica, discussa nel paragrafo 4.3.3, che può influire sull'usabilità e sulla potenzialità che un'interfaccia grafica può offrire. Questa categoria può essere suddivisa in due sezioni, che rappresentano due aspetti fondamentali nella concezione di un buon sistema di immissione dei parametri:

- a. Metodologia di immissione:** deve essere il più intuitiva possibile, deve fornire un alto tasso di interattività e garantire, a richiesta dell'utente, una breve sintesi del significato dei vari parametri presenti. Inoltre, l'utente deve avere la possibilità di accedervi facilmente, al fine di modificare i vari parametri precedentemente introdotti. Una soluzione possibile potrebbe essere quella di visualizzare in automatico il menù di immissione dei parametri, ogni volta che si inserisce un operatore che li prevede, e riaccedervi semplicemente cliccando sull'operatore stesso.
- b. Controlli sull'immissione:** devono essere studiati approfonditamente ed essere indipendenti dalla specifica metodologia utilizzata. Tali controlli riguardano, ad esempio, l'immissione di parametri obbligatori, il tipo corretto ed il giusto *range* dei dati forniti dall'utente. Queste caratteristiche sono fondamentali nel guidare l'utente nella corretta stesura della query, e nel fornirgli *dialog box* di immissione dati molto specializzati, ad esempio inserendo *combo box* con solo

i valori ammissibili per quel determinato parametro. Questa proprietà è comunque legata, oltre che alla costruzione corretta della query, grazie ad una strutturazione più guidata e ad un alto tasso di interattività, anche a favorire il passo di validazione, esonerandolo dai controlli sui parametri. Tale discorso si lega irrimediabilmente anche all'espandibilità del sistema. Infatti, se supponiamo che il livello dell'interfaccia grafica non debba mai essere modificato, dobbiamo garantire, da parte della struttura sottostante, una serie di informazioni che permettano all'interfaccia di capire quali sono i parametri obbligatori, quelli opzionali con i relativi valori di *default*, e quali sono ad esempio i *range* di valori da visualizzare.

R. 5.0: Deve essere garantita massima semplicità per immissione, visualizzazione e reperimento dei vari parametri degli operatori.

R. 5.1: Per ogni parametro inseribile deve essere fornito un help on line con la spiegazione del significato di tale parametro all'interno del contesto di applicazione.

S. 5.0: Il nucleo del sistema deve fornire a livello GUI informazioni sufficienti per attuare controlli e visualizzazioni specifiche sui vari parametri ammissibili³³.

4.8.6 VISUALIZZAZIONE INPUT/OUTPUT

Il sistema deve integrare nella sua interfaccia grafica una serie completa di visualizzatori, attraverso i quali fornire all'utente informazioni sia sui dati in input, sia sugli output richiesti. Deve garantire una buona rappresentazione grafica sia per i dati, che per i modelli e per le performance, inoltre per ognuno di essi devono essere possibili diverse prospettive di visualizzazione.

Scegliendo la metafora a grafo, tali visualizzatori devono essere inseriti esplicitamente dall'utente, complicando leggermente la costruzione della query, come osservato nel paragrafo 4.3.2. Reputiamo quindi che, seguendo un approccio intermedio, possiamo fornire ad ogni nodo di output un visualizzatore per garantire

³³ Questo suggerimento è legato anche alle caratteristiche dell'espandibilità del sistema.

all'utente almeno le informazioni base, senza esplicita immissione di opportuni operatori.

R. 6.0: La GUI deve essere integrata con vari strumenti di visualizzazione per rappresentare graficamente l'informazione manipolata.

R. 6.1: Un numero minimo di visualizzatori deve essere introdotto implicitamente dal sistema.

4.8.7 MODELLI ESTRATTI E UTILIZZO DI PMML

È auspicabile che il software permetta il riutilizzo dell'informazione estratta, senza doverla elaborare ex-novo ogni volta (par. 4.2). Deve essere possibile salvare i modelli e importarli. Inoltre il sistema deve supportare in maniera nativa la manipolazione dei modelli in PMML, come visto nel paragrafo 4.5. Devono essere forniti metodi per importare, esportare e per visualizzare modelli in PMML. Questo non riguarda quindi la sola interfaccia grafica, ma si inserisce in un contesto ben più ampio che include il sistema nella sua completezza.

R. 7.0: Deve essere possibile riutilizzare la conoscenza estratta.

R. 7.1: Completo supporto al PMML.

4.8.8 POSSIBILITÀ DI SALVATAGGIO DEL KNOWLEDGE FLOW

Il software, oltre a fornire il salvataggio dei modelli estratti, deve permettere di salvare, caricare e successivamente modificare l'intero processo. Deve essere possibile salvare l'intero processo, ed inoltre è necessario valutare la possibilità di utilizzare un formato leggibile anche al di fuori della sola interfaccia, come XML, oppure utilizzare un formato proprietario. Può risultare utile salvare l'intero processo in HTML, al fine di renderlo *browsable* per effettuare analisi, indipendentemente dal software con cui tale processo è stato creato.

R. 8.0: Garantire la possibilità di salvare/caricare la query che rappresenta il processo KDD.

S. 8.0: Fornire la possibilità di salvare sia l'intero processo che i risultati ottenuti in HTML.

S. 8.1: Possibile visualizzazione/salvataggio dei risultati intermedi.

4.8.9 DOCUMENTAZIONE FORNITA

Come analizzato nel paragrafo 4.7, è importante offrire all'utente una documentazione completa, riguardante le caratteristiche e l'uso dell'interfaccia. Inoltre, sarebbe auspicabile aiutare l'utente nel prendere confidenza con il sistema attraverso una serie di *tutorial*, che lo guidino passo dopo passo nella scoperta delle varie potenzialità offerte, e nella costruzione di processi tipici KDD.

R. 9.0: Fornire il sistema ed in particolare la sua GUI di adeguata e completa documentazione.

S. 9.0: Introdurre una serie di tutorial, per facilitare l'utente nel prendere confidenza con il sistema.

4.9 CONSIDERAZIONI FINALI

Concludiamo questo capitolo, riguardante la comparazione tra i vari tool analizzati nel capitolo 3, presentando una tabella riassuntiva (Tabella 4.1), che mette in evidenza come i requisiti delineati nel paragrafo 4.8 siano o meno soddisfatti da tali software. È da evidenziare inoltre che alcuni requisiti non vengono ancora supportati da nessuno dei software analizzati.

Il prossimo capitolo si concentrerà nell'analizzare se e come questi requisiti e suggerimenti si adattano ad essere attuati nella costruzione di un'interfaccia grafica per il sistema KDDML. Inoltre, dato che non tutti i requisiti riguardano la sola interfaccia grafica, possono rappresentare un punto di partenza per garantire al

sistema caratteristiche più avanzate, rispetto a quelle di cui è già provvisto, in modo da diversificarlo maggiormente dai sistemi open source suoi concorrenti.

	Requisito	WEKA KF	YALE	ORANGE	G.M.D.	TANAGRA
Accesso ai dati						
Database	R. 1.0	+	+	-	+	-
Multi-source	R. 1.1	+	+	+	-	-
Metafora grafica						
Albero Vs. Grafo	R. 2.0	+	(Albero)	+	(Albero)	(Albero)
Div. in Fasi del processo KDD	R. 2.1	-	-	-	-	-
Controllo op. inseriti	R. 2.2	+/-	+/-	+/-	+/-	+/-
Espansione del sistema						
Kernel Espandibile	R. 3.0	+	+	+/-	-	+/-
Level Gui non mod.	R. 3.1	-	+/-	-	-	-
Esecuzione e meta-esecuzione						
Query non valide	R. 4.0	+/-	+/-	+/-	+	-
Meta-esec.	R. 4.1	-	-	-	-	-
Presentazione errori	R. 4.2	+/-	+	-	+	+/-
Immissione dei parametri						
Semplicità di immissione	R. 5.0	+/-	+	+/-	+	+/-
Help on line	R. 5.1	+	+	+/-	+	+/-
Visualizzazione input/output						
Strumenti di visualizz.	R. 6.0	+	+	+	+	+/-
Visualizz. di default	R. 6.1	-	+	-	+	+
Modelli estratti & utilizzo PMML						
Riutilizzo conoscenza	R. 7.0	-*	+	-	-*	-
PMML	R. 7.1	-	-	-	-	-
Possibilità di salvataggio						
Salvataggio Processo	R. 8.0	+	+	+	+	+
Documentazione fornita						
Documentazione	R. 9.0	-	+	+/-	+	+

Tabella 4.1 Tabella riassuntiva dei requisiti.

Legenda:

+ : requisito pienamente soddisfatto.

- : requisito non soddisfatto.

+/-: requisito in parte soddisfatto.

* In questi casi il modello salvato può essere riutilizzato attraverso WEKA Explorer, nel caso di WEKA Knowledge Flow GUI e GhostMiner Analyzer, nel caso di GhostMiner Developer.

Capitolo 5

STUDIO DEI REQUISITI DEL LINGUAGGIO VISUALE PER KDDML

5.1 INTRODUZIONE

In questo capitolo finale, cerchiamo di riassumere lo studio che ha portato alla stesura dei requisiti elencati alla fine del precedente, andando ad analizzare se tali requisiti possono adattarsi ad una realtà ben precisa e già piuttosto delineata come quella del sistema KDDML, descritto nel corso del capitolo 2.

La nostra trattazione verterà principalmente su fattori riguardanti l'interfaccia grafica, e su caratteristiche ritenute importanti che attualmente il sistema non possiede.

I requisiti stilati possono essere infatti divisi in due grandi famiglie, in quanto riguardano sia il sistema nella sua interezza, sia più specificamente il livello grafico.

Il seguito del capitolo quindi tratterà uno ad uno i vari requisiti, cercando di valutare in primo luogo se questi vengono già soddisfatti dal sistema attuale, ed in caso positivo, se vi possono essere difficoltà nella loro introduzione all'interno dell'interfaccia grafica.

Per quanto riguarda gli altri, cerchiamo di capire se e come è possibile soddisfarli, fornendo di volta in volta possibili soluzioni, o accennando a problemi di cui tenere conto in fase di implementazione. Utilizzeremo inoltre alcuni dei suggerimenti,

presentati nel capitolo 4, per ampliare le potenzialità del sistema, o per proporre più soluzioni alternative.

La trattazione manterrà, comunque, sempre un livello piuttosto generico di caratterizzazione, quindi raramente farà riferimento all'implementazione fisica del sistema. Tuttavia, tenendo conto della sua attuale architettura, cercherà di capire se e dove modificare gli elementi esistenti, per fornire al sistema un'interfaccia grafica che non sia solo una guida text-oriented alla scrittura di query KDDML, ma che si configuri come un mezzo potente per rappresentare in maniera semplice ed intuitiva la maggior parte delle potenzialità offerte dal sistema KDDML.

5.2 KDDML & ACCESSO AI DATI

“R. 1.0: Devono essere forniti strumenti per l'accesso a database.

R. 1.1: Deve essere possibile utilizzare più data set in una stessa query, per permettere l'estrazione contemporanea di più modelli.”

Il sistema KDDML prevede già appositi operatori per l'interfacciamento a database relazionali. Questi operatori sono:

- **DATABASE_LOADER**: garantisce un accesso trasparente a tabelle relazionali, appartenenti a RDBMS locali o remoti, attraverso query SQL. La sua sintassi prevede che gli siano forniti gli attributi riguardanti la query SQL con cui interrogare il database e i dati sui database a cui connettersi.
- **DATABASE_WRITER**: permette di accedere a tabelle nel *repository*, e le trasforma in tabelle SQL. La corrispondenza tra i data source ed i tipi logici del database non è automatica. Un requisito dell'operatore è che nel database di destinazione deve esistere una tabella vuota SQL, corrispondente alla tabella relazionale target. L'operatore ha inoltre dei precisi formalismi sul formato dei dati per permettere tale operazione.

Dal punto di vista grafico, rimane solo la necessità di integrare tali operatori con i rispettivi parametri nell'interfaccia grafica. Il requisito *R. 1.0* è quindi già pienamente soddisfatto e non presenta grosse difficoltà di implementazione nella GUI.

