Previsione di Sintomatologie Post-Dialisi

Studio e realizzazione di un sistema supervisionato di estrazione regole per la sintomatologia post-dialitica

Francesco Pontillo (mat. 600119)
Universitá degli Studi di Bari
Dipartimento di Informatica
Via E. Orabona, 4 - 70125 Bari, Italy
francescopontillo@gmail.com

ABSTRACT

Si presenta un'implementazione base di C4.5, algoritmo di apprendimento di alberi di decisione, in Prolog, con *k-fold cross-validation* test e generazione di regole di decisione. L'algoritmo viene quindi confrontato con Weka, un *tool* per il data mining, e con BayesDB, un sistema di classificazione bayesiana.

1. OBIETTIVO

Obiettivo del processo di Data Mining del sistema da sviluppare é di **prevedere possibili sintomatologie successive ad una seduta di emodialisi**. A partire da specifici dati registrati durante una dialisi, si vuole prevedere quale classe di sintomatologia il paziente potrá riscontrare dal momento in cui la dialisi termina al momento in cui esegue la seduta di dialisi successiva.

In questo modo, il medico puó confermare la possibilitá di occorrenza di una o piú problematiche suggerite, ed eventualmente prescrivere una opportuna terapia per contrastare la sua insorgenza.

2. SELEZIONE DEGLI ATTRIBUTI

I dati a disposizione nella base di dati da analizzare sono numerosi, e devono essere selezionati appropriatamente per evitare l'introduzione di attributi poco rilevanti con lo scopo del sistema.

Ogni seduta di dialisi memorizza (1) una **data** di svolgimento, (2) la **durata** della seduta stessa, (3) un identificativo del **paziente**, (4) altri **parametri** registrati durante la sessione e (5) eventuali **sintomatologie** riscontrate.

2.1 Dati del paziente

Le informazioni relative ai pazienti sono ricavate dalla base di dati originale. Ai fini del processo di estrazione delle regole, é opportuno considerare il **sesso** del paziente e la sua **etá** al momento della seduta di dialisi in analisi[1].

2.2 Parametri della seduta di dialisi

I parametri piú rilevanti di una seduta di dialisi, al fine di prevedere eventuali sintomatologie successive, sono divisi in piú categorie [4] [5].

A. L'**efficienza della rimozione dei prodotti di scarto** é indotta dai valori dei parametri riportati in Tabella 1.

KT/V	indice di efficienza dialitica
QB	flusso di sangue trattato
WS	peso iniziale
WE	peso finale
PWE	peso finale ottimale
PT	durata ottimale
T	durata reale

Table 1: Parametri di efficienza eliminazione scarti

B. L'efficienza dell'eliminazione dell'acqua all'interno del corpo del paziente é indotta dai parametri in Tabella 2.

SPS	pressione sistolica iniziale
SPE	pressione sistolica finale
DPS	pressione diastolica iniziale
DPE	pressione diastolica finale
BV	volume ematico finale

Table 2: Parametri di efficienza eliminazione acqua

C. Altre tipologie di dati che potrebbero risultare utili a fornire previsioni significative sono riportati in Tabella 3.

PBF	flusso sangue teorico
BF	flusso sangue reale
PUF	ultrafiltrazione media teorica
UF	ultrafiltrazione media reale

Table 3: Altri parametri di efficienza dialitica

[1] Ció non esclude la possibilitá di considerare altri dati relativi al paziente; l'algoritmo da realizzare potrebbe essere esteso andando a considerare anche i dati relativi alle malattie pregresse ed eventuali comorbiditá registrate.

2.3 Attributi derivati

A partire dalle informazioni disponibili nella base di dati, risulta evidente la presenza di alcuni attributi "nascosti" che possono essere più utili ai fini dell'apprendimento.

In tabella 4 sono elencati gli attributi derivati dalle precedenti tabelle; ad esempio, ΔWL rappresenta la differenza fra la perdita di peso programmata e quella effettiva, a sua volta calcolata come differenza fra peso iniziale e peso finale.

PWL	perdita peso programmata
RWL	perdita peso reale
ΔWL	differenza perdita peso
ΔT	differenza durata trattamento
SPA	pressione sistolica media
DPA	pressione diastolica media
ΔBF	differenza flusso sangue
ΔUF	differenza UF medio

Table 4: Parametri derivati

2.4 Sintomatologie

Il sistema verrá addestrato con istanze di esempio pre-classificate. La classificazione consiste nell'assegnazione, ad ogni esempio, di una categoria di sintomatologia, ad esempio: aritmia sintomatica, aritmia asintomatica, astenia, brividi, brividi e dispnea, cefalea, collasso (PA $< 30\%$ inizio), conati di vomito, crampi, depressione, ansia, diarrea, dispnea e molti altri.

Inoltre, é prevista la classe "asintomatico", che definisce una sintomatologia assente, corrispondente ad un esempio negativo dal punto di vista della classificazione.

3. SELEZIONE DEI DATI

Le informazioni sottoposte all'algoritmo di apprendimento sono state selezionate a partire da una base dati molto ricca[2] e sono state sottoposte ad una serie di passaggi[3].

3.1 Creazione dei valori derivati

Per poter istanziare i valori degli attributi definiti in 2.3, é stata eseguita una query di tipo `SELECT` che preleva informazioni dalla tabella di origine ed effettua semplici calcoli di trasformazione.

In questo modo, alla fine del processo di trasformazione, gli attributi per ogni seduta di dialisi sono:

- `PATIENT`, l'identificativo del paziente
- `SESSION_ID`, l'ID della seduta di dialisi, utile per identificare la seduta in ogni momento
- `SESSION_DATE`, la data di esecuzione, non utilizzata ai fini dell'apprendimento
- `KTV`, il valore di KT/V
- `QB`, il valore di QB
- `PROG_WEIGHT_LOSS`, la perdita peso programmata
- `REAL_WEIGHT_LOSS`, la perdita peso reale
- `DELTA_WEIGHT`, la differenza fra la perdita di peso reale e quella programmata
- `PROG_DURATION`, la durata programmata della dialisi
- `REAL_DURATION`, la durata effettiva della dialisi
- `DELTA_DURATION`, la differenza fra la durata reale e quella programmata
- `SAP_START`, la pressione sistolica arteriosa prima della seduta
- `SAP_END`, la pressione sistolica arteriosa dopo la seduta
- `AVG_SAP`, la pressione sistolica arteriosa media
- `DAP_START`, la pressione diastolica arteriosa prima della seduta
- `DAP_END`, la pressione diastolica arteriosa dopo la seduta
- `AVG_DAP`, la pressione diastolica arteriosa media
- `BLOOD_VOLUME`, il volume di sangue trattato
- `DELTA_BLOOD_FLOW`, la differenza fra flusso di sangue teorico ed effettivo
- `DELTA_UF`, la differenza dell'ultrafiltrazione media reale e teorica

Come si nota, sono stati eliminati alcuni attributi originali: il flusso di sangue teorico e reale e l'ultrafiltrazione media teorica e reale.

3.2 Associazione con sintomatologie

Nel programma che genera i dati, le sintomatologie vengono comunicate e quindi inserite, dal medico o dall'infermiere, qualche momento prima della dialisi successiva del paziente. Per poter mettere a confronto i dati della seduta di dialisi (di cui sopra) con i dati della sintomatologia rilevata, é stato necessario eseguire una query molto complessa per mettere in correlazione:

- il paziente
- la data di dialisi
- la data di dialisi minore fra quelle successive alla data di riferimento della seduta originaria

3.3 Associazione con dati del paziente

Infine, il dato della sintomatologia è stato associato univocamente con il paziente di riferimento, tramite l'apposito identificativo.

Tutte queste operazioni sono state eseguite staticamente, ovvero andando a creare una copia dei record in altre tabelle; ció si é reso necessario in quanto, anche utilizzando macchine potenti, la selezione completa dei record impiegava interi minuti per completare, soprattutto a causa dell'associazione poco ottimizzata con le date (cfr. 3.2).

[2]Circa dal 1999 ai primi mesi del 2014.

[3]Tutte le trasformazioni e selezioni di dati descritte in questa sezione sono codificate nel `scripts/01-sql-server-tables.sql`.

3.4 Migrazione dei dati

Per una gestione piú libera dei dati, si é scelto di migrare le tabelle create da Microsoft SQL Server a MySQL[4], anche in ottica futura (cfr. 9).

4. PULIZIA DEI DATI

Una volta spostati i dati su un database MySQL, si é scelto di eliminare alcuni record e mantenerne altri piú rilevanti[5]. La base dati originaria, infatti, contiene 185476 record.