Anche il requisito *R. 1.1* non riguarda la sola metafora grafica, ma deve essere inserito in un discorso più ampio relativo all'intero sistema. La questione merita quindi di essere approfondita, in quanto se il sistema non permette di utilizzare più data set contemporaneamente, allora la metafora grafica più adatta, sia per motivi essenzialmente di semplicità di costruzione del processo, sia di implementazione, è sicuramente quella ad albero, simile a quella vista per TANAGRA (par. 3.6) e GhostMiner Developer (par. 3.5).

La possibilità di utilizzare più data set contemporaneamente nello stesso processo potrebbe quindi invalidare il requisito (*R. 2.0*), che considera il grafo come la metafora grafica più appropriata per il processo KDD.

A tal fine, osserviamo che il sistema KDDML offre la possibilità di utilizzare più data set contemporaneamente, rendendo quindi possibile l'estrazione di più modelli, sia dagli stessi dati che da dati diversi in una stessa query, soddisfacendo *R. 1.1*, e lasciando aperta la possibilità di soddisfare anche il requisito *R. 2.0*.

Un esempio di quanto appena descritto è rappresentato dal seguente caso, mostrato in Figura 5.1, in cui è presentata una query KDDML che estrae sia un insieme di RdA (attraverso il tag `<RDA_MINER>`), che un albero di classificazione (`<TREE_MINER>`) da data set diversi, `adult.xml` e `iris.xml`, presenti nel *repository*. La query fa uso anche dell'operatore `<SEQ_QUERY>`, che permette di sequenzializzare due o più operatori. Alla fine dell'intero processo troveremo i due modelli nei rispettivi *repository*.

```

<?xml version="1.0" encoding="UTF-8"?>
<KDDML_OBJECT>
  <KDD_QUERY name="doubleModel">
    <SEQ_QUERY>
      <RDA_MINER xml_dest="rules.xml">
        <TABLE_LOADER xml_source="adult.xml"/>
        <ALGORITHM algorithm_name="DCI">
          <PARAM name="min_support" value="0.4"/>
          <PARAM name="min_confidence" value="0.6"/>
          <PARAM name="max_number_of_rules" value="10"/>
        </ALGORITHM>
      </RDA_MINER>
      <TREE_MINER target_attribute="class" xml_dest="tree.xml">
        <TABLE_LOADER xml_source="iris.xml"/>
        <ALGORITHM algorithm_name="YADT">
          <PARAM name="confidence_for_pruning" value="0.4"/>
          <PARAM name="num_instances_for_leaf" value="2.0"/>
        </ALGORITHM>
      </TREE_MINER>
    </SEQ_QUERY>
  </KDD_QUERY>
</KDDML_OBJECT>

```

Figura 5.1 Query KDDML per estrarre due modelli diversi da dati diversi.

Alla base di tutto è la concezione del sistema di trattare i dati, ed i modelli, come oggetti che rappresentano semplicemente i tipi utilizzati da determinati operatori. La natura del linguaggio è funzionale, quindi ogni operatore è visto come una funzione, che da un opportuno dominio restituisce valori di un dato codominio.

Ad esempio, l'operatore `<RDA_MINER>` è una funzione che, dato un oggetto di tipo *table* (rappresentante un insieme di dati) e dato un oggetto di tipo *alg* (la specifica di un algoritmo di estrazione di RdA), restituisce un oggetto di tipo *rda*, quindi un insieme di regole. Più formalmente lo possiamo esprimere nel seguente modo:

$$f_{\langle RDA_MINER \rangle}: table \times alg \rightarrow rda$$

Da evidenziare comunque il fatto che in input a tale operatore, non è obbligatorio che vi sia un operatore che legge un file su disco, ma vi può essere un altro operatore che restituisce semplicemente un oggetto di tipo *table*. Inoltre, sul tipo *alg* vi è il controllo sul fatto che l'algoritmo passato appartenga effettivamente alla categoria delle RdA.

5.3 KDDML & METAFORA GRAFICA

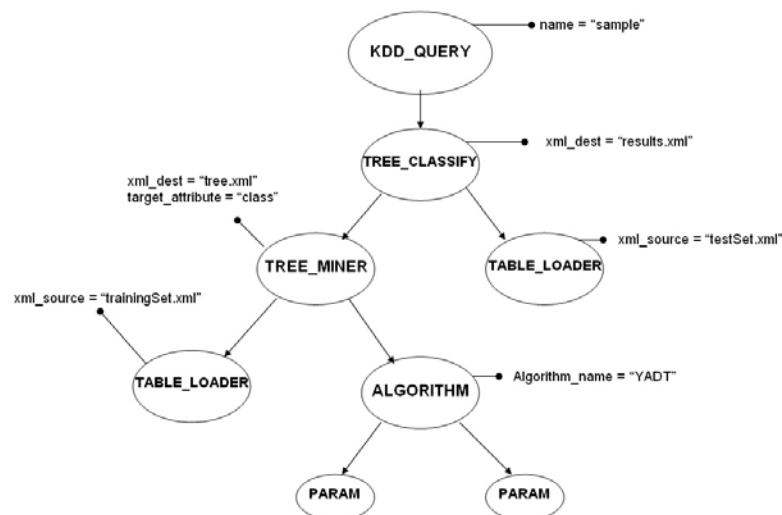
“R. 2.0: Utilizzo del grafo come metafora grafica per rappresentare la query.”

Per riuscire a comprendere meglio se il requisito R. 2.0 può essere soddisfatto senza andare a modificare l'intera architettura del sistema, bisogna innanzitutto richiamare l'attuale struttura di una query KDDML.

La query KDDML, come visto nel capitolo 2, fisicamente è rappresentata attraverso un documento XML. La sua struttura logica è rappresentata da un albero DOM e sembrerebbe quindi improponibile pensarla e rappresentarla come un grafo.

```
<KDD_QUERY name="sample">
  <TREE_CLASSIFY xml_dest="results.xml">
    <TREE_MINER xml_dest="tree.xml" target_attribute="class">
      <TABLE_LOADER xml_source="trainingSet.xml"/>
      <ALGORITHM algorithm_name="YADT">
        <PARAM name="confidence_for_pruning" value="0.4"/>
        <PARAM name="num_instances_for_leaf" value="0.6"/>
      </ALGORITHM>
    </TREE_MINER>
    <TABLE_LOADER xml_source="testSet.xml"/>
  </TREE_CLASSIFY>
</KDD_QUERY>
```

a)



b)

Figura 5.2 Query KDDML e corrispondente albero DOM.

Andando ad analizzare meglio gli elementi costitutivi di una query, è facile convincersi che le foglie dell'albero DOM non sono altro che gli oggetti in input dei vari *task* che stanno al livello immediatamente superiore, e che rappresentano di conseguenza l'input dell'intero processo.

Questo ci può aiutare nel capire che, analizzando prima tutte le foglie e salendo, è possibile ricavare un grafo, in cui abbiamo nodi che ricevono input da più operatori. In pratica, semplificando, possiamo dire che è come invertire le frecce dell'albero DOM. Per chiarire meglio questo concetto, possiamo riprendere l'esempio fornito nel paragrafo 2.4.4, in cui venivano mostrate sia la query KDDML (Figura 5.2.a) rappresentata dal file XML, che il risultante albero DOM (Figura 5.2.b).

Dall'albero DOM è possibile generare un flusso di dati rappresentato da un grafo, semplicemente partendo dalle foglie e salendo fino alla radice. Un possibile flusso della query in Figura 5.2 è rappresentato in Figura 5.3.

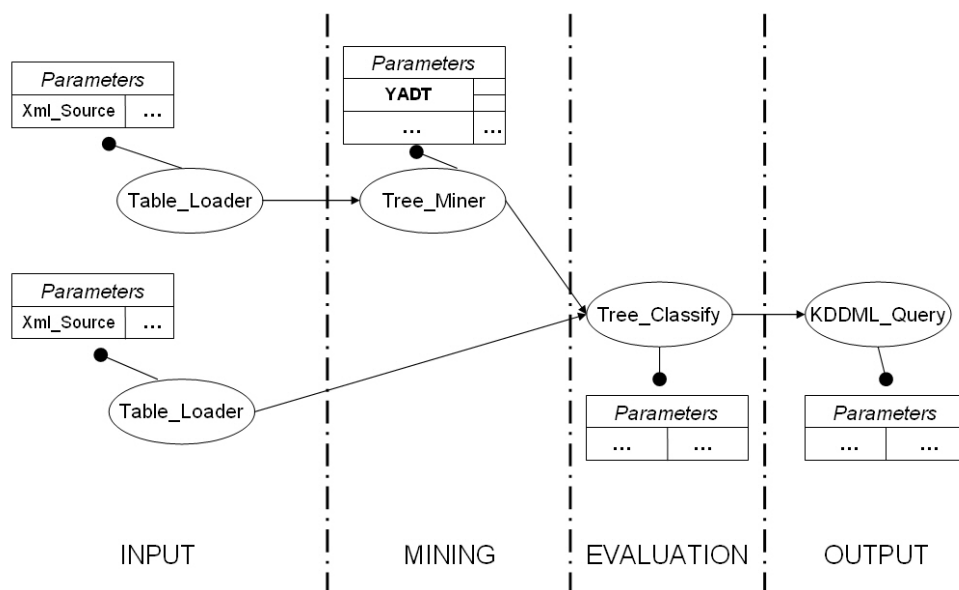


Figura 5.3 Grafo rappresentante la query di Figura 5.2.

Abbiamo appena visto che tale trasformazione può essere attuata partendo dall'albero DOM della query, ed adottando opportuni accorgimenti per rappresentare concetti riguardanti attributi e parametri.

Purtroppo, durante la stesura del processo, ci troviamo a risolvere il problema inverso, partendo dal flusso di dati, rappresentato da un grafo, dobbiamo garantire all'interprete sottostante l'albero DOM.

Nel pensare un linguaggio grafico per il sistema KDDML, al di là della sua estetica, dobbiamo quindi fissare dei punti fermi, su cui basare le nostre osservazioni. Tali punti riguardano la necessità di non stravolgere completamente il linguaggio, né tanto meno l'interprete. Per correttezza, diciamo che con l'introduzione della GUI, dovranno essere apportate alcune modifiche all'interprete, per garantire la piena soddisfazione dei vari requisiti che incontreremo nella nostra trattazione. L'eventualità che siano necessarie alcune modifiche e la modalità della loro eventuale attuazione verranno discusse di volta in volta, e comunque in tutti i casi si cercherà di non modificare la sintassi del linguaggio e di limitare il più possibile le modifiche all'interprete, ed in generale all'intera architettura.

La nuova GUI, dal punto di vista dell'architettura, deve essere concepita come quella attuale, quindi non può essere integrata nel core del sistema. Grazie a tale caratteristica, il sistema mantiene la sua indipendenza dal livello GUI, e può essere utilizzato indipendentemente dalla presenza della stessa. Inoltre applicazioni diverse possono linkare al loro interno l'interprete KDDML come libreria esterna, al fine di eseguire alcuni passi di KDD. Avendo fatto le opportune considerazioni, andiamo a studiare come possono essere legati il linguaggio grafico adottato a livello GUI ed il linguaggio KDDML.

Gli approcci possibili sono molteplici, ed in particolare, a nostro avviso, possono essere divisi in due famiglie, da un lato possiamo avere un approccio più "interpretativo" e dall'altro uno "compilativo".

Dal punto di vista logico, l'approccio "interpretativo" prevede una corrispondenza marcata tra ciò che viene rappresentato graficamente e la query KDDML prodotta.

Il linguaggio adottato al livello grafico, oltre a possedere informazioni prettamente grafiche, si discosterebbe poco dalla sintassi attuale di KDDML, e la soluzione adottata sarebbe molto simile a quella vista per YALE, in cui man mano che si costruisce graficamente il processo, viene modificato anche il file XML corrispondente. Una possibile soluzione per questo approccio potrebbe essere la seguente. Partendo dall'unico nodo finale, che può essere rappresentato

graficamente o meno, si visitano ricorsivamente tutti i suoi predecessori, fino ad arrivare al nodo che rappresenta un input per il processo, ed è riconoscibile, poiché non ha predecessori. Per ogni operatore attraversato si genererebbe l'opportuno codice XML, facendo particolare attenzione a come trattare elementi che graficamente sono rappresentati come parametri, mentre nella sintassi KDDML sarebbero tag e quindi nodi dell'albero DOM corrispondente.

Brevemente, si creerebbe una sottoquery per ogni cammino che dal nodo finale del grafo orientato porta al nodo in input, semplicemente invertendo il senso del collegamento. La query potrebbe essere generata durante la costruzione, in quanto il sistema ogni volta che si inserisce un operatore interpreterebbe il nuovo flusso implicitamente unendo il nuovo operatore con il nodo finale.

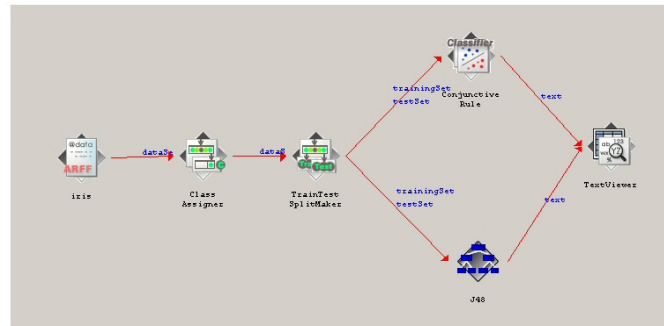
Questa soluzione appare piuttosto semplice, ma ci sono comunque alcune problematiche da considerare.

Un primo problema riguarda la gestione dell'ambiguità, poiché la visita a ritroso del grafo potrebbe non essere univoca e generare query KDDML diverse, semplicemente perché i vari operatori sono disposti in maniera diversa nello spazio, cambiando così l'ordine di esecuzione degli stessi.

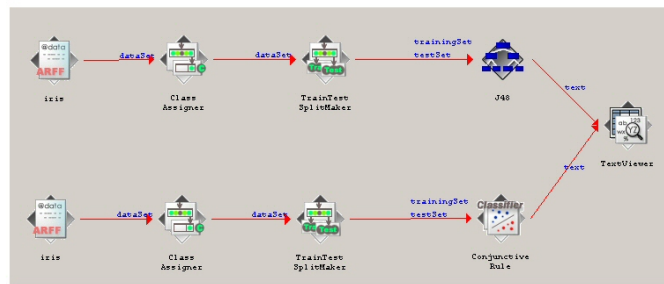
Un altro problema ben più grave riguarda la possibilità di introdurre con questo procedimento una grossa fonte di inefficienza.

Per comprendere meglio questa problematica, riprendiamo una figura già vista nel capitolo 4, in cui sono presenti due semplici flussi di dati generati con WEKA. Brevemente ricordiamo che la Figura 5.4 mostra due processi, che seppur piuttosto simili, concettualmente rappresentano due forme di estrazione di conoscenza diversa, in quanto uno vincola la conoscenza ad essere estratta da un solo data set, mentre l'altro estrae i due modelli diversi da data set indipendenti.

Per quel che riguarda la Figura 5.4 b) l'albero DOM ottenuto dal procedimento di visita a ritroso è del tutto simile a quello che si otterrebbe dalla query in Figura 5.1 del paragrafo 5.2, e rappresentato in Figura 5.5.



a)



b)

Figura 5.4. a) Estrazione di due modelli dallo stesso data set.
b) Estrazione di due modelli da data set indipendenti in WEKA.

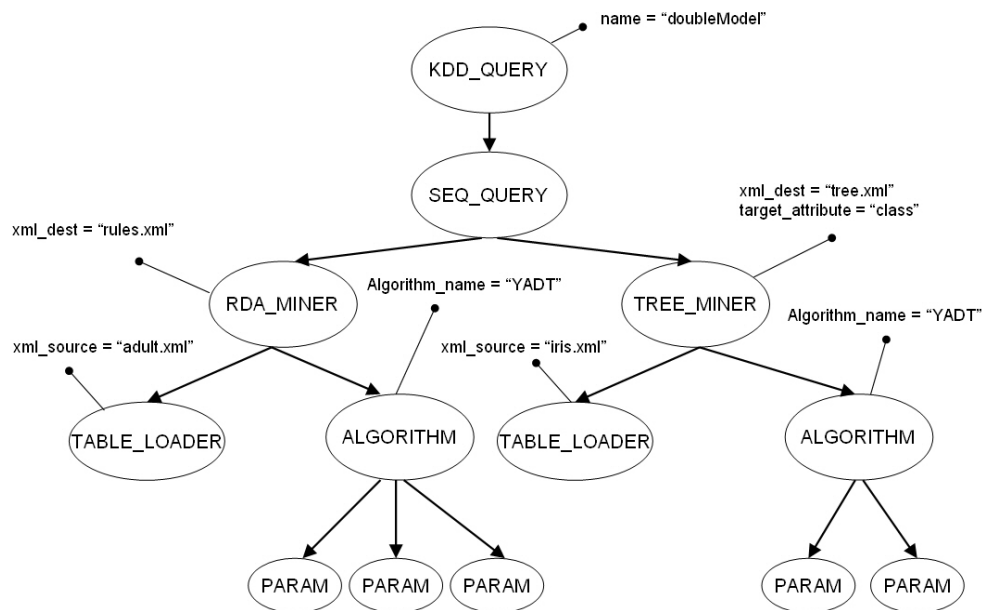


Figura 5.5 Albero DOM per la query di Figura 5.1.

Purtroppo, i problemi nascono nell'interpretare processi come quelli rappresentati in Figura 5.4.a. Questo processo ha infatti una fase iniziale comune, per poi dividersi in due flussi per estrarre modelli differenti. La sua interpretazione in KDDML, vista attraverso l'albero DOM (Figura 5.6), risulta la stessa, al di là degli specifici operatori presenti, di quella dell'albero DOM generato dalla Figura 5.4.b.

È facile convincersi di ciò, in quanto seguendo a ritroso il grafo, otteniamo due cammini che da *text viewer* arrivano al nodo *iris*. Questi due cammini hanno dato origine a due sottoquery KDDML. Andando ad analizzare meglio Figura 5.6, ci possiamo accorgere che essa è però formata da due sottoquery uguali, evidenziate in Figura 5.6 da un bordo grigio, che verranno eseguite due volte, pur producendo lo stesso risultato.

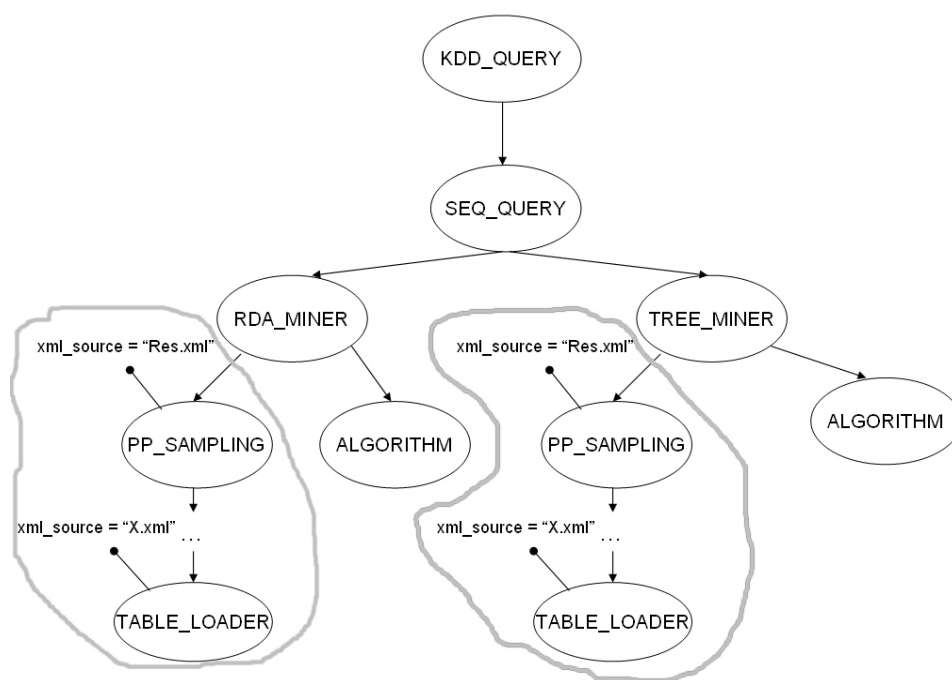


Figura 5.6 Pseudo-Albero DOM per la query di Figura 5.4.b).

I problemi sorgono quindi in fase di esecuzione della query; se ad esempio, la fase in comune è una fase di preprocessing molto costosa in termini temporali, essa viene ripetuta due volte, incidendo in maniera negativa sulle prestazioni del sistema.

In generale, questo approccio non può essere adottato in modo superficiale, in quanto farebbe perdere efficienza al sistema. Esso può risultare valido solo se affiancato da un ulteriore passo di ottimizzazione, come avviene tuttora in SQL. Tale ottimizzatore prenderebbe in input la query KDDML e ne studierebbe la struttura, andando ad esempio a ricercare le sottoquery comuni presenti al suo interno. Una volta individuate queste ultime, verrebbe ridotta la complessità delle query, eseguendo solo una volta tale sottoquery, ed in tutte le altre occorrenze verrebbe sostituita con il risultato ottenuto dall'unica esecuzione. Questa sostituzione è possibile in KDDML, poiché il linguaggio rispetta il principio di chiusura.

L'approccio "compilativo" prevede l'adozione di un formalismo proprietario a livello GUI, che indipendentemente dalla sintassi del linguaggio KDDML, rappresenta i flussi di dati in maniera autonoma, senza voler mantenere alcuna similarità e corrispondenza con esso. Questo approccio prevede l'introduzione di un compilatore che traduca il processo KDD dal linguaggio grafico a KDDML, ed anche in questo caso, insieme al compilatore può essere introdotto un modulo ottimizzatore.

L'implementazione di questo meccanismo è sicuramente più complessa, di come può apparire dalle semplici osservazioni appena descritte, ma queste ultime ci permettono di dire che il requisito *R. 2.0* può essere soddisfatto senza modifiche rilevanti al sistema attuale. La scelta del grafo è inoltre dettata dalla possibilità del sistema di soddisfare anche il requisito *R. 1.1*. Un gran numero di problematiche dovranno essere affrontate in sede di implementazione, come ad esempio la suddivisione in fasi del KDD, che richiede la definizione del numero massimo di fasi inseribili nella creazione di un processo, e come interagire con l'utente per la scelta delle fasi da inserire.

Un numero considerevole di scelte devono essere quindi prese in considerazione anche sull'architettura della sola interfaccia grafica, come ad esempio se integrare il compilatore all'interno di essa, cosiccome se introdurre il passo di ottimizzazione

direttamente nel compilatore, o se prevederlo legato all'interprete del linguaggio, in modo da ottimizzare qualsiasi query KDDML redatta anche senza l'ausilio dell'interfaccia grafica.

Riprenderemo questo discorso nel paragrafo 5.9, quando andremo a parlare delle opzioni di salvataggio del processo che la GUI deve fornire, ed introdurremo, nelle conclusioni, una possibile stratificazione dei livelli con l'introduzione del livello GUI.

Oltre a queste considerazioni prettamente grafiche, bisogna anche considerare i costrutti che il linguaggio KDDML ci mette a disposizione.

Sotto questo punto di vista il linguaggio risulta piuttosto ricco e prevede anche costrutti che difficilmente si prestano ad essere schematizzati attraverso una metafora grafica. Questi costrutti comprendono il condizionale (<IF>), la chiamata a programmi esterni (<EXT_CALL>), la possibilità di specificare l'esecuzione seriale o parallela dei vari operatori presenti nella query (<SEQ_QUERY> e <PAR_QUERY>), cosiccome la possibilità di scrivere e successivamente invocare query parametriche (<CALL_QUERY>).

Per i primi basti pensare al dover schematizzare una query con uno o più costrutti condizionali annidati, in cui ad ogni condizione corrisponda un diverso processo di mining, sia per tipo di operatori utilizzati, che come fasi costituenti. Dal punto di vista grafico sarebbe piuttosto arduo gestire i vari contesti che si verrebbero a creare, mantenendo inalterata la semplicità di costruzione del processo visto nella sua interezza. Discorso a parte meritano comunque i costrutti <SEQ_QUERY> e <PAR_QUERY>, in quanto questi due operatori non devono essere rappresentati graficamente, ma sono necessari, come abbiamo visto dagli esempi appena illustrati, alla costruzione di alcuni processi. Quindi, seppur non esplicitamente inseribili dall'utente, il loro uso è obbligatorio. Inoltre il passo di ottimizzazione potrebbe riguardare anche la sostituzione di catene di operatori seriali con operatori eseguibili in parallelo etc.