4.1 Calcolo dello score

Ad ogni riga di rilevazione sintomo é stato associato un punteggio, o *score*, che permetta di capire quanto quella riga é completa (e quindi piú o meno rilevante rispetto alle altre).

Fissato il numero degli attributi a 15, un record con *score* piú elevato sará selezionato con piú probabilitá per avviare il processo di apprendimento.

4.2 Pazienti rilevanti

Inoltre, lo *score* é stato utilizzato anche per poter eliminare, dai record giá selezionati, tutti quelli che appartengono a pazienti che hanno meno di 5 rilevazioni di sintomi con uno *score* percentuale piú basso dell'80%.

Tutti i dati selezionati fino a questo punto, quindi, appartengono a pazienti che hanno **almeno** 5 **rilevazioni di sintomi ottimali**.

4.3 Gestione dei valori nulli

I valori nulli sono stati gestiti "staticamente", ovvero per ogni paziente sono state calcolate le medie dei valori di ogni attributo (ignorando i valori nulli); in un passo successivo, sono stati scansionati tutti i record e, qualora fosse rilevato un valore nullo, é stato inserito il valore medio relativo al paziente associato.

Tutto ció, tuttavia, ha portato comunque a mantenere alcuni valori nulli all'interno della base dati. Ad esempio, poche rilevazioni di sintomatologia contengono valori effettivi di `KTV`, probabilmente perché si tratta di una misura di difficile calcolo da parte dei medici. Tali valori sono stati gestiti diversamente (cfr. 7.5.2).

Alcuni record, inoltre, non contenevano l'ID del sintomo target rilevato, e si é pertanto assunto che l'utente avesse erroneamente cancellato (dall'interfaccia del sistema) la dicitura "asintomatico"; é stata pertanto aggiunta una sintomatologia nulla (ID uguale a 1).

5. APPRENDIMENTO DI REGOLE

L'obiettivo del sistema é l'apprendimento di regole utilizzabili per fare previsioni significative. Avendo a disposizione una moltitudine di dati pre-classificati, risulta immediato pensare ad un approccio guidato che generi un albero di decisione.

5.1 Alberi di decisione

Nei sistemi TDIDT, il concetto é rappresentato in termini di un albero di decisione costruito in modalitá top-down con una tecnica model driven: il sistema seleziona un attributo che meglio rappresenta (in qualche forma) tutti gli esempi, quindi procede ricorsivamente fino a raggiungere la copertura totale.

Questi algoritmi distinguono tra uno spazio degli esempi e uno spazio delle ipotesi, riuscendo a ridurre di molto la complessità di ricerca nello spazio delle ipotesi.

Siano dati:

- S_0 insieme di oggetti
- C insieme di classi
- $\mathbb{A}$ insieme di attributi
- $\Lambda_A = \{a_1, \cdots, a_r\}$ valori discreti che un attributo $A \in \mathbb{A}$ può assumere

La costruzione dell'albero di decisione equivale a trovare un albero T che classifichi correttamente tutti gli oggetti in S_0.

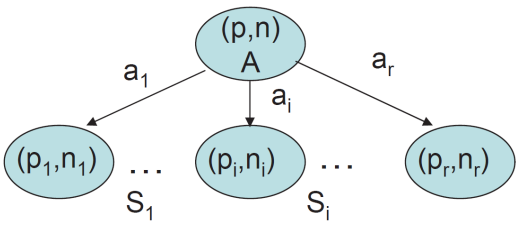

Figure 1: Esemplificazione di un albero di decisione

Con riferimento alla figura 1, si nota che:

- $S_i \cap S_j = \emptyset, \forall i \neq j$, cioè ogni sotto-albero S_i è disgiunto dagli altri S_j adiacenti (nello stesso livello)
- $S_i = \{e \in S | A(e) = a_i\}$, per ogni esempio e appartenente ad un sotto-albero S_i, l'attributo $A(e)$ assume sempre lo stesso valore a_i (l'attributo selezionato per caratterizzare il sotto-albero S_i ha valore costante in tutti i sotto-alberi figli)

5.2 Costruzione dell'albero di decisione

A partire da esempi pre-classificati in fase di training, il sistema genera un albero di decisione in cui S_0 è la radice. Ad ogni nodo σ viene selezionato il **miglior attributo** A^*, in accordo a qualche criterio (cfr. 5.3), per effettuare il test a quel nodo. Infine, si assegna il nome di una classe ad ogni nodo foglia.

L'albero può anche essere validato su un insieme di testing: si segue il cammino dalla radice ad una foglia testando ad ogni nodo il valore dell'attributo selezionato, e procedendo il cammino sul nodo con il valore (o range di valori) dell'osservazione da classificare.

[4]Lo script di migrazione é presente in `scripts/mysql-migration/02-mysql-migration-script.sql` e viene richiamato in automatico, tramite appositi parametri di connessione, dal file batch `scripts/mysql-migration/03-mysql-copy-migrated-tables.cmd`.

[5]Gli script rilevanti sono contenuti nel file `scripts/mysql-migration/04-mysql-scores.sql`.

5.3 Scelta dell'attributo più discriminante

Esistono diversi criteri euristici per scegliere l'attributo più discriminante ad ogni livello, ognuno dei quali utilizza una misura specifica per massimizzare l'effetto discriminante:

- Massimizzazione dell'**informazione**
  - entropia minima
  - guadagno di informazione normalizzato
  - rapporto di guadagno
  - riduzione della lunghezza della descrizione
- Minimizzazione dell'**errore**
  - riduzione dell'errore nel training set
  - dissimilarità
  - indice di diversità di Gini
- Massimizzazione della **significatività**, basato su statistiche varie (χ^2, G, $\cdots$)

5.3.1 *Entropia*

L'entropia definisce la ***impurity* di un insieme** arbitrario di dati S, contenente esempi positivi (in proporzione p_+) e negativi (in proporzione p_-):

$$entropia(S) = -p_+ \log_2 p_+ - p_- \log_2 p_-$$

In questo modo il valore dell'entropia agli estremi è:

- 0 se c'è **minima entropia**, ovvero se tutti gli esempi sono positivi o negativi
- 1 se c'è **massima entropia**, ovvero se gli esempi sono positivi e negativi in ugual numero

La funzione entropia relativa ad una classificazione booleana varia, quindi, come la proporzione p_+, cioè fra 0 e 1.

Più in generale, se il concetto target può assumere c valori anziché 2, la funzione entropia diventa:

$$entropia(S) = \sum_{i=1}^{c} -p_i \log_2 p_i$$

dove p_i è la proporzione di esempi in S appartenenti alla classe i; In questo senso, la categorizzazione booleana è una sua specializzazione a due classi.

5.3.2 *Information Gain*

L'information gain di un attributo A relativo ad un insieme di esempi S è così calcolato:

$$Gain(S, A) = entropia(S) - \left[\sum_{a \in \Lambda_A} \left(\frac{|S_a|}{|S|} \times entropia(S_a) \right) \right]$$

dove S_a è il sottoinsieme di S per il quale l'attributo A ha valore a.

$Gain(S, A)$ rappresenta la **riduzione attesa in entropia** (disordine), causata dal conoscere il valore dell'attributo A. Maggiore è la riduzione dell'entropia, maggiore è il guadagno che si ottiene selezionando A come attributo discriminante.

5.4 Considerazioni sui sistemi TDIDT

É possibile passare da alberi di decisione a regole di decisione, semplicemente analizzando ogni cammino dai nodi foglia al nodo radice e generando, per ognuno, una regola in cui la scelta di uno specifico arco introduce un nuovo congiunto.

I sistemi TDIDT sono molto efficienti, e gli alberi di decisione possono trattare sia attributi a valori categorici, gestendo più archi uscenti da ogni nodo, che continui, tramite definizione di intervalli che vanno a configurarsi come categorie.

Essi sono inoltre non parametrici e non incrementali, in quanto l'introduzione di un nuovo esempio potrebbe rovinare la rigida struttura ad albero, e quindi il suo potere inferenziale.
Inoltre, nel caso di dati rumorosi, potrebbero essere generati alberi di dimensioni enormi, che andrebbero quindi gestiti con opportuni algoritmi di pruning.

Altri metodi, infine, rimpiazzano i nodi foglia con distribuzioni di probabilità della classe inferita, soprattutto con numerosi esempi di apprendimento.