Per quanto riguarda la scrittura e l'invocazione di query parametriche, in fase di stesura non avremmo alcun controllo sui parametri ammissibili, e non potremmo garantire una certa modularità di costruzione. Pensiamo ad esempio ad una query in

cui i parametri di un algoritmo da utilizzare vengono fissati solo al momento dell'invocazione della query. Su tali parametri al momento della stesura non possiamo dire niente, e quindi nemmeno fornire i giusti menù di input nella GUI.

La corrispondenza tra ciò che è rappresentabile attraverso la GUI e ciò che è esprimibile con il linguaggio KDDML, a nostro avviso, non può essere biunivoca, in quanto quest'ultimo presenta caratteristiche che appartengono più ad un linguaggio di programmazione vero e proprio, che ad un solo strumento per produrre processi KDD.

Bisogna in ogni caso garantire che tutto ciò che è possibile esprimere attraverso la GUI rappresenti una query valida per il sistema KDDML.

5.4 KDDML & ESPANDIBILITÀ

“R. 3.0: Il kernel del sistema deve essere facilmente espandibile.”

R. 3.1: Non deve essere necessario modificare il livello GUI ogni volta che si estende il sistema.”

“S. 3.0: Possibile procedura guidata, a livello GUI, per l'estensione del sistema.”

KDDML garantisce come punto di forza la facile espansione del sistema. Questa prevede, in base al tipo di caratteristica da aggiungere, di agire su una o più parti dell'architettura e sull'implementazione di precise interfacce JAVA. Come visto nel paragrafo 2.4.2, KDDML presenta un'architettura su più livelli, ed è facilmente espandibile grazie a tale suddivisione, permettendo all'utente di modificare solo alcuni livelli e non necessariamente l'intero sistema.

Di seguito forniamo la tipologia di caratteristiche inseribili nel sistema, già viste nel capitolo 2, e per ognuna di esse delineiamo le parti del sistema da modificare.

In KDDML è possibile inserire, nuovi:

- *Data source*: l'introduzione di nuove sorgenti dati prevede l'inserimento a livello *repository* degli opportuni *wrapper*, per trasformare i dati dal formato sorgente a quello utilizzato da KDDML e viceversa. Sono inoltre richieste

modifiche al livello *operators*, per introdurre nel linguaggio gli operatori che permettono di utilizzare effettivamente la nuova sorgente introdotta.

- *Algoritmi / Operatori*: l'introduzione di nuovi algoritmi richiede solo di toccare il livello *operators* con l'introduzione di una classe JAVA, che fornisce l'implementazione di opportune interfacce (JAVA), senza bisogno di nessun'altra modifica.

Anche l'introduzione di un nuovo operatore necessita di modifiche al livello *operators*, ma dell'implementazione di opportune interfacce richiede l'estensione di alcune classi JAVA. Inoltre devono essere modificate anche le DTD, per inserire la struttura sintattica dell'operatore nel linguaggio, e per garantire al sistema il corretto uso e controllo dei sottoelementi, in fase di utilizzo ed esecuzione del nuovo operatore.

- *Modelli*: l'introduzione di nuovi modelli equivale ad inserire un nuovo tipo nel sistema. Questa modifica è sicuramente quella di gran lunga più difficile e laboriosa, in quanto richiede sia l'introduzione di nuovi algoritmi per l'estrazione della conoscenza vera e propria, sia l'introduzione di tutti gli operatori per permettere all'utente di manipolare il nuovo tipo di conoscenza inserito nel sistema. Potrebbe inoltre richiedere nuove sorgenti dati per caricare dati proprietari, su cui estrarre la nuova conoscenza. Per attuare questa modifica, potrebbe essere quindi necessario modificare sia il livello *operators* che quello *repository*, oltre alla modifica delle DTD per l'inserimento dei nuovi operatori.

La caratteristica da evidenziare è che in tutti i casi sopra elencati, non vi è la necessità di toccare il livello *interpreter*, che è il motore di tutto il linguaggio, garantendo quindi ottima modularità ed indipendenza tra i vari livelli dell'architettura.

Il requisito *R. 3.0* è soddisfatto, poiché l'espansione del sistema è in linea di principio guidata dalla realizzazione di opportune interfacce JAVA, dettate dal tipo di caratteristica da aggiungere al sistema. Sono necessarie sicuramente buone

conoscenze di JAVA, dato che si parla dell'espansione di un sistema piuttosto complesso, ma oltre a questo, un utente non si trova a dover modificare codice già scritto da altri, o dover rimodellare l'intero sistema. Lo sforzo richiesto è solo quello di valutare cosa le varie interfacce JAVA richiedono, ed implementare i rispettivi metodi.

Per completezza, diciamo che la difficoltà di espansione del sistema KDDML è legata innanzitutto al tipo di caratteristica da aggiungere, e possiamo notare che aggiungere un nuovo algoritmo è sicuramente più semplice che inserire un nuovo tipo di data source o un modello.

Questa osservazione ci permette di precisare che la possibilità di espandere il sistema con nuovi modelli (o sorgenti dati) è una caratteristica rivolta a favorire gli sviluppatori del software stesso, nell'introduzione di nuove metodologie per garantire ad utenti finali un prodotto sempre aggiornato. L'idea è quella che un utente può avere comunque la necessità di inserire algoritmi, o operatori propri per estrarre o manipolare modelli già previsti nel linguaggio, mentre difficilmente un utente finale escogiterà nuovi modelli di DM.

In Figura 5.7 è sintetizzata tale filosofia. Partendo dagli algoritmi ed andando verso il basso aumenta la difficoltà di espansione, ma diminuisce la probabilità che un utente decida di inserire un determinato elemento nel sistema. La linea tratteggiata demarca un confine tra ciò che può essere inserito dall'utente e ciò che invece richiede uno studio approfondito del sistema per il suo inserimento.

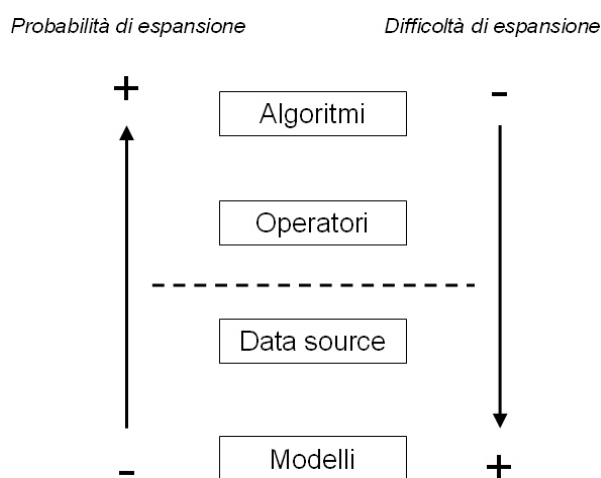


Figura 5.7 Schema riassuntivo sull'espansione di KDDML.

In generale, possiamo affermare che estendere KDDML è leggermente più complicato rispetto ai sistemi basati su JAVA analizzati nel capitolo 3, come ad esempio YALE³⁴. Questa maggiore difficoltà è legata alla necessità, in alcuni casi, come quando si inserisce un nuovo operatore, ma non quando si inserisce un nuovo algoritmo, di modificare anche le DTD, costringendo l'utente a dover prendere confidenza anche con questo strumento. Un vantaggio, tuttavia, in questo caso è rappresentato dal fatto che KDDML, rispetto a YALE, garantisce controlli più accurati sui dati già attraverso l'analisi statica, mentre YALE li demanda a run-time.

Questa politica di gestione dell'espandibilità deve essere adottata anche per il livello GUI, e quindi è evidente che i requisiti *R. 3.0* e *R. 3.1* sono in stretta correlazione.

Infatti, se integriamo i concetti appena menzionati, risulta chiaro che con l'introduzione della GUI, nel momento in cui andiamo ad espandere il sistema, dobbiamo fornire per ogni nuova *feature* inserita non solo il core funzionale, ma anche una serie d'informazioni aggiuntive per permettere di catturare a livello grafico non solo, il fatto che tale *feature* è stata inclusa nel sistema, -ad esempio inserendola in un menù-, ma anche a quale fase del KDD appartiene, per garantirne il giusto inserimento solo in determinate fasi.

Inoltre dobbiamo visualizzare anche il numero ed il tipo di parametri eventualmente richiesti, per fornirne la giusta visualizzazione. Ad esempio, se utilizziamo un algoritmo per estrarre RdA, bisogna vincolare la visualizzazione e l'immissione per il valore dei parametri di confidenza e supporto tra 0 e 1, oppure se abbiamo altri parametri a valori booleani, bisogna presentare un box che permetta di scegliere solo tra i valori TRUE o FALSE, etc.

Purtroppo *R. 3.1* è un requisito piuttosto complesso da soddisfare, soprattutto tenendo conto che può essere in contrasto con *R. 3.0*. I metodi possibili in questo caso sono, a nostro avviso, essenzialmente due, e rappresentano in entrambe i casi una via di mezzo tra mantenere la facilità di espansione attuale e garantire la completa indipendenza al livello GUI.

³⁴ In questo caso si fa riferimento alla sola espansione del sistema base privo dell'interfaccia grafica, in quanto in tal caso anche YALE prevede la redazione di un opportuno file XML.

La prima è del tutto simile a quella adottata da YALE (par. 3.3.1). Ogni volta che si inserisce un nuovo elemento nel sistema, a livello GUI deve essere fornito o modificato un file di configurazione, attraverso il quale specificare la serie di informazioni di cui abbiamo necessità. Questa strada lascerebbe sostanzialmente immutati i vantaggi di estendibilità del sistema attuale, ma metterebbe l'utente davanti alla difficoltà di dover stilare un file di configurazione utilizzando ad esempio XML, con l'onere di dover prendere confidenza anche con questo mezzo e soprattutto con la politica adottata nell'implementazione di tale file. In più, il requisito *R. 3.1* non sarebbe completamente soddisfatto e si perderebbe l'attuale modularità fornita dal sistema. L'interfaccia non si baserebbe sui livelli sottostanti, ma utilizzerebbe solo informazioni a livello locale, garantite dal file di configurazione. Ad esempio un utente potrebbe modificare solo il file di configurazione GUI in maniera erronea, credendo di aver inserito anche l'operatore nel sistema. Questa scelta a nostro avviso farebbe perdere robustezza all'intero sistema, in quanto gli elementi "inseriti" a livello GUI, ma non integrati nel sistema, darebbero luogo a query non valide o a comportamenti anomali; basti pensare al caso di omonimie tra operatori forniti ex-novo nel file di configurazione GUI, e operatori già presenti nel sistema di base.

L'altra possibile soluzione prevede una maggiore specializzazione, rispetto all'uso attuale delle interfacce JAVA, attraverso le quali l'interprete collabora con i livelli sottostanti. Attualmente, a titolo esemplificativo, notiamo che chi vuole aggiungere un nuovo algoritmo in KDDML deve andare a livello *operators*, ed implementare i metodi previsti dalle interfacce *AlgorithmResolverTask* e *AlgorithmSettingsTask*.

Semplificando, possiamo dire che la prima riguarda il nucleo operativo dell'algoritmo, e quindi il codice per l'implementazione vera e propria, invocato dall'interprete per eseguire l'algoritmo. I metodi da implementare in questa interfaccia sono "task-dependent" e variano a seconda del tipo di algoritmo che vogliamo introdurre nel sistema. Dell'interfaccia *AlgorithmSettingsTask* fanno parte i metodi che permettono all'interprete di attuare una serie di controlli sui parametri richiesti dall'algoritmo. Attraverso tali metodi vengono effettuati controlli sul tipo e sul valore passato in input per un determinato parametro, per valutare se tale valore è ammissibile.

Risulta chiaro quindi il bisogno di inserire una nuova interfaccia JAVA da utilizzare esclusivamente per fornire informazioni al livello GUI. Tale interfaccia dovrebbe tenere in considerazione i molteplici fattori e caratteristiche, in parte definiti precedentemente, che sono ritenuti utili a livello GUI.

Questo approccio ha sicuramente il pregio di soddisfare il requisito *R. 3.1*, ma rende più complessa l'espandibilità, andando in parte contro il requisito *R. 3.0*. Come già discusso nel paragrafo 4.6, l'inserimento dell'interfaccia in un sistema già esistente, come in questo caso, può vanificare tutti gli sforzi perseguiti in passato, per mantenere bassa la difficoltà di espansione.