5.5 C4.5

Per apprendere regole utili a classificare appositamente una seduta di dialisi si é scelto di implementare il funzionamento base dell'algoritmo C4.5 di Ross Quinlan [7] [8], estensione del precedente ID3 [6]. In questo modo risulta possibile:

- utilizzare la sintomatologia come attributo target (da classificare)
- poter sfruttare i numerosi dati disponibili
- generare un albero di decisione
- convertire l'albero in un insieme di regole Prolog

L'implementazione del C4.5 é stata descritta, concettualmente e nella sua implementazione, in 7.5.2.

6. TEST DELL'ALGORITMO

Gli esempi di rilevazioni di sintomatologie a disposizione sono, purtroppo, mal distribuiti: alcune sintomatologie sono molto presenti, mentre altre sono molto rare. Per questo motivo, e per valutare la stabilitá del sistema, si é ritenuto opportuno validare le regole generate.

6.1 Test su algoritmi di IA

Il processo di apprendimento per gli schemi più comuni opera in più fasi:

1. Costruzione della struttura base a partire dai **training data**
2. Ottimizzazione delle impostazioni sui parametri, eseguita con i **validation data**
3. Test del sistema ottenuto, tramite i **test data**

Ovviamente training, validation e test data sono insiemi di dati completamente indipendenti.

A valutazione completa, infine, tutti i dati rilevati devono essere utilizzati per ricostruire un nuovo classificatore per l'utilizzo effettivo.

Non potendo disporre di un dataset molto stabile[6], tuttavia, si é dovuto ricorrere ad altre tecniche per realizzare un classificatore che possa essere ritenuto valido.

6.2 K-Fold Cross-Validation

La **cross-validation** é una metodologia di test che divide l'insieme dei dati in:

- **training set**, di solito $\frac{9}{10}$ dei dati originali
- **testing set**, di solito $\frac{1}{10}$ dei dati originali

Operando in questa maniera, l'insieme dei dati di test é sempre differente dall'insieme dei dati di *training*. In generale, il processo:

1. Divide i dati in k sotto-insiemi di dimensioni uguali.
2. Utilizza, a turno, ogni sotto-insieme per il testing, ed i rimanenti per il training.
3. Esegue, alla fine dei k passi, una media delle stime di errore per ottenere una **stima di errore finale**.

Poiché la cross-validation partiziona l'insieme degli esempi in k sotto-insiemi, è chiamata anche **k-fold cross-validation**.

Il metodo standard per la valutazione è il **10-fold cross-validation**, che divide lo spazio degli esempi in 10 sotto-insiemi, in quanto sperimentazioni estensive (ma anche dimostrazioni teoriche) hanno rilevato che 10 è la scelta migliore di partizionamento per poter fornire una stima accurata.

7. IMPLEMENTAZIONE IN PROLOG

Il programma Prolog é diviso in 6 file, ognuno dei quali si occupa di parti differenti del programma[7].

Si é scelto di utilizzare SWI-Prolog[2] come ambiente Prolog.

7.1 Utility

Sono stati realizzati 2 file che realizzano funzioni di utilitá.

[6]Al momento é in corso un procedimento per raccogliere dati da piú centri di dialisi in tutta Italia, in modo tale da disporre di un insieme di esempi piú ricco e fare meno ricorso a filling-in di dati nulli. Sarebbe inoltre utile rilevare dati per razze diverse da quella caucasica, che copre l'intero *set* di esempi, rendendo l'analisi di tale attributo praticamente inutile.

[7]Ogni regola definita nei diversi file é stata documentata. La documentazione é consultabile aprendo in un browser il file `doc/index.html`. Per un problema nel modulo di generazione della documentazione, i link diretti dalla `index.html` alle regole documentate non funzionano; utilizzare, invece, i collegamenti ai file, dai quali si puó comunque accedere alla documentazione delle regole. Alternativamente, si puó consultare la documentazione in PDF, nel percorso `prolog/doc/doc_full.pdf` o la stampa allegata.

7.1.1 `util.pl`

`util.pl` contiene brevi regole che implementano:

- **timer generico**, con avvio, lettura e stop (`timer_start`, `timer_get`, `timer_stop`, tra gli altri)
- formattazione di millisecondi (`format_ms`) e secondi (`format_s`) nel formato intellegibile `{M}m {S}s {MS}ms`
- **stampa a video** di generiche liste di elementi o di un elemento singolo, con ritorno a capo (`println`)
- generici *helper* per **liste**, che realizzano funzioni di minimo e massimo (`list_min`, `list_max`), ricerca dell'elemento piú comune (`list_most_common`) e dell'indice di un elemento specifico (`index_of`), oltre che funzionalitá di aggi e rimozione di elementi
- concatenazione di elementi di una lista in una stringa
- realizzazione del logaritmo in base 2 (`log2`), utilizzato successivamente (cfr. 7.5).

7.1.2 `log.pl`

`log.pl` implementa un sistema di *logging* simile a quello di `adb` per Android, ossia con piú livelli e con output colorato. La necessitá di dover filtrare e visualizzare meglio la stampa a video deriva dai numerosi messaggi che ogni regola Prolog realizzata stampa a fini di *debug*.

I livelli gestiti sono 6, in ordine di prioritá dalla piú bassa alla piú alta:

- `verbose`, il livello minimo, da usare per mostrare i messaggi meno utili a fini di debug ma che possono risultare necessari per rivedere calcoli e punti di ingresso/uscita
- `debug`, genericamente usato per mostrare informazioni di *debug* di una certa rilevanza
- `info`, di solito mostra messaggi informativi a livello utente; rappresenta il livello standard meno verboso, e non deve perció essere sovraccaricato di log inutile
- `warn`, livello che mostra gli avvisi, o condizioni inattese ma comunque gestite (esempio: "file non trovato, utilizzo i parametri standard")
- `error`, mostra messaggi di errore relativamente a qualcosa che non sarebbe dovuta accadere (esempio: "file non trovato, impossibile continuare")
- `assert`, anche detto livello "wtf", mostra errori catastrofici dai quali é impossibile recuperare

La classe di predicati `log_X` (dove `X` rappresenta uno dei livelli precedenti) scrive il messaggio di log insieme ad un eventuale *tag* (se specificato), che definisce una sorta di "titolo" del log; generalmente il *tag* é stato impostato al nome della regola che eseguiva la stampa del log. Se il tag é troppo lungo, inoltre, viene tagliato e vengono aggiunti dei punti sospensivi.

Il sistema mette a disposizione la regola `log_level/1` che imposta (o restituisce se il parametro non é istanziato) il

minimo livello di logging che verrá effettivamente stampato da quel momento in poi; é la regola `can_log/1` che controlla se un certo livello puó essere mostrato o meno.
Il codice del livello, infine, viene formattato con il colore associato (ad esempio, verde per `I`, giallo per `W`).

7.2 Avvio del programma

Il programma principale é definito nel file `main.pl`, che si occupa del caricamento in memoria di tutti i file Prolog necessari e di definire il metodo `main(Config, Symptom)`:

- `Config` dichiara al programma qual é il file di configurazione con il quale si vuole accedere al database.[8] (vedi 7.3 per l'accesso al database). Si é optato per un file di configurazione che contenesse tutti i parametri di connessione poiché la scrittura degli stessi ad ogni avvio del programma sarebbe risultata troppo verbosa.
- `Symptom` definisce l'ID del sintomo che si vuole utilizzare come esempio positivo per l'attributo target.

Entrambi i parametri supportano i meta-valori `default` e `ask`: `default` esegue un *fallback* dei parametri sul file di configurazione `prolog/config/database.properties` (si veda A per l'importazione dei dati) e sul sintomo con ID 2, mentre `ask` imposta il programma in modo da chiedere all'utente, in maniera interattiva e quando necessario, gli stessi parametri. La figura 2 mostra l'avvio dell'algoritmo con il file di configurazione di default e per il sintomo 8.

Sono anche presenti funzioni di avvio rapido: `main_def/0` (che avvia il programma con parametri di default), `main/0` (che avvia in modalitá `ask`) e `make_doc/0` (che genera la documentazione HTML e LaTeX).

Il sistema di *logging* implementato é, di default, molto verboso. Se non si vuole avere in output il dettaglio completo di ció che sta avvenendo nel programma, é possibile utilizzare le regole messe a disposizione dal modulo di *logging* (vedi 7.1.2) per settare un livello di *logging* meno verboso.

Per uscire dal programma e chiudere in maniera pulita la connessione, basta chiamare la regola `out/0`.

7.3 Lettura dal database

Una volta letti i parametri impostati dal `main` vengono eseguiti i processi di connessione e lettura dei dati utili alla generazione dell'albero di decisione.

La lettura dal database é possibile grazie alla libreria `odbc` di SWI-Prolog [9].