Com'è intuibile, l'inserimento di un'interfaccia JAVA dedicata esclusivamente alla GUI complica le cose, soprattutto se quest'ultima prevede un gran numero di metodi, che possono scoraggiare l'utente dall'intraprendere tale percorso. Al fine di mantenere tale meccanismo più semplice possibile, possiamo adottare alcuni accorgimenti.

Il primo potrebbe riguardare la possibilità di permettere all'utente di espandere il sistema senza essere vincolato a dover inserire tutti i metodi per l'espansione dell'interfaccia grafica. Tutto questo informando l'utente che l'operatore non sarà disponibile a livello grafico, ma il suo utilizzo sarà comunque possibile attraverso l'invocazione con query in KDDML-XML, come avviene tuttora.

L'altro metodo riguarda la possibilità di eliminare, o per meglio dire spostare la parte del controllo dei parametri nell'interfaccia JAVA riguardante la GUI (*interfaceGUI*), ed esonerare l'interprete dai controlli sui parametri, che verrebbero attuati al momento dell'immissione dei dati direttamente al livello GUI. Quest'ultima scelta da una parte ridurrebbe di nuovo le interfacce JAVA da implementare, ma farebbe perdere modularità al sistema, soprattutto se si mantiene la possibilità di utilizzarlo anche sottoponendogli direttamente query KDDML redatte senza l'utilizzo della GUI. In questo caso, infatti, non possiamo fidarci solo dei parametri forniti dall'utente, e l'interprete si deve fare carico di effettuare tali controlli. Quindi i metodi previsti dall'interfaccia *Settings* verrebbero spostati nella *interfaceGUI*, ma fondamentalmente la complessità rimarrebbe invariata e si perderebbe l'attuale distribuzione dei compiti garantiti dai vari livelli. Inoltre sarebbero necessari dei metodi per indicare all'interprete se effettuare il controllo sui

parametri, e se la query è stata redatta attraverso la GUI o manualmente.

Queste considerazioni vanno comunque intese a carattere generale, per delineare le problematiche che dovranno essere risolte nel momento in cui si andrà ad implementare l'interfaccia grafica vera e propria.

La caratteristica da evidenziare maggiormente in questo caso riguarda la necessità di modificare i livelli sottostanti, e l'entità di questa modifica è legata alle caratteristiche che noi vogliamo garantire all'interfaccia, in parte già indicate nei requisiti e nei suggerimenti nella paragrafo 4.8, e ciò che vogliamo garantire attraverso il sistema indipendentemente dalla sua interfaccia grafica.

Tutte queste considerazioni potrebbero aiutare anche nell'implementazione, come suggerito da *S. 3.0*, di uno strumento ad alto livello che permetta all'utente di estendere il sistema con nuovi algoritmi, senza accedere fisicamente alla struttura delle classi, e fornendogli un *editor* che, in base alla caratteristica dell'algoritmo da inserire, gli presenti i vari metodi da dover implementare.

5.5 KDDML & META-ESECUZIONE

“R. 4.0: Evitare l'esecuzione di query non valide.”

R. 4.1: Utilizzare meta-dati e meta-modelli per attuare la meta-esecuzione.

R. 4.2: Fornire la GUI di semplici strumenti per il reperimento, la guida nella correzione degli errori eventualmente commessi e per attuare la meta-esecuzione e l'esecuzione vera e propria.”

“S. 4.0: Aggiunta di strumenti per l'esecuzione parziale della query.”

KDDML attualmente prevede già la caratteristica di non permettere l'esecuzione di query non valide. I controlli sulla query riguardano principalmente l'aspetto sintattico - attraverso le DTD si vede se la query è sintatticamente corretta-, e aspetti che riguardano i tipi ed i parametri utilizzabili per i vari operatori. Diciamo quindi che il requisito *R. 4.0* è già soddisfatto dal sistema, che prima di eseguire la query vera e propria effettua un'analisi statica.

Per quel che riguarda *R. 4.1*, bisogna dire che il sistema prevede già, come visto nel capitolo 2, meta-informazioni per gestire dati e modelli, ma attualmente non vi è alcun controllo statico su tali strutture.

Il requisito di fornire il sistema della possibilità della meta-esecuzione richiede opportune modifiche sia a livello *interpreter*, che a livello *operators*.

L'interprete dovrebbe essere modificato per attuare la meta-esecuzione, in quanto nell'attuale implementazione abbiamo solo la possibilità dell'analisi statica. Mentre a livello *operators* occorrerebbe inserire i vari metodi per permettere all'interprete di accedere alle sole meta-informazioni, senza accedere ai dati fisici, necessari per attuare la meta-esecuzione.

Per quanto riguarda R. 4.2, l'introduzione della finestra di output per presentare gli errori è piuttosto semplice, mentre nel caso di opportuni bottoni per la sola meta-esecuzione, prevede che l'interprete divida le due fasi in modo da poter richiamare solo la meta-esecuzione. Rimane fermo il fatto che se si cerca di eseguire la query, sia i controlli statici sia quelli previsti dalla meta-esecuzione devono essere attuati senza esplicita richiesta dell'utente. Sotto questo punto di vista possono essere escogitati dei meccanismi che evitino di effettuare i controlli statici e la meta-esecuzione, nel caso siano già stati effettuati in precedenza, e la query da questi controlli non sia stata modificata, velocizzando l'intero meccanismo, peraltro non così gravoso in termini di tempo.

Sui controlli statici può essere fatta un'ulteriore osservazione. Essi riguardano per lo più aspetti sintattici, e di tipi. Con l'introduzione dell'interfaccia, tali controlli risulterebbero ridondanti, poiché deve essere compito della GUI evitare che l'utente crei query scorrette dal punto di vista dei tipi, ad esempio impedendo che due operatori incompatibili possano essere collegati. Riguardo alla parte sintattica, sarà il passo di trasformazione della query da visuale a KDDML a garantirne la correttezza.

Questa osservazione deve tenere conto anche della possibilità di sottomettere al sistema query redatte senza l'ausilio della GUI, nelle quali l'analisi statica deve essere attuata.

Purtroppo anche nel caso della meta-esecuzione bisogna prestare particolare attenzione a due fattori; il primo è lo stesso individuato nel paragrafo 5.4, ovvero la necessità di offrire sempre maggiori servizi, senza per questo incrementare a dismisura il livello di complessità dell'espansione del sistema.

Il secondo riguarda il fatto di dover sempre attuare controlli a run-time, anche se la meta-esecuzione va a buon fine. Questo è necessario, poiché quest'ultima non è condizione sufficiente a garantire la correttezza della query. Tale precisazione è in parte ovvia, la teoria ci dice che in un sistema, attraverso la sola analisi statica non è possibile in generale garantire la completa correttezza di un programma, poiché non abbiamo informazioni sufficienti per testarla. In parte può essere utile precisare che durante la meta-esecuzione, noi agiamo sulla struttura logica dei dati e non sui dati fisici. Tale schema logico potrebbe essere stato intenzionalmente manomesso, ed in realtà non rappresentare più i dati fisici su cui, una volta in esecuzione la query, il sistema lavora.

Da un punto di vista prettamente grafico, l'interfaccia deve fornire semplici mezzi, come bottoni, per attuare la meta-esecuzione e l'esecuzione vera e propria del processo, soddisfacendo *R. 4.2*. Un metodo per impedire l'esecuzione di query non valide potrebbe essere quello di disabilitare il bottone di *Execute* fino a che il processo non ha superato il test di meta-esecuzione.

Merita tuttavia un maggior approfondimento la modalità di invocazione della meta-esecuzione. Essa infatti può essere invocata esplicitamente dall'utente dopo aver completato la stesura del processo, o quando lo si ritiene opportuno. Nonostante che questa possibilità presenti alcuni vantaggi, può limitare le reali potenzialità di tale meccanismo, anche durante la stesura grafica del processo KDD.

Un approccio alternativo all'invocazione esplicita può essere quello di attuare automaticamente la meta-esecuzione in maniera trasparente all'utente. Tale procedimento, manipolando solo meta-informazioni, sarebbe piuttosto rapido e non comporterebbe eccessivi rallentamenti al sistema. L'invocazione della meta-esecuzione avverrebbe implicitamente ogniqualvolta si connettono due nodi del grafo.

Seguendo questo approccio, l'utente viene maggiormente guidato nell'immissione dei parametri per i vari operatori. Per chiarire quanto appena detto, introduciamo una situazione tipica in cui la meta-esecuzione implicita garantirebbe ottime potenzialità anche a livello visuale. In Figura 5.8 è rappresentato un processo di KDD in cui abbiamo un operatore per leggere un file *arff* (*Arff Loader*) e due operatori di preprocessing, *Remove Attribute* e *Rename Attribute* in sequenza, rispettivamente

per rimuovere e rinominare un attributo all'interno di una tabella. Inoltre nella parte bassa vi sono i meta-dati relativi all'esecuzione dell'operatore soprastante. L'idea è quindi quella di offrire ad ogni passo il maggior numero di informazioni possibili sui dati, come la lista degli attributi su cui poter attuare i passi di preprocessing. In generale, è importante riuscire a visualizzare le informazioni sullo schema logico dell'oggetto in input ad ogni operatore.

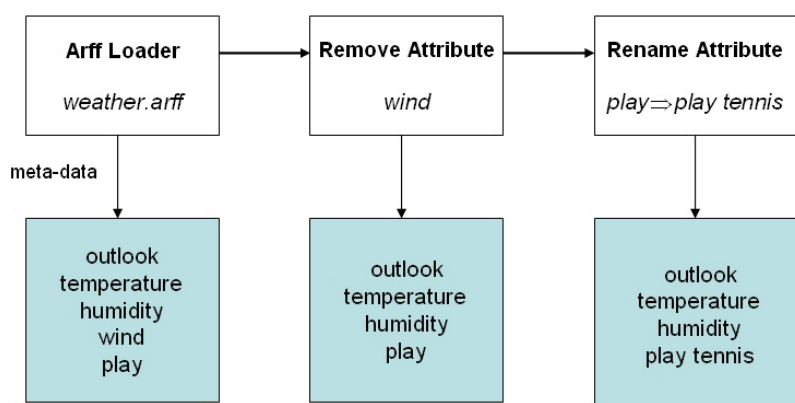


Figura 5.8 Esempio di processo KDD con utilizzo di meta-dati.

I problemi nascono quindi nell'avere per ogni operatore le informazioni sui meta-dati da visualizzare, come ad esempio il nome degli attributi presenti nel file *arff*, selezionabili attraverso un *combo box*. Queste informazioni non sono disponibili staticamente, poiché nessuno degli operatori presenti è stato ancora eseguito.

Attuando la meta-esecuzione, non appena avviene il collegamento tra due operatori, otterremmo ad ogni passo, la struttura logica dei dati, e potremmo visualizzare gli attributi su cui effettivamente operare, per ogni operatore di preprocessing nella catena.

Nel caso dell'esempio in questione, nel *combo box* relativo all'operatore *Rename Attribute* non sarà più presente l'attributo *wind*, poiché eliminato al passo precedente dall'operatore *Remove Attribute*.

Questo rappresenta solo un semplice esempio di applicazione. In realtà vi sono innumerevoli problematiche da affrontare per implementare questo tipo di meccanismo, ad esempio il linguaggio grafico deve permettere la meta-esecuzione anche di query parziali, e quindi deve permettere trasparentemente all'utente di poter

sottoporre sottoquery al sistema, etc. Comunque questo esempio ci permette di capire che l'introduzione del passo di meta-esecuzione non solo potenzierebbe enormemente il sistema indipendentemente dalla sua interfaccia grafica, ma potrebbe aiutare quest'ultima a potenziare le possibilità offerte all'utente nella stesura del processo.

Il suggerimento *S. 4.0* ci permette di concludere questo paragrafo, parlando degli strumenti per un'esecuzione parziale della query. Dal punto di vista grafico, questo compito non prevede grosse difficoltà, basta dare la possibilità all'utente di inserire dei *break point* dopo gli operatori o prevedere la disabilitazione di collegamenti tra di essi. Con l'introduzione di questa caratteristica, potrebbero essere necessarie modifiche sia a livello *interpreter*, per evitare la visita di sottoalberi DOM esclusi dall'esecuzione, sia al linguaggio, per esprimere quali rami non devono essere valutati dall'interprete.