7.3.1 Connessione

[8] Alcuni file di configurazione di esempio sono presenti nella cartella `prolog/config`.
[9] Per questa ragione, é necessario che sulla macchina client siano installati i driver di connessione ODBC al database target. Poiché i parametri sono impostati dal programma Prolog, non é necessario creare nessuna connessione sulla macchina.

```
%  database.pl compiled 0.02 sec, 35 clauses
%  categories.pl compiled 0.00 sec, 33 clauses
%  util.pl compiled 0.00 sec, 42 clauses
%  learner.pl compiled 0.02 sec, 45 clauses
%  log.pl compiled 0.00 sec, 28 clauses
%    library(error) compiled into error 0.00 sec, 81 clauses
%    library(pairs) compiled into pairs 0.00 sec, 22 clauses
%   library(lists) compiled into lists 0.00 sec, 205 clauses
%   library(assoc) compiled into assoc 0.00 sec, 103 clauses
%  library(dialect/hprolog) compiled into hprolog 0.00 sec, 356 clauses
Welcome!
% main.pl compiled 0.03 sec, 565 clauses
connect       D  Using default connection configuration.
connect       D  Using a configuration file.
database      V  Reading database parameters...
database      I  Database parameters read.
database      I  Connected to dialysis_connection
database      V  Fetching symptoms...
database      I  61 symptoms fetched in 0m 0s 1ms.
database      V  Fetching records for symptom 1...
database      V  Preparing statement...
database      V  Statement prepared.
database      I  100 records fetched in 0m 2s 839ms.
database      V  Fetching records for symptom 2...
database      V  Preparing statement...
database      V  Statement prepared.
database      I  19 records fetched in 0m 2s 341ms.
categories    V  Updating categories...
categories    V  Making class PatientSex
categories    V  Making class PatientRace
categories    V  Making class PatientAge
categories    V  Making class ProgWeightLoss
categories    V  Making class RealWeightLoss
categories    V  Making class DeltaWeight
categories    V  Making class ProgDuration
categories    V  Making class RealDuration
categories    V  Making class DeltaDuration
categories    V  Making class SAPStart
categories    V  Making class SAPEnd
categories    V  Making class SAPAverage
categories    V  Making class DAPStart
categories    V  Making class DAPEnd
categories    V  Making class DAPAverage
categories    V  Making class BloodVolume
categories    V  Making class DeltaBloodFlow
categories    V  Making class DeltaUF
categories    V  Making class SymptomID
categories    I  Categories updated.
split_ex      D  Splitting examples with 10 folds and test fold = 10
split_ex      D  Positive test examples for this run are [11691,171311]
```

Figure 2: Avvio del programma per il sintomo 2

La connessione al database avviene tramite la regola `connect` (nelle varianti) e i parametri nel file di configurazione impostato in precedenza: driver ODBC, indirizzo e porta, username, password e database. Se si é in modalitá `ask`, il file di configurazione verrá chiesto all'utente, e nel caso in cui non esista viene eseguito un *fallback* sul file di default.

La lettura del file `.properties` é eseguita da

```
read_database_params(Path, Driver, Server,
 Port, Database, User, Password)
```

che utilizza il costrutto `open_table` di SWI-Prolog per leggere i campi del file di proprietá e unificare con le variabili passate in input.

7.3.2 Lettura sintomi

Il passo successivo consiste nella lettura di tutte le possibili sintomatologie che potrebbero verificarsi nel corso di una seduta di dialisi. La regola `get_symptoms/0` esegue una

semplice `SELECT` sulla base di dati, ottenendo e salvando in memoria i sintomi.

7.3.3 Lettura degli esempi

La regola `update_records/0` si occupa di ottenere e salvare in memoria tutti gli esempi che devono essere utilizzati per avviare il processo di apprendimento.

Poiché é necessaria la selezione sia degli esempi positivi che di quelli negativi, `update_records/0` esegue lo stesso *statement* di `SELECT`, opportunamente creato e preparato, andando a modificare l'ID della sintomatologia target da ottenere[10]. Per lo scopo del progetto é stato imposto un limite di 100 esempi positivi e 100 esempi negativi, in modo tale da velocizzare il processo di generazione delle regole.

Il salvataggio dei record avviene andando ad asserire, nella memoria del programma Prolog, strutture del tipo:

```
positive(ID, Attribute, Value)
negative(ID, Attribute, Value)
```

In questo modo é sempre possibile accedere a qualsiasi coppia attributo-valore di un esempio con uno specifico ID, sia esso positivo o negativo.

É stata realizzata anche la regola:

```
example(Type, ID, Attribute, Value)
```

dove `Type` puó essere `positive` o `negative`. Tramite questa modalitá é possibile accedere a tutti gli esempi prelevati dalla base di dati. Sono presenti, inoltre, regole di conteggio e di verifica di esistenza (cfr. la documentazione Prolog in HTML, LaTeXo la stampa allegata).

7.4 Suddivisione attributi in range

Gli attributi dei dati di esempio possono essere numerici (a virgola mobile) o categorici, ma in ogni caso sono identificati da un numero. Essi sono stati dichiarati esplicitamente tramite la clausola `data_type(Attribute, Type)`, che definisce la tipologia per ogni attributo.

Per ogni attributo, quindi, si é avviato un processo di suddivisione (`update_categories/0` e `make_class/2`) in piú *range*, definite dal predicato `class/2`:

- gli attributi di tipo `category` sono giá automaticamente partizionati, per cui una classe di tipo categorico avrá i range corrispondenti a tutti i valori di categoria
- ogni attributo di tipo `number` é stato suddiviso in 10 range di dimensioni uguali[11]

Il modulo `categories` contiene, nelle sue due varianti, il predicato `is_in_range`, che risulta soddisfatto solo se il valore (numerico o categorico che sia) rientra nel *range* specificato.

[10] La regola `get_records/2` prepara lo statement all'esecuzione.

[11] L'approccio utilizzato é semplicistico, quindi passibile di miglioramento, vedi 9.

7.5 Apprendimento e test

Dopo aver generato le categorie, é possibile avviare il processo (iterativo e ricorsivo) di apprendimento e test tramite la regola `learn_please/0`.

Lo scopo di questa fase é di generare due tipi di regole:

- `is_positive(ID, LearningStep)` determina se, secondo le regole generate ad un certo passo `LearningStep`, un esempio con un determinato `ID` é ritenuto un positivo.
- `test_step(LearningStep, StepData)` restituisce diverse misure relative allo step specificato, fra cui *true positive rate*, *false positive rate*, *F-Measure*, ecc.

L'iterazione principale del processo di apprendimento é definita dal *k-fold cross-validation*:

1. L'insieme degli esempi (positivi e negativi) viene diviso in k sottoinsiemi. Per lo scopo del progetto, si é impostato un k fisso a 10.
2. Ad ogni iterazione, un sottoinsieme viene messo da parte per la fase di test, mentre gli altri 9 concorrono all'apprendimento vero e proprio.
3. L'apprendimento viene avviato, generando regole di tipo `is_positive/2`.
4. Viene eseguito il test con l'insieme di esempi selezionato, generando regole di tipo `test_step/2`.
5. Si itera dal punto 2 fino ad esaurimento dei k *fold* creati.
6. Tutte le regole create vengono unite in un unico insieme, rimuovendo i duplicati.

L'apprendimento, per ogni fase, é avviato da `learn(Step)`.

7.5.1 Suddivisione in fold

La suddivisione in k fold distinti viene eseguita, ad ogni passo, dal predicato `split_examples/2` (vedi figura 3 per un esempio), che asserisce in memoria alcune liste di `ID` di esempi. Tali liste sono poi utilizzate nel filtraggio, quando richiesto, degli esempi in *training* e *testing*.

Nei primi test del programma, si é notato un bassissimo livello di distribuzione degli esempi nei diversi fold di ogni esecuzione, per cui i primi *run* avevano un bassissimo livello di *FP rate* (spesso 0), che cresceva con l'aumentare dell'indice del passo.
Ció era dovuto non alla generazione di regole ottimali, ma al fatto che gli esempi positivi venivano selezionati, da Prolog, sempre dopo quelli negativi, comportando un insieme di test composto, ai primi passi, da molti esempi positivi (mentre erano pochi o nulli alla fine) e da pochi negativi (mentre erano molti alla fine).

Si é pertanto reso necessario "forzare" una distribuzione piú o meno uniforme andando a dividere i *set* di positivi e negativi

in *fold* separati, che sono poi stati comunque accorpati in due liste complessive, visionabili a fine step con:

```
?- train_examples(TrainExamples).
?- test_examples(TestExamples).
```

Il filtraggio degli esempi viene poi eseguito da `test_example/4` e `train_example/4` e da altre regole derivate.

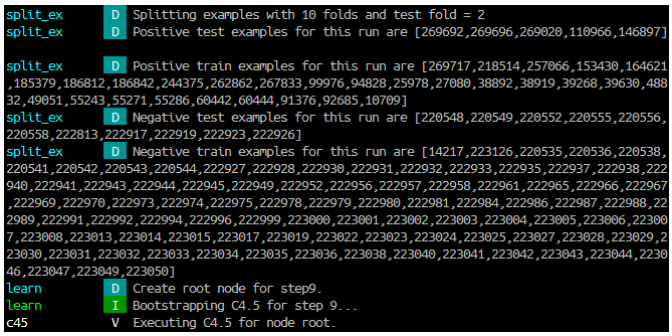

Figure 3: Suddivisione in 10 fold, al passo 9: il secondo *fold* viene selezionato per il test, gli esempi rimanenti come training set

7.5.2 *Apprendimento*

L'implementazione dell'algoritmo C4.5 é abbastanza diretta, e si compone di passi ricorsivi che iniziano dalla generazione di un nodo radice.

In generale, tutti i nodi dell'albero di decisione sono cosí composti:

```
node(Name, Parent, SplitAttribute, SplitRange)
```

dove:

- `Name` corrisponde ad un generico nome, che non é detto sia univoco, per il nodo. Genericamente, il nome del nodo é composto dall'attributo su cui il nodo genitore é stato suddiviso e una descrizione testuale del *range* che il nodo rappresenta.
- `Parent` rappresenta il `node/4` genitore; per questa ragione, ogni nodo puó essere identificato univocamente dalla coppia `(Name, Parent)`.
- `SplitAttribute` é l'attributo su cui é stato eseguito lo *split* per il livello corrente dell'albero.
- `SplitRange` é il range dell'attributo di *split* che il nodo corrente espande.

Il nodo radice, non potendo derivare nessuno dei precedenti parametri, ha come valori l'atomo `root`, per semplicitá.

I nodi foglia dell'albero sono identificati dalla regola:

```
node_label(Node, Value)
```

che specifica che un nodo dell'albero é terminale e identifica tutti gli esempi rimanenti come appartenenti a uno specifico range della classe target.

Passi base. Il processo ricorsivo dell'algoritmo C4.5 inizia, quindi, dal nodo `root`. Ogni chiamata alla regola `c45/3` richiede che siano istanziate le seguenti variabili:

- `Node` deve essere un `node/4`, e deve contenere le informazioni sul nodo che si vuole suddividere in piú *split*.
- `Examples` é una lista di `ID` di esempi, ovvero quei dati che devono essere processati al passo corrente.
- `Attributes` è la lista di nomi di attributi che non sono ancora stati selezionati per lo *split*.

Ovviamente il primo passo vedrá `Examples` contenere tutti gli esempi e `Attribute` tutti gli attributi.

Il primo controllo che viene eseguito da `c45/3` verifica se tutti gli `Examples` appartengono giá ad un'unica classe target, ovvero se tutti gli esempi hanno una stessa sintomatologia. In caso affermativo:

1. Si asserisce in memoria un `node_label/2` che dichiara che il nodo corrente ha come unica classe target quella rilevata.
2. Il nodo corrente non viene piú espanso.
3. Viene restituito il controllo al chiamante, ovvero il livello (nodo) superiore.

Se tale controllo non é soddisfatto, se ne fa un altro relativamente alla lista di attributi rimanenti; se la lista é vuota, infatti, si segue lo stesso procedimento per il controllo precedente, con l'unica differenza che il range associato a `node_label` é quello relativo al valore piú comune fra gli esempi rimanenti.

Se nessuna delle verifiche precedenti porta a risultati, il nodo corrente dovrá essere nuovamente diviso in *split*. Il processo, allora, prosegue con:

1. Selezione del migliore attributo.
2. Split sui range dell'attributo selezionato.
3. Chiamata ricorsiva a `c45/3`.

Attributo migliore. Per poter decidere quale, fra gli attributi rimanenti, costituisce il migliore ai fini del processo di apprendimento, si é deciso di utilizzare la misura dell' *information gain* (vedi 5.3.2), che a sua volta si basa fortemente sull'entropia (vedi 5.3.1). Un esempio di esecuzione é visibile in figura 4, dove viene scelto `PatientSex` poiché possiede il massimo *information gain*.

L'entropia viene calcolata dalla regola `entropy/2`, che utilizza i `train_example/4`; l'*information gain*, invece, viene calcolato da `info_gain/3`, che si basa sul calcolo dell'entropia e che somma, per ogni range dell'attributo A, il valore restituito da `partial_info_gain/4`:

$$\frac{|S_a|}{|S|} \times entropia(S_a)$$

```
c45-test      V  Non null different training targets left: [1,6]
c45           D  Looking for the best attribute to split [ SAPStart : rang
e(152.8,163.60000000000002) ]
info_gain     V  Info gain for PatientSex is 0.17106214692568505
info_gain     V  Info gain for PatientRace is 0.0
info_gain     V  Info gain for ProgWeightLoss is 0.06109149418738888
info_gain     V  Info gain for RealWeightLoss is 0.08844760214785355
info_gain     V  Info gain for DeltaWeight is 0.03688100226567101
info_gain     V  Info gain for ProgDuration is 0.0028521585923558246
info_gain     V  Info gain for RealDuration is 0.06109149418738888
info_gain     V  Info gain for DeltaDuration is 0.08844760214785355
info_gain     V  Info gain for SAPEnd is 0.10107777847191066
info_gain     V  Info gain for SAPAverage is 0.07794434474845499
info_gain     V  Info gain for DAPStart is 0.10107777847191066
info_gain     V  Info gain for DAPEnd is 0.17106214692568505
info_gain     V  Info gain for DAPAverage is 0.10107777847191066
info_gain     V  Info gain for DeltaBloodFlow is 0.1382409511794473
info_gain     V  Info gain for DeltaUF is 0.0028521585923558246
best_attr     D  Best info gain is achieved with attribute PatientSex with
 a value of 0.17106214692568505
best_attr     V  Best attribute calculus took 0m 0s 675ms.
```

Figure 4: Scelta dell'attributo migliore secondo l'*information gain*

Si noti che la misura dell'*information gain* equivale all'entropia quando tutti i sottoinsiemi selezionati dalle coppie attributo-range:

- sono vuoti, rendendo $|S_a| = 0$
- alternativamente se tutti gli esempi appartengono a una sola delle due classi (positivo o negativo), cioé se il sottoinsieme selezionato ha minima entropia (0)

Split sui range. Dopo aver calcolato l'*information gain* per tutti gli attributi, si seleziona quello con la misura piú elevata e si ottengono tutti i suoi *range*.

Creazione nodi figli. Per ogni range dell'attributo si crea un nuovo nodo avente come nome il template seguente:

```
[ Attribute : range(Inizio, Fine) ]
```

Si chiama, infine, la regola `c45/3` con parametri:

- `Node`, il nodo appena creato
- `Examples`, l'insieme degli esempi in ingresso al passo corrente che soddisfano il *range* selezionato
- `Attributes`, l'insieme di attributi in ingresso al passo corrente, eccetto l'attributo su cui é stato eseguito lo *split*