Un altro modo potrebbe essere quello di spezzare la query in più sottoquery ed instanziare l'interprete solo sulla sottoquery richiesta. Seguendo questa strada nessuna modifica sarebbe richiesta né a livello *interpreter* né al linguaggio. Mentre sarebbe l'interfaccia grafica che in maniera trasparente all'utente dovrebbe sottoporre al sistema solo una parte della query.

5.6 KDDML & IMMISSIONE PARAMETRI

“R. 5.0: Deve essere garantita massima semplicità per immissione, visualizzazione e reperimento dei vari parametri degli operatori.”

R. 5.1: Per ogni parametro inseribile deve essere fornito un help on line riguardante la spiegazione del significato di tale parametro all'interno del contesto di applicazione.”

“S. 5.0: Il nucleo del sistema deve fornire a livello GUI informazioni sufficienti per attuare controlli e visualizzazioni specifiche sui vari parametri ammissibili.”

Relativamente a questi requisiti, nel sistema esistente non abbiamo molto da osservare, poiché chi implementerà l'interfaccia grafica avrà piena libertà di strutturarli e rappresentarli come riterrà opportuno.

Una possibile soluzione per soddisfare *R. 5.0* potrebbe essere garantita fornendo ogni operatore di un menù di immissione che si apre al momento del primo inserimento o collegamento dello stesso, e raggiungibile attraverso un bottone, magari posto a lato dell'operatore, oppure selezionando l'operatore stesso e premendo un bottone specifico per la visualizzazione dei parametri.

Il meccanismo dovrebbe inoltre evitare la possibilità di aprire contemporaneamente più menù di immissione parametri riguardanti operatori diversi, come può accadere in WEKA Knowledge Flow GUI o in ORANGE.

Anche per la visualizzazione dell'help on-line riguardante i vari parametri, *R. 5.1*, possono essere scelte diverse possibilità, ad esempio può essere previsto un pulsante nel menù di immissione dei parametri, per visualizzare la guida al significato dei vari parametri inseribili.

Finora abbiamo analizzato i parametri solo dal punto di vista della loro visualizzazione grafica. Ricordiamo comunque che, per quel che riguarda questi requisiti bisogna intervenire anche sui livelli sottostanti del sistema. Abbiamo già introdotto le motivazioni che richiedono di intervenire su tali livelli nel paragrafo 5.4. Qui riprendiamo brevemente tali concetti, facendo riferimento alla sola problematica dell'immissione dei parametri.

Dato che reputiamo valido *R. 3.1*, senza l'utilizzo di file di configurazione a livello GUI, è facile convincersi che il suggerimento *S. 5.0* diventa in realtà un requisito. Tale requisito richiede che sia la GUI ad effettuare i controlli sulla validità dei parametri immessi. Senza addentrarci troppo nello specifico, il progettista dovrà prevedere di inserire per ogni algoritmo presente nel sistema una serie di informazioni che non solo permettano alla GUI di attuare i controlli sulla validità dei parametri appena immessi, ma che guidino anche la visualizzazione dei menù stessi. In effetti, il controllo dei parametri avviene in maniera implicita, vincolando l'utente ad immettere solo valori ammissibili.

Purtroppo, attraverso la GUI sarebbe impossibile prevedere tutte le casistiche di inserimento e controllo dei parametri, basti pensare al caso in cui un algoritmo inserito dall'utente richieda in input una lista di stringhe in un certo ordine e con un certo separatore. A livello GUI possiamo catturare solo alcune delle caratteristiche richieste, ma dobbiamo porre una limitazione, al fine di permettere una buona

implementazione sulla serie di controlli attuabili. Questo ci dice che il controllo dei parametri dovrà essere mantenuto anche a livello *interpreter*.

Il progettista dovrà in ogni caso prevedere dei formalismi da immettere ad esempio nell' *interfaceGUI*, discussa nel paragrafo 5.4, al fine di garantire alla GUI, non solo l'indipendenza dagli operatori effettivamente presenti, ma anche e soprattutto per la visualizzazione degli elementi presenti nei menù stessi, e dei vari help che devono essere visualizzati per i parametri previsti.

5.7 KDDML & VISUALIZZAZIONE

“R. 6.0: La GUI deve essere integrata con vari strumenti di visualizzazione per rappresentare graficamente l'informazione manipolata.

R. 6.1: Un numero minimo di visualizzatori deve essere introdotto implicitamente dal sistema.”

Allo stato attuale KDDML non presenta caratteristiche avanzate ed interattive per la visualizzazione dei risultati. Questi ultimi vengono mostrati senza l'utilizzo esplicito di operatori di visualizzazione e vengono presentati in formato HTML. In realtà il linguaggio non prevede ancora operatori e costrutti sintattici per inserire esplicitamente visualizzatori, quindi la conoscenza estratta viene semplicemente mostrata attraverso l'apertura di un browser HTML.

Come abbiamo già introdotto nel capitolo 2, e quando abbiamo trattato la meta-esecuzione, KDDML ha il vantaggio di rappresentare la struttura logica dei dati e dei modelli.

Attraverso questa struttura, è possibile capire il tipo di oggetto da visualizzare ed associargli un insieme opportuno di visualizzatori, senza dover andare a modificare i livelli sottostanti alla GUI.

Grazie a questa caratteristica, potrebbe risultare utile suddividere la GUI in due applicazioni distinte, e creare una GUI DEVELOPER specializzata per la costruzione di processi KDD, ed una GUI VISUALIZER atta alla visualizzazione ed all'analisi dei risultati ottenuti attraverso tali processi.

Ad esempio, se alla GUI VISUALIZER venisse passato un oggetto di tipo *tree*, l'utente si troverebbe davanti a vari tipi di visualizzatori associati a tale oggetto,

come la rappresentazione grafica dell'albero di classificazione, le regole estratte da tale albero etc.

In linea di principio per ogni tipo di conoscenza estratta dal sistema (e per i tipi rappresentanti i dati), avremmo a disposizione un insieme di visualizzazioni.

Questa suddivisione porterebbe vantaggi a livello di sviluppo. Infatti un team potrebbe concentrarsi solo sulla costruzione, o per meglio dire sull'integrazione della GUI che implementa il linguaggio grafico di KDDML, mentre un altro team si dedicherebbe al tool di visualizzazione, che risulterebbe indipendente dal linguaggio e dagli algoritmi/operatori presenti, ma che si baserebbe soltanto sul tipo di oggetti ottenibili come risultato finale del processo. Un ulteriore vantaggio riguarda il fatto che un analista potrebbe essere interessato solo a visualizzare i risultati ottenuti dai vari modelli estratti in precedenza da altre persone, mantenendo separate la fase di costruzione del processo da quella di valutazione.

Purtroppo la suddivisione in due strumenti separati, ed in parte indipendenti, potrebbe complicare la fase di valutazione della conoscenza estratta, e compromettere in parte l'interattività del sistema nella sua interezza. L'utente potrebbe infatti trovare difficoltà nell'integrare e nel vedere in realtà sotto un'unica ottica le due applicazioni, soprattutto se queste sono sviluppate con formule estetiche diverse.

La suddivisione in due applicazioni sembra pertanto una strada accettabile, dato che KDDML si basa su *repository* per dati e modelli, e quindi la parte di visualizzazione sarebbe invocata dalla GUI, semplicemente attraverso il passaggio del nome dell'oggetto da visualizzare e del *path* del *repository* da cui prenderlo.

I visualizzatori, attraverso questa suddivisione, continuerebbero a non far parte del linguaggio, e spetterà poi alla GUI VISUALIZER, attraverso il tipo di dato che le verrà chiesto di visualizzare, fornire gli strumenti ritenuti più validi, analizzare i dati specifici, soddisfacendo i requisiti *R. 6.0* e *R. 6.1*.

Anche qualora aggiungessimo nuovi algoritmi/operatori al sistema, la GUI VISUALIZER non necessiterebbe di alcuna modifica, in quanto non sarebbero stati modificati i tipi esportati dal sistema.

Dal punto di vista dell'espansione nella sua totalità, le cose in parte si complicano, e se da una parte per GUI DEVELOPER vale quanto è stato detto nel paragrafo 5.4,

è necessario studiare attentamente come prevedere l'espansione della GUI VISUALIZER, nel caso si introduca nel sistema un nuovo tipo di oggetto, con l'inserimento di modelli di mining in KDDML. Chi implementerà da zero questa interfaccia dovrà pertanto tenere conto del fatto che il tipo di oggetti esportati da KDDML può variare nel tempo, e conseguentemente anche la GUI VISUALIZER stessa deve garantire ottime caratteristiche di estensione. Non è pensabile, infatti, dover implementare ex-novo la GUI VISUALIZER ogniquale volta al sistema si aggiunga un nuovo tipo.

Quest'ultima osservazione ci permette di ragionare anche su un possibile approccio diverso, che consiste nell'integrare anche la visualizzazione dei risultati in un'unica GUI. Le politiche di gestione di questa caratteristica possono essere molteplici. La prima sarebbe del tutto simile a quella definita sopra - la visualizzazione basata sul tipo di dato da visualizzare -, solo che invece di suddividere il livello GUI in due applicazioni distinte, anche i risultati verrebbero presentati attraverso strumenti integrati in una stessa interfaccia, con l'implementazione dei visualizzatori integrati a livello GUI. Seguendo questa politica, non rispetteremmo più il requisito *R. 2.1*, in quanto potremmo aggiungere un nuovo modello al sistema, ma non toccando la GUI, non riusciremmo a visualizzare gli oggetti prodotti da quel modello.

Inoltre un'integrazione totale delle due interfacce, che soddisfi anche i requisiti di espandibilità, deve tener conto della possibilità di inserire nella sintassi del linguaggio opportuni operatori per definire esplicitamente le informazioni da visualizzare ed il tipo di visualizzatore da adottare. Bisogna inoltre decidere a che livello inserire l'implementazione dei visualizzatori, che volendo rispettare il requisito *R. 3.1*, non possono essere inseriti a livello GUI.

5.8 KDDML & MODELLI

“R. 7.0: Deve essere possibile riutilizzare la conoscenza estratta.

R. 7.1: Completo supporto al PMML.”

Il linguaggio KDDML permette, attraverso vari costrutti, di caricare modelli precedentemente estratti sia dal sistema stesso, e quindi presenti nel *model*

repository, che da fonti esterne in formato PMML. Ad esempio è possibile caricare un albero di classificazione attraverso l'operatore `<TREE_LOADER>`, che carica un albero di classificazione dal *repository* del sistema. Inoltre attraverso `<PMML_TREE_LOADER>` è possibile caricare un modello PMML preso da un file esterno che contiene un albero di classificazione espresso attraverso le specifiche di PMML 2.0.

In linea generale sono presenti operatori per caricare i vari modelli estraibili dal sistema, come *RdA*, *cluster*, *sequential pattern*, e per ognuno di questi è prevista la possibilità di importarli da fonti esterne attraverso il formato PMML.

Alla luce di quanto appena affermato, possiamo dire che il sistema a tutt'oggi soddisfa pienamente il requisito *R.7.0* ed in parte *R.7.1*.

L'unica osservazione da fare, che invalida il pieno supporto a *R.7.1*, riguarda l'utilizzo da parte del sistema di un PMML esteso per riuscire a rappresentare concetti, come la matrice di confusione, o il *committee* di alberi di classificazione, che non sono ancora previsti dallo standard PMML.

Purtroppo, oltre a questa caratteristica, il sistema non prevede nemmeno i costrutti sintattici per esportare i modelli in PMML standard. Comunque l'esportazione di modelli estratti da KDDML in PMML puro può avvenire semplicemente eliminando il tag `<Extension>` dal modello salvato da KDDML. Bisogna precisare, comunque che rispetto alla versione di KDDML, vi è una perdita di informazione, e che certi modelli, ad esempio i meta-classificatori, non sono esportabili, poiché la specifica PMML non prevede ancora uno standard di rappresentazione.

5.9 KDDML & SALVATAGGI

“R. 8.0: Garantire la possibilità di salvare/caricare la query che rappresenta il processo KDD.”

“S. 8.0: Fornire la possibilità di salvare sia l'intero processo sia i risultati ottenuti in HTML.