Terminazione locale. Quando non ci sono piú esempi da analizzare, o gli attributi su cui eseguire lo *split* sono terminati, l'algoritmo termina, avendo prodotto un albero di decisione completo.

```
[ _G27115 : root ]
    ✓ [ PatientAge : range(13395,15636) ]
    ✓ [ PatientAge : range(15636,17877) ]
    ✓ [ PatientAge : range(17877,20118) ]
    ✓ [ PatientAge : range(20118,22359) ]
    ✓ [ PatientAge : range(22359,24600) ]
    ✓ [ PatientAge : range(24600,26841) ]
    ▼ [ PatientAge : range(26841,29082) ]
        ✓ [ DeltaBloodFlow : range(-20.0,-7.6) ]
        ✓ [ DeltaBloodFlow : range(-7.6,4.800000000000001) ]
        ✓ [ DeltaBloodFlow : range(4.800000000000001,17.200000000000003) ]
        ✓ [ DeltaBloodFlow : range(17.200000000000003,29.6) ]
        ✓ [ DeltaBloodFlow : range(29.6,42.0) ]
        ✗ [ DeltaBloodFlow : range(42.0,54.4) ]
        ✓ [ DeltaBloodFlow : range(54.4,66.8) ]
        ✓ [ DeltaBloodFlow : range(66.8,79.2) ]
        ✓ [ DeltaBloodFlow : range(79.2,91.60000000000001) ]
        ✓ [ DeltaBloodFlow : range(91.60000000000001,104.00000000000001) ]
    ▼ [ PatientAge : range(29082,31323) ]
        ✗ [ DeltaBloodFlow : range(-20.0,-7.6) ]
        ▼ [ DeltaBloodFlow : range(-7.6,4.800000000000001) ]
            ✗ [ DAPAverage : range(80.0,94.0) ]
            ✗ [ DAPAverage : range(94.0,108.0) ]
            ✗ [ DAPAverage : range(108.0,122.0) ]
            ▼ [ DAPAverage : range(122.0,136.0) ]
                ✓ [ PatientSex : range(0,0) ]
                ✗ [ PatientSex : range(1,1) ]
            ▼ [ DAPAverage : range(136.0,150.0) ]
                ✗ [ RealDuration : range(150.0,165.0) ]
                ✗ [ RealDuration : range(165.0,180.0) ]
                ✗ [ RealDuration : range(180.0,195.0) ]
                ✓ [ RealDuration : range(195.0,210.0) ]
                ✗ [ RealDuration : range(210.0,225.0) ]
                ✗ [ RealDuration : range(225.0,240.0) ]
                ✗ [ RealDuration : range(240.0,255.0) ]
                ✗ [ RealDuration : range(255.0,270.0) ]
                ✗ [ RealDuration : range(270.0,285.0) ]
                ✗ [ RealDuration : range(285.0,300.0) ]
                ✗ [ RealDuration : range(300.0,315.0) ]
            ✓ [ DAPAverage : range(150.0,164.0) ]
            ✓ [ DAPAverage : range(164.0,178.0) ]
            ✗ [ DAPAverage : range(178.0,192.0) ]
            ✗ [ DAPAverage : range(192.0,206.0) ]
            ✗ [ DAPAverage : range(206.0,220.0) ]
            ✗ [ DAPAverage : range(220.0,234.0) ]
        ✗ [ DeltaBloodFlow : range(4.800000000000001,17.200000000000003) ]
        ✗ [ DeltaBloodFlow : range(17.200000000000003,29.6) ]
        ✗ [ DeltaBloodFlow : range(29.6,42.0) ]
        ✗ [ DeltaBloodFlow : range(42.0,54.4) ]
        ✗ [ DeltaBloodFlow : range(54.4,66.8) ]
        ✗ [ DeltaBloodFlow : range(66.8,79.2) ]
        ✗ [ DeltaBloodFlow : range(79.2,91.60000000000001) ]
        ▼ [ DeltaBloodFlow : range(91.60000000000001,104.00000000000001) ]
            ✗ [ PatientSex : range(0,0) ]
            ✓ [ PatientSex : range(1,1) ]
    ✓ [ PatientAge : range(31323,33564) ]
    ✗ [ PatientAge : range(33564,35805) ]
    ✓ [ PatientAge : range(35805,38046) ]
```

Figure 5: Stampa dell'albero generato al passo corrente

Viene quindi stampata a video, richiamando `print_le_tree/0`, una rappresentazione grafica e formattata dell'albero di decisione prodotto, in cui ogni elemento foglia (positivo) avrá un check verde (vedi figura 5).

7.5.3 Generazione di regole

Una volta appreso l'albero di decisione ad uno specifico passo i, il programma genera le regole `is_positive/2` andando a risalire l'albero di decisione da ogni nodo foglia positivo `node_label/2`, e aggiungendo in lista una condizione di appartenenza dell'attributo su cui il nodo é stato "splittato" al

range del nodo stesso[12].

La regola principale che si occupa della generazione delle regole é `gen_all_the_rulez/1`, che chiama ricorsivamente `gen_rule/2` per ogni nodo foglia.

7.5.4 Test del passo

Dopo la generazione delle regole al passo i, si esegue un processo di test, che asserisce in memoria (nel predicato `test_step/2`) una lista con diverse informazioni relative al passo in analisi.

I predicati `p/1` e `n/1` esplicitano, banalmente, il numero di positivi e negativi analizzati al passo corrente. Tali valori sono utilizzati, insieme ad altri calcoli, per generare anche:

- `tn(TrueNegatives)`, ovvero la quantitá di negativi non classificati (ovvero classificati come negativi, poiché il sistema genera regole per la sola classe positiva)
- `fn(FalseNegatives)`, cioé i positivi erroneamente classificati come negativi
- `tp(TruePositives)`, i positivi correttamente classificati
- `fp(FalsePositives)`, cioé i negativi erroneamente classificati come positivi
- `tp_rate(TruePosRate)`, la proporzione dei veri positivi su tutti i positivi
- `tn_rate(TrueNegRate)`, la proporzione dei veri negativi su tutti i negativi
- `fp_rate(FalsePosRate)`, la proporzione dei falsi positivi su tutti i negativi
- `fn_rate(FalseNegRate)`, la proporzione dei falsi negativi su tutti i positivi

Inoltre, vengono calcolate anche *precision* e *recall*:

$$\begin{cases} precision = \frac{tp}{tp+fp} \\ recall = \frac{tp}{tp+fn} \end{cases}$$

Tali valori vengono esposti, nella lista restituita da `test_step/2`, come `precision(Precision)` e `recall(Recall)`.

Infine, viene calcolata e salvata anche la F_1-$Measure$, equivalente a:

$$F_1\text{-}Measure = \frac{2 * precision * recall}{precision + recall}$$

7.6 Terminazione dell'apprendimento

Una volta che tutti i k cicli di apprendimento-test sono terminati, il programma termina l'esecuzione tramite alcuni passi finali.

7.6.1 Generazione regole

Tutte le regole generate ai diversi passi vengono inglobate in un unico insieme di regole e riasserite nella memoria Prolog come `final_positive(ID, ListaPassi)`, dove `ListaPassi` é la lista degli indici dei passi che hanno generato quella regola.

Il metodo che si occupa di unire le regole in un insieme singolo é `purge_rules/0` che, avvalendosi dei predicati `clause/2` e `bagof/3`, ricerca tutti i corpi delle regole `is_positive/2` e li unisce, rimuovendo i duplicati.

7.6.2 Test finale

Una volta unite le regole, queste vengono applicate all'**intero insieme di esempi**, in modo da validare in maniera completa tutto il processo di apprendimento. Tale passo si rivela importante soprattutto nel caso in cui l'insieme degli esempi di partenza é molto piccolo.

Il test asserisce, in memoria Prolog, il fatto `test_final/1`, che contiene le stesse informazioni presenti in `test_step/2` (cfr. 7.5.4).

7.6.3 Stampa report

Tutte le informazioni vengono quindi stampate a schermo da `print_report/0` (vedi figura 6) che, avvalendosi del sistema di *logging* implementato, mostra a livello `info`:

- una ricapitolazione di tutte le informazioni asserite ad ogni esecuzione
- la media delle singole esecuzioni[13]
- l'esecuzione del test finale

7.6.4 Salvataggio log e regole

Al termine di ogni processo di apprendimento viene creato un file, `prolog/runs/log_{ID}.csv`[14] contenente, per ogni esecuzione, il sintomo, il passo, il numero di regole generate e altre informazioni (vedi 8.1); infine, il file riporta il tempo di esecuzione in secondi, il numero totale di esempi positivi presenti, il numero totale di regole generate e tutti gli altri tipi di dati giá riportati per ogni passo.

Inoltre, tutte le regole unite alla fine dell'apprendimento (vedi 7.6.1) vengono salvate nel file `runs/rules_{ID}.pl`[15], ad esempio:

```
final_positive(A, [1,2,3,4,5,6,7,8,9,10]) :-
check_condition_list(A,

    [ condition('PatientAge',
 range(22231.000000000004,
       24032.200000000004))
    ]).
```

[12] Vengono generate regole per i soli nodi foglia positivi.

[13] Come giá detto, la media potrebbe essere poco indicativa per dataset molto piccoli.

[14] Per la creazione dei file di log, é necessario che la cartella `runs` esista giá.

[15] Per la verifica delle regole é necessario che sia caricato in memoria Prolog anche la regola `check_condition_list/2`, che serve ad eseguire il *matching* dei *range* descritti dalle regole generate.

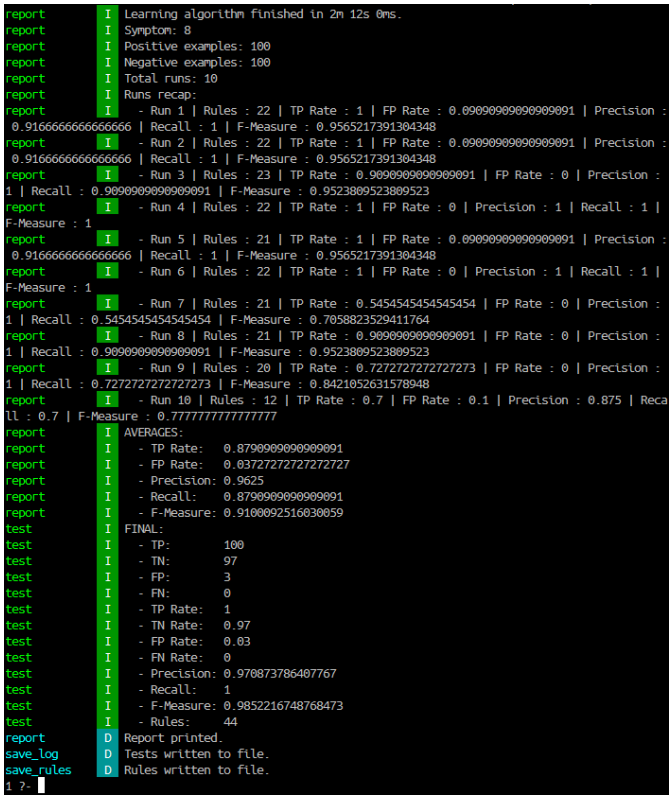

Figure 6: Stampa del report di fine processo

8. RISULTATI

Il programma é stato eseguito su 9 sintomi target distinti e con un diverso numero di esempi positivi[16].

8.1 Prime considerazioni

La tabella 5 mostra un breve riepilogo dei risultati dei test (finali) eseguiti sull'intero dataset, per i sintomi 2, 3, 4 ,5, 6, 7, 8, 9, 10. I valori mostrati in tabella[17] sono:

- `ID` del sintomo analizzato
- Tempo di esecuzione, in secondi, del processo di apprendimento e di test con la 10-*fold cross-validation*
- Numero di esempi positivi analizzati (i negativi sono sempre 100)
- Numero di regole (differenti) generate
- *True Positive Rate*, ovvero il rapporto dei veri positivi sul numero dei positivi
- Misura di *precision*
- Misura di *recall*
- La *F-Measure*

	T	+	#Reg	TPR	Prec.	Rec.	F-M.
2	66	19	2	0.053	0.5	0.053	0.095
3	33	56	2	0.179	0.909	0.179	0.299
4	12	7	0	0	0	0	0
5	50	100	10	0.79	0.94	0.79	0.859
6	15	34	0	0	0	0	0
7	11	1	1	1	1	1	1
8	33	100	1	0.01	1	0.01	0.0198
9	16	5	14	1	0.833	1	0.909
10	25	59	0	0	0	0	0

Table 5: Risultati dei test su 9 sintomi, includendo fra gli attributi sia `KTV` che `QB`

Durante l'esecuzione del programma sui vari sintomi, peró, si é notato che gli attributi `KTV` e `QB` condizionavano enormemente le regole prodotte, poiché venivano scelti molto spesso come attributo con maggior *information gain*.

8.2 Eliminazione di KTV e QB

Si é quindi effettuata un'altra esecuzione del programma, sugli stessi sintomi, eliminando gli attributi `KTV` e `QB` dall'analisi. I risultati di tale esecuzione sono riportati in tabella 6, dalla quale risultano evidenti alcune situazioni.

Innanzitutto si nota che con un **basso numero di esempi positivi** vengono generate numerose regole. Tuttavia, bisogna considerare che alcune regole potrebbero essere superflue, ovvero hanno senso di esistere esclusivamente all'interno di uno specifico insieme di *fold* considerato ad un passo; potrebbe capitare, infatti, che esistano regole piú generali che coprano almeno gli stessi esempi di una o piú regola specifica, senza introdurre grosse variazioni di precisione[18].

Avendo a disposizione un **elevato numero di esempi positivi** (si vedano i sintomi 3, 5 e 8), invece, la proporzione *regole/positivi* si mantiene intorno a 0.5. Anche in questo caso, comunque, si deve fare la stessa considerazione precedente, ovvero che alcune regole potrebbero essere eliminate senza grossi problemi.

Altre regole, invece, potrebbero essere eliminate andando ad accorpare range "adiacenti" (immediatamente successivi) di attributi si trovano in alto nell'albero di decisione generato.

8.3 Altri sistemi

Lo stesso dataset é stato sottoposto ad altri sistemi di apprendimento automatico, in modo tale da poter valutare ció che é stato realizzato, paragonandolo ad algoritmi ben piú collaudati.

[16] La macchina utilizzata per il test ha le seguenti caratteristiche:
- CPU Intel i7-4500U @ 1.80GHz
- 8GB RAM DDR3
- Samsung SSD EVO 840 250GB (fino a 540 MB/s in lettura, fino a 520 MB/s in scrittura)
- Windows 8.1 x64

[17] Tutti i report delle esecuzioni, con tutti i valori calcolati, anche parziali, si trovano nella cartelle `prolog/runs/` e `prolog/runs_ktv_qb/`.

[18] In tal senso, sarebbe utile estendere il programma con una funzionalitá di pulizia delle regole superflue.

	T	+	#Reg	TPR	Prec.	Rec.	F-M.
2	90	19	22	1	0.826	1	0.905
3	90	56	26	1	0.875	1	0.933
4	25	7	23	1	0.636	1	0.778
5	358	100	60	1	0.909	1	0.952
6	220	34	49	1	0.829	1	0.907
7	12	1	1	1	1	1	1
8	132	100	44	1	0.971	1	0.985
9	14	5	15	1	0.833	1	0.909
10	51	59	25	1	0.967	1	0.983

Table 6: Risultati dei test su 9 sintomi, escludendo dagli attributi sia KTV che QB

8.3.1 Weka

Il tool di Data Mining utilizzato é Weka [3], che mette a disposizione numerosi algoritmi pre-implementati per la classificazione e generazione di alberi di decisione (e regole). Sono stati testati 3 diversi algoritmi sul sintomo target 8: J48, J48Graft e ConjunctiveRule.

Per l'importazione degli esempi é stata eseguita la query:

```
(SELECT
  ‘PATIENT_SEX‘, ‘PATIENT_RACE‘, ‘PATIENT_AGE‘,
  ‘PROG_WEIGHT_LOSS‘, ‘REAL_WEIGHT_LOSS‘,
  ‘DELTA_WEIGHT‘, ‘PROG_DURATION‘, ‘REAL_DURATION‘,
  ‘DELTA_DURATION‘, ‘SAP_START‘, ‘SAP_END‘,
  ‘AVG_SAP‘, ‘DAP_START‘, ‘DAP_END‘, ‘AVG_DAP‘,
  ‘BLOOD_VOLUME‘, ‘DELTA_BLOOD_FLOW‘,
  ‘DELTA_UF‘, ‘SYMPTOM_ID‘
FROM ‘patient_dialysis_symptom_for_analysis‘
WHERE
  ‘SYMPTOM_ID‘ = 1 ORDER BY ‘SCORE‘ DESC LIMIT 100)
UNION
(SELECT
  ‘PATIENT_SEX‘, ‘PATIENT_RACE‘, ‘PATIENT_AGE‘,
  ‘PROG_WEIGHT_LOSS‘, ‘REAL_WEIGHT_LOSS‘,
  ‘DELTA_WEIGHT‘, ‘PROG_DURATION‘, ‘REAL_DURATION‘,
  ‘DELTA_DURATION‘, ‘SAP_START‘, ‘SAP_END‘,
  ‘AVG_SAP‘, ‘DAP_START‘, ‘DAP_END‘, ‘AVG_DAP‘,
  ‘BLOOD_VOLUME‘, ‘DELTA_BLOOD_FLOW‘,
  ‘DELTA_UF‘, ‘SYMPTOM_ID‘
FROM ‘patient_dialysis_symptom_for_analysis‘
WHERE
  ‘SYMPTOM_ID‘ = 8 ORDER BY ‘SCORE‘ DESC LIMIT 100)
```

Una volta importati i dati del dataset di esempio in Weka Explorer (si veda B.2 per una guida all'importazione e trasformazione), si possono andare a selezionare gli algoritmi da applicare.

J48. J48 è un'implementazione in Java di C4.5 per Weka che opera applicando, all'albero di decisione generato, tecniche di *pruning* per semplificarne la descrizione.