S. 8.1: Possibile visualizzazione/salvataggio dei risultati intermedi.”

Una volta costruito il processo, è necessario mettere a disposizione dell'utente mezzi per salvarlo, in modo da poterlo riutilizzare. Il sistema KDDML, come è lecito

aspettarsi, permette già di fare ciò, garantendo sia il salvataggio che il caricamento e la modifica di query KDDML esistenti.

Aggiungendo il livello GUI, il formato attuale di salvataggio non è sufficiente, in quanto sarebbe utile disporre nel file XML anche di una serie di informazioni riguardanti aspetti prettamente grafici, per permettere alla GUI di caricare la query presentandola graficamente nello stesso modo in cui è stata salvata. Questo aspetto è molto delicato, poiché tutte queste informazioni risultano inutili all'interprete per l'esecuzione fisica della query. Tuttavia, se dal punto di vista computazionale, ciò non modificherebbe l'esecuzione, poiché i tag riguardanti la GUI verrebbero scartati dall'interprete, dal punto di vista della leggibilità della query anche al di fuori del sistema avremmo ulteriori difficoltà.

Una possibile soluzione, legata anche a quanto detto nel paragrafo 5.3, potrebbe essere quella di fornire la query KDDML, nello stesso formato attuale, come input all'interprete per l'esecuzione fisica, adottando un formato (e formalismo) proprietario, legato al solo livello GUI, che contenga al suo interno indicazioni sul numero di fasi di cui la query è composta, gli oggetti presenti e la loro distribuzione nello spazio, insieme ai parametri in input per ogni oggetto ed ai collegamenti previsti tra gli oggetti stessi. Rimane aperta la possibilità di permettere all'utente di salvare anche la query KDDML in XML o solo in formato proprietario del livello GUI, e quindi invocabile solo attraverso l'interfaccia.

Adottando un formalismo proprietario, che permetta la gestione grafica della query, e non volendo intaccare il livello *interpreter*, è necessario introdurre un compilatore che trasformi le *query in linguaggio GUI* in *query KDDML*, rendendole quindi eseguibili.

Per quel che riguarda il suggerimento *S. 8.0*, il sistema presenta attualmente i risultati in formato HTML, è quindi necessario introdurre una rappresentazione in tale formato anche dell'intero processo.

Anche dal punto di vista dell'attuazione del suggerimento *S. 8.1*, diciamo che essa non prevede grosse difficoltà, in quanto il sistema può salvare già tutte le varie tabelle e modelli, compresi quelli intermedi, nei *repository*. Inoltre, per quanto concerne anche la scelta dei dati intermedi da visualizzare, possono essere previsti

strumenti che indichino al sistema quali delle informazioni disponibili nei vari step del processo dover inviare alla GUI VISUALIZER, al fine di visualizzarli. Queste caratteristiche sono legate sia al meccanismo scelto per visualizzare i risultati, sia ai meccanismi per permettere esecuzioni parziali delle query.

5.10 KDDML & DOCUMENTAZIONE FORNITA

“R. 9.0: Fornire il sistema ed in particolare la sua GUI di adeguata e completa documentazione.”

“S. 9.0: Introdurre una serie di tutorial, per facilitare l’utente nel prendere confidenza con il sistema.”

Quest’ultimo requisito non solo non riguarda la sola interfaccia grafica, ma si inserisce in un contesto ben più ampio dei soli sistemi per il *knowledge discovery*, in quanto sarebbe buona regola fornire qualsiasi software di adeguata documentazione indipendentemente dal suo campo di applicazione.

Purtroppo, per quanto concerne KDDML, la documentazione fornita è ancora piuttosto scarsa, ma bisogna comunque evidenziare che il sistema è ancora in fase di “alpha-testing”, e quindi pur avendo ormai una struttura ben delineata, può ancora essere soggetto ad ulteriori correzioni o modifiche.

Queste ultime saranno infatti necessarie, al fine di introdurre un’interfaccia grafica potente che soddisfi appieno i requisiti e le scelte che si andranno a delineare durante la fase di implementazione.

Conclusioni

Nel corso di questo studio, non solo abbiamo individuato un insieme di requisiti, ma abbiamo anche verificato se questi possono essere o sono già soddisfatti dal sistema KDDML. Sono stati inoltre forniti una serie di spunti sui quali riflettere prima di intraprendere l'implementazione vera e propria dell'interfaccia grafica per tale sistema.

In Figura 6.1 è rappresentata una possibile architettura ad alto livello di KDDML, basata sulle ipotesi e le osservazioni condotte nel corso del capitolo 5.

Bisogna innanzitutto evidenziare che tale architettura non è definitiva e rappresenta solo una possibile strutturazione del sistema, prevista con l'introduzione del livello GUI.

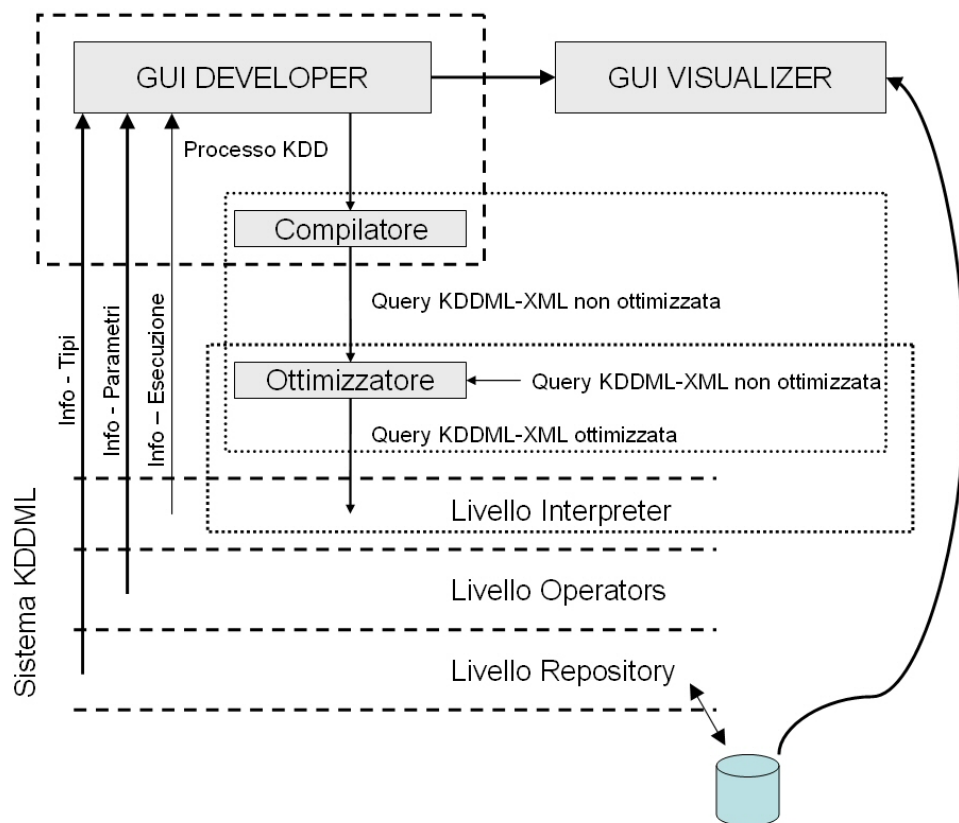


Figura 6.1 Architettura ad alto livello del sistema KDDML con l'introduzione del livello GUI.

Andiamo ad analizzare più in dettaglio tale architettura. Appare subito evidente la scelta di suddividere la GUI in due applicazioni distinte. Le uniche informazioni che le due applicazioni si scambiano riguardano il *path* che permette di localizzare l'oggetto fisico da visualizzare all'interno del *repository*. La freccia che unisce la GUI VISUALIZER al *repository* dei dati si riferisce al fatto che quest'ultima accede direttamente ai file contenuti in essa.

Per quel che riguarda la GUI DEVELOPER, essa deve interfacciarsi con tutti i livelli del sistema, per ottenere le informazioni necessarie al suo funzionamento. Questo è necessario, poiché, come richiesto dal requisito *R. 3.1*, il livello GUI non deve essere modificato nel momento in cui si espande il sistema.

Le interazioni riguardano:

- Il livello *repository*, poiché il livello GUI deve accedere ad informazioni sui tipi implementati nel sistema, ed in particolare ai modelli supportati, per garantire estensioni future di KDDML.
- Il livello *operators*, per permettere al livello GUI di capire quali operatori sono presenti nel sistema, e quali sono i parametri necessari al loro funzionamento.
- Il livello *interpreter*, per fornire al livello GUI informazioni sull'esecuzione o meta-esecuzione della query, al fine di presentare eventuali errori o informare quando l'esecuzione della query è finita, etc.

Nel corso delle ultime pagine del capitolo 5, abbiamo inoltre fatto riferimento anche alla possibile introduzione di un compilatore e di un ottimizzatore. Per entrambi bisogna decidere a che livello dell'architettura inserirli, e se integrarli o meno con i livelli già esistenti. Ad esempio, il compilatore può essere integrato direttamente nella GUI DEVELOPER, sfruttando tutte le informazioni sul sistema

KDDML a cui quest'ultima può accedere, invece l'ottimizzatore può essere integrato all'interno del livello *interpreter*.

Un'altra possibile soluzione potrebbe essere quella di vedere il compilatore e/o l'ottimizzatore come un livello intermedio tra la GUI ed il sistema. Tuttavia bisogna evidenziare che, anche in questo caso, devono essere garantite, per questi nuovi moduli, caratteristiche già discusse per l'introduzione della GUI, come ad esempio la necessità di non doverli modificare ogniqualevolta si inserisca un nuovo elemento nel sistema. In Figura 6.1 le parti tratteggiate indicano come le varie componenti possono essere integrate tra loro. A nostro avviso la prima soluzione è sicuramente quella più facilmente implementabile, senza complicare ulteriormente la struttura del sistema

Nel nostro studio abbiamo presentato i vari argomenti, evitando volutamente di entrare troppo in dettaglio sulla struttura ed il codice sorgente di KDDML, poiché sarà compito di colui che implementerà l'interfaccia confrontarsi con le scelte di implementazione presenti nel sistema attuale, e con i problemi che tali scelte possono comportare nella stesura della GUI stessa.

Come potevamo aspettarci, gli interrogativi ancora aperti sono molti, e molti di essi troveranno risposta solo nel momento di effettiva implementazione dell'interfaccia.

Ci siamo comunque potuti rendere conto che i requisiti forniti non sono così ben distinti ed indipendenti, come potrebbe apparire dalla loro presentazione nel capitolo 4. In alcuni casi tali requisiti si intersecano, come nel caso di *R. 2.0* e *R. 8.0*, mentre in altri appaiono in antitesi, come nel caso dei requisiti *R. 3.0* e *R. 3.1*. L'implementazione reale della GUI richiede quindi ulteriore studio, partendo da quanto espresso nel capitolo finale, per compiere scelte ben precise, in alcuni casi andando verso dei compromessi, in altri eliminando o ampliando il raggio di azione di requisiti ritenuti troppo restrittivi.

A nostro avviso, deve essere concepita una gerarchia di tali requisiti, che tenga in considerazione alcune proprietà ritenute importanti dal progettista, per esaltare al meglio sia caratteristiche della sola interfaccia sia dell'intero sistema. Attraverso questa gerarchia si possono fissare dei punti ritenuti fondamentali, e

successivamente si possono andare a manipolare gli altri, in modo da ottenere un buon compromesso tra tutti i requisiti ritenuti importanti.

Noi, seppur in maniera implicita, abbiamo comunque cercato di dare più importanza a taluni requisiti piuttosto che ad altri. Se in un primo tempo li abbiamo presentati tutti su uno stesso piano, poiché rappresentavano elementi importanti rilevati nell'analisi dei vari sistemi nel corso dei capitoli 3 e 4, dalle osservazioni condotte nel corso del capitolo 5, appare chiaro che esistono dei punti fermi nell'implementazione dell'interfaccia e nell'evoluzione di KDDML.

Sebbene uno studio di questo tipo non possa prescindere dall'intero elenco dei requisiti forniti, ve ne sono almeno 3 che garantirebbero a KDDML caratteristiche non riscontrate o riscontrate solo in parte nei software analizzati. Tali requisiti riguardano la suddivisione in fasi del processo, il livello GUI indipendente e per questo aggiornabile senza modifiche esplicite, e la meta-esecuzione. Il pieno supporto a tali richieste garantirebbe peculiarità che renderebbero KDDML un eccellente software, diversificandolo soprattutto da strumenti come WEKA e YALE, che rappresentano un punto di riferimento in ambito accademico.

Nello stilare la gerarchia dei requisiti, un compito importante del progettista risulta quindi quello di tenere in considerazione, e valorizzare soprattutto quei requisiti che permettano al software di diversificarsi, introducendo nuove funzionalità e strumenti non ancora previsti dagli altri, per presentare un lavoro che non sia solo una “copia” di software esistenti, ma che offra idee originali e rappresenti una strada autonoma nell'ambito del KDD.

Concludendo, possiamo affermare che l'implementazione di un linguaggio grafico su cui basare una GUI, o addirittura un intero sistema di KDD, è un passo molto complesso, che richiede notevole esperienza e sensibilità, poiché per rappresentare un processo non banale come l'estrazione di conoscenza si deve tenere conto di molteplici fattori, non solo rivolti all'estetica, ma anche alla semplicità d'uso, all'espressività ed agli strumenti di interazione che il sistema offre.

Il progettista deve quindi relazionarsi con concetti piuttosto complessi ed eterogenei, cercando di ottenere un amalgama di tutte le varie componenti descritte nel corso di questa tesi.

Ringraziamenti

Un sentito ringraziamento va a tutte le persone che mi sono state vicine nel corso di questo cammino, in particolare alla mia famiglia ed agli amici.

Desidero inoltre ringraziare il Prof. Franco Turini per l'attenzione, l'aiuto ed il tempo che mi ha dedicato, ed il Dott. Andrea Romei che è stato disponibile a discutere con me molti aspetti relativi a questo lavoro, fornendomi utili spunti di riflessione.

Un ringraziamento speciale va a Maria Ivana e lei sa perchè.

Bibliografia

- [ABKS99] M. Ankerst, M. M. Breunig, H. P. Kriegel and J. Sander. OPTICS: Ordering Points to Identify the Clustering Structure. In *Proceedings of the ACM-SIGMOD 1999 International Conference on Management of Data*, 49-60, Philadelphia, PA., 1999.
- [AGGR98] R. Agrawal, J. Gehrke, D. Gunopulos and P. Raghavan. Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications. In *Proceedings of the ACM-SIGMOD 1998 International Conference on Management of Data*, 94-105, Seattle, WA., 1998.
- [AIS93] R. Agrawal, T. Imielinski and A. Swami. Mining Association Rules between Sets of Items in Large Databases. In *Proceedings of the ACM-SIGMOD 1993 International Conference on Management of Data*, 207-216, Washington, D.C., 1993.
- [AS94] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB)*, 487- 499, Santiago del Chile, Chile, 1994.
- [AS95] R. Agrawal and R. Srikant. Mining Sequential Patterns. In *Proceedings of the 11th International Conference on Data Engineering (ICDE)*, 3-14, Taipei, Taiwan, 1995.
- [BL97] M. J. Berry and G. Linoff. *Data Mining Techniques for Marketing, Sales and Customer Support*. John Wiley & Sons, Inc., New York. 1997.
- [Cha03] C. Chatfield. *The Analysis of Time Series*. 6th edition. Chapman & Hall/CRC. Boca Raton, FL., 2003.
- [CD97] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *ACM-SIGMOD Record*, 26 (1): 65-74, 1997.
- [DELPHI] Delphi Borland.
(<http://www.borland.com/us/products/delphi/index.html>) © Borland Software Corporation. 1994-2005.
- [DMG] The Data Mining Group. (<http://www.dmg.org/pmml-v2-0.html>). 2003.

- [DZ04] J. Demsar and B. Zupan. Orange: From Experimental Machine Learning to Interactive Data Mining. White Paper. (<http://www.ailab.si/orange/doc>), Faculty of Computer and Information Science, University of Ljubljana, Slovenia, 2004.
- [EK SX96] M. Ester, H. P. Kriegel, J. Sander and X. Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining*, (KDD'96), 226-231, Portland, OR., 1996.
- [FCK94] K. H. Fasman, A. J. Cuticchia, and D. T. Kingsbury. The GDB Human Genome Data Base. *Nucleic Acids Research*, 22 (17): 3462-3469, 1994.
- [FGM] Fujitsu GhostMiner.
(http://www.fqspl.com.pl/?a=product_view&id=2&lang=en&x=) © FQS Poland, 2002-2005.
- [Fis36] R. A. Fisher. The Use of Multiple Measurements in Taxonomic Problems. *Annual Eugenics*, 7: 179-188, 1936. Also in *Contributions to Mathematical Statistics*. John Wiley & Sons, Inc., New York, 1950.
- [Fis87] D. H. Fisher. Improving Inference through Conceptual Clustering. In *Proceedings of AAAI-87 Sixth National Conference on Artificial Intelligence*, 461-465, Seattle, WA., 1987.
- [FPSSU96] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth and R. Uthurusamy. *Advances in Knowledge Discovery and Data Mining*. AAAI Press / The MIT Press, Cambridge, MA., 1996.
- [GLF89] J. H. Gennari, P. Langley and D. Fisher. Models of Incremental Concept Formation. *Journal of Artificial Intelligence*, 40: 11-61, 1989.
- [GRS98] S. Guha, R. Rastogi and K. Shim. CURE: An Efficient Clustering Algorithm for Large Databases. In *Proceedings of the ACM-SIGMOD 1998 International Conference on Management of Data*, 73-84, Seattle, WA., 1998.
- [Har01] E. R. Harold. *XML Bible*. 2nd edition. John Wiley & Sons, Inc. New York, 2001
- [Hed96] S. R. Hedberg. Searching for the Mother Lode: Tales of the First Data Miners. *IEEE Expert: Intelligent Systems and Their Applications*, 11, (5):4-7, 1996.

-
- [HF95] J. Han and Y. Fu. Discovery of Multiple-level Association Rules from Large Databases. In *Proceedings of the 21st International Conference on Very Large Databases (VLDB)*, 420-431, Zurich, Switzerland, 1995.
- [HK98] A. Hinneburg and D. A. Keim. An Efficient Approach to Clustering in Large Multimedia Databases with Noise. In *Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining (KDD'98)*, 58-65, New York, 1998.
- [HK00] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann. San Francisco, CA., 2000.
- [IM96] T. Imielinski and H. Mannila. A Database Perspective on Knowledge Discovery. *Communications of the ACM*, 39 (11): 58-64, 1996.
- [KDN] KDnuggets. Software Suites for Data Mining and Knowledge Discovery. (<http://www.kdnuggets.com/software/suites.html>). © KDnuggets. 2005.
- [KR90] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley & Sons, Inc. New York, 1990.
- [Mac67] J. MacQueen. Some Methods for Classification and Analysis of Multivariate Observations. In *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, 281-297, Berkeley, CA., 1967.
- [MPC96] R. Meo, G. Psaila and S. Ceri. A New SQL-like Operator for Mining Association Rules. In *Proceedings of 22nd International Conference on Very Large Databases (VLDB)*, 122-133, Mumbai (Bombay), India, 1996.
- [ORANGE] ORANGE. Data Mining Fruitful & Fun. (<http://www.aillab.si/orange>)
- [ORS98] B. Özden, S. Ramaswamy and A. Silberschatz. Cyclic Association Rules. In *Proceedings of the 14th International Conference on Data Engineering (ICDE)*, 412-421, Orlando, FL., 1998.
- [PCY95] J. S. Park, M. S. Chen, and P. S. Yu. An Effective Hash-based Algorithm for Mining Association Rules. In *Proceedings of the ACM-SIGMOD 1995 International Conference on Management of Data*, 175-186, San José, CA., 1995.

- [PERL] PERL. O'Reilly perl.com. The Source for Perl. (<http://www.perl.com/>) © O'Reilly Media, Inc. All Rights Reserved. 2000-2005.
- [PYTHON] PYTHON. (<http://www.python.org>) © Python Software Foundation. 2005.
- [Qui86] J. R. Quinlan. Induction of Decision Trees. *Machine Learning*, 1 (1): 81-106, 1986.
- [Qui93] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann. San Francisco, CA., 1993.
- [RRT05] A. Romei, S. Ruggeri and F. Turini. KDDML: a Middleware Language and System for Knowledge Discovery in Databases. In *Proceedings of the 13th Italian Symposium on Advanced Database Systems (SEBD 2005)*, 208-219, Brixen-Bressanone, 2005.
- [SCZ98] G. Sheikholeslami, S. Chatterjee and A. Zhang. WaveCluster: A Multi-Resolution Clustering Approach for Very Large Spatial Databases. In *Proceedings of the 24th International Conference on Very Large Databases (VLDB)*, 428-439, New York, 1998.
- [SON95] A. Savasere, E. Omiecinski and S. B. Navathe. An Efficient Algorithm for Mining Association Rules in Large Databases. In *Proceedings of the 21st International Conference on Very Large Databases (VLDB)*, 432-444, Zurich, Switzerland, 1995.
- [Str97] B. Stroustrup. *The C++ Programming Language*. 3rd edition. Addison Wesley Longman. Reading, MA., 1997.
- [SUN] Sun Developer Network. (<http://java.sun.com/>). © Sun Microsystems, Inc. 1994-2005
- [TAN] TANAGRA. A Free Data Mining Software for Research and Education. (<http://eric.univ-lyon2.fr/~ricco/tanagra/>). 2005.
- [TCL] Tcl Developer Xchange (<http://www.tcl.tk/>).
- [TQT] Trolltech QT. (<http://www.trolltech.com/>). © Trolltech®. 2005
- [ZDKO+01] B. Zupan, J. Demsar, M. W. Kattan, M. Ogori, M. Graefen, M. Bohanec and J. R. Beck. Orange and Decisions-at Hand: Bridging Predictive Data Mining and Decision Support In C. Giraud-Carrier, N. Lavrac and Steve Moyle (eds.) *ECML/PKDD'01 Workshop on Integrating Aspects of Data Mining, Decision Support and Meta-Learning*, 151-162, Freiburg, Germany, 2001.

-
- [ZLDC04] B. Zupan, G. Leban, J. Demsar and T. Curk. Widgets and Visual Programming. White Paper. (<http://www.ailab.si/orange/doc>), Faculty of Computer and Information Science, University of Ljubljana, Slovenia, 2004.
- [ZRL96] T. Zhang, R. Ramakrishnan and M. Livny. BIRCH: An Efficient Data Clustering Method for Very Large Databases. In *Proceedings of the ACM-SIGMOD 1996 International Conference on Management of Data*, 103-114, Montreal, Canada, 1996.
- [YALE] YALE. Yet Another Learning Environment. (<http://www-ai.cs.uni-dortmund.de/SOFTWARE/YALE/index.html>)
- [W3CDOM] Document Object Model (DOM). (<http://www.w3.org/DOM/>) © W3C. 1994-2005.
- [W3CXSL] eXtensible Stylesheet Language (XSL). (<http://www.w3.org/Style/XSL/>) © W3C. 1994-2005.
- [W3CXML] eXtensible Markup Language (XML). (<http://www.w3.org/XML/>) © W3C. 1996-2003.
- [WB98] C. Westphal and T. Blaxton. *Data Mining Solutions. Methods and Tools for Solving Real-World Problems*. John Wiley & Sons, Inc. New York, 1998.
- [WEKA] Weka 3: Data Mining Software in Java (<http://www.cs.waikato.ac.nz/ml/weka/>)
- [WF00] I. H. Witten and E. Frank. Nuts and Bolts: Machine Learning Algorithms in Java. In *Data Mining: Practical Learning Tools and Techniques with Java Implementations*, 265-320, Morgan Kaufman. San Francisco, CA., 2000.
- [WPV] Weka PMML and Visualization. (<http://www.cs.bris.ac.uk/home/jl2092/index.html>). Jiwen Li.
- [WS91] J. Way and E. A. Smith. The Evolution of Synthetic Aperture Radar Systems and their Progression to the EOS SAR. *IEEE Transactions on Geoscience and Remote Sensing*, 29 (6): 962-985, 1991.
- [WYM97] W. Wang, J. Yang and R. R. Muntz. STING: A Statistical Information Grid Approach to Spatial Data Mining. In *Proceedings of the 23rd International Conference on Very Large Databases (VLDB)*, 186-195, Athens, Greece, 1997.