Lasciando i parametri di default in Weka, selezionando la 10-*fold cross-validation* come modalitá di test e avviando la classificazione per il sintomo target 8, sono stati ottenuti i risultati mostrati in tabella 7.

L'output di esecuzione restituito[19] é il seguente:

```
=== Classifier model (full training set) ===

J48 pruned tree
------------------

PATIENT_AGE <= 29067: 8 (88.0/1.0)
PATIENT_AGE > 29067
|   PATIENT_AGE <= 30141
|   |   DAP_START <= 79: 1 (93.0/1.0)
|   |   DAP_START > 79
|   |   |   DELTA_BLOOD_FLOW <= 12: 8 (5.0/1.0)
|   |   |   DELTA_BLOOD_FLOW > 12: 1 (6.0)
|   PATIENT_AGE > 30141: 8 (8.0)

Number of Leaves  :   5

Size of the tree :  9


Time taken to build model: 0 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances   189  94.5 %
Incorrectly Classified Instances 11   5.5  %
```

J48	
T	0
+	100
#Reg	3
TPR	0.94
Precision	0.949
Recall	0.94
F-Measure	0.945

Table 7: Risultato di esecuzione con J48

A fronte di un tempo di esecuzione di gran lunga inferiore (praticamente nullo), precision e recall non differiscono di molto:

$$\begin{cases} precision_m = 0.971 \\ recall_m = 1 \\ fmeasure_m = 0.985 \end{cases} \text{ vs } \begin{cases} precision_{j48} = 0.949 \\ recall_{j48} = 0.94 \\ fmeasure_{j48} = 0.945 \end{cases}$$

J48 Graft. J48 Graft é un algoritmo incluso in Weka che genera un albero di decisione con lo stesso algoritmo di J48, ma applicando la tecnica del *grafting*[9], ovvero cercando di aggiungere, dove necessario, nuovi nodi all'albero prodotto per cercare di ridurre l'errore predittivo.

Utilizzando lo stesso dataset precedente e la stessa modalitá di testing con 10-fold cross-validation, l'output principale ottenuto[20] é il seguente:

[19] L'output non é completo; per l'output completo, vedere il file `weka/trees.J48.log`.

[20] L'output completo é leggibile nel file

```
=== Classifier model (full training set) ===

J48graft pruned tree
------------------

PATIENT_AGE <= 29067: 8 (88.0/1.0)
PATIENT_AGE > 29067
|   PATIENT_AGE <= 30141
|   |   DAP_START <= 79: 1 (93.0/1.0)
|   |   DAP_START > 79
|   |   |   DELTA_BLOOD_FLOW <= 12: 8 (5.0/1.0)
|   |   |   DELTA_BLOOD_FLOW > 12: 1 (6.0)
|   PATIENT_AGE > 30141: 8 (8.0)

Number of Leaves  :   5

Size of the tree :  9


Time taken to build model: 0.01 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances   187  93.5 %
Incorrectly Classified Instances 13   6.5  %
```

L'albero di decisione generato, e le misurazioni complete mostrate in tabella 8 non indicano alcuna differenza con J48.

J48 Graft	
T	0.01
+	100
#Reg	3
TPR	0.92
Precision	0.948
Recall	0.92
F-Measure	0.934

Table 8: Risultato di esecuzione con J48 Graft

Conjunctive Rule. Un output piú simile a quello ottenuto dal programma realizzato é restituito dall'algoritmo Conjunctive Rule di Weka, che genera un insieme di regole composte da un antecedente (il corpo) di clausole in congiunzione ed un sequente (la testa) che corrisponde al valore classificato.
Anche Conjunctive Rule calcola la misura dell'*information gain* del corpo della regola generata ed esegue il *pruning* in base al numero delle clausole presenti. Per la classificazione, l'informazione relativa al corpo della regola generata é la media pesata delle entropie degli esempi coperti dalla regola e di quelli non coperti.

L'esecuzione di Conjunctive Rule sul solito dataset ha prodotto le misure in tabella 9 e il seguente output[21]:

```
=== Classifier model (full training set) ===

Single conjunctive rule learner:
--------------------------------
(PATIENT_AGE <= 29653.5) => SYMPTOM_ID = 8

Class distributions:
Covered by the rule:
1 8
0 1

Not covered by the rule:
1 8
0.893333  0.106667

Time taken to build model: 0.21 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances   185  92.5 %
Incorrectly Classified Instances 15   7.5  %
```

Conjunctive Rule	
T	0.21
+	100
#Reg	1
TPR	0.87
Precision	0.978
Recall	0.87
F-Measure	0.921

Table 9: Risultato di esecuzione con Conjunctive Rule

A fronte degli stessi esempi di training, quindi, Conjunctive Rule ha generato una sola regola, mantenendo elevata la precisione a discapito del *recall*, comportando un maggior numero di falsi negativi (mancate classificazioni).

$$\begin{cases} precision_m = 0.971 \\ recall_m = 1 \\ fmeasure_m = 0.985 \end{cases} \text{ vs } \begin{cases} precision_{cr} = 0.978 \\ recall_{cr} = 0.87 \\ fmeasure_{cr} = 0.921 \end{cases}$$

Andando a confrontare le regole generate dal sistema realizzato e la regola generata da Conjunctive Rule, si trova qualche disaccordo. Ad esempio:

```
final_positive(A, [1, 2, 3, 4, 5, 6, 7, 8, 10]) :-
check_condition_list(A,
     [condition('PatientAge', range(35805, 38046))]).
```

é chiaramente opposta all'unica regola generata:

```
(PATIENT_AGE <= 29653.5) => SYMPTOM_ID = 8
```

Tra l'altro, la regola in esempio é stata generata in piú passi di apprendimento, con l'unica eccezione il passo in cui alcuni esempi che la generavano si trovavano, evidentemente, nel sottoinsieme di *testing*.

`weka/trees.J48graft.log`.

[21] L'output completo é leggibile nel file `weka/rules.ConjunctiveRule.log`.

8.3.2 BayesDB

BayesDB [1] è un "database di tabelle bayesiane che permette di inferire le probabili implicazioni dei dati tabulati, con la stessa facilitá con cui si chiedono i dati ad un database SQL". É un tool per inferenze molto recente, sviluppato dal MIT in collaborazione con DARPA e Google.

BayesDB, a partire da una tabella con qualsiasi attributo e valore, permette di:

- selezionare i dati con una regolare `SELECT`
- inferire i dati mancanti su alcune (o tutte le) colonne con uno specifico livello di confidenza, tramite `INFER`
- simulare uno o piú attributi a partire da altri forniti, con `SIMULATE`
- stimare le probabilitá di dipendenza fra gli attributi della tabella, con `ESTIMATE DEPENDENCE PROBABILITIES`

Dopo il setup iniziale di BayesDB (vedi C.1), si é proceduto prima all'importazione dei dati, quindi al *querying* e *testing*.

Esportazione dei dati. La migrazione dei dati da MySQL a BayesDB é avvenuta[22] tramite selezione di:

- 100 esempi negativi (sintomo 1)
- 100 esempi positivi (sintomo 8)
- 200 esempi non classificati

In realtá, per eseguire un test che avesse significato, i 200 esempi non classificati sono gli stessi esempi positivi e negativi; per simulare il *testing*, tutti i dati hanno sia una colonna `SYMPTOM_ID`, contenente il valore target (nullo negli esempi non classificati), che una colonna `REAL_SYMPTOM_ID` contenente il valore reale del dato di testing da controllare dopo la classificazione.
Successivamente, su BayesDB é stato specificato di non considerare come attributo di apprendimento `REAL_SYMPTOM_ID`, che altrimenti avrebbe avuto una probabilitá di dipendenza con l'attributo target `SYMPTOM_ID` di 1!

Lo script di migrazione, troppo lungo per essere riportato[23] include una prima riga (header fisso) contenente i nomi delle colonne, e di seguito i 400 esempi sopra descritti.

Tramite la *query* di MySQL:

```
SELECT
...
INTO OUTFILE 'learn_data.csv'
FIELDS TERMINATED BY ','
```

si genera un file `csv` contenente l'*header*, i dati e i campi separati da virgola, come richiesto da BayesDB.

Importazione dei dati. Una volta esportati i dati, ci si é collegati alla macchina virtuale su cui BayesDB é installato e sono stati copiati i seguenti file (per il collegamento vedi C.2, per la copia C.3):

- `learn_data.csv` precedentemente generato
- lo script python `learn_data_import.py`, appositamente creato per caricare i dati

Dalla macchina virtuale (o tramite accesso SSH), é stato eseguito il comando:

```
$ python /home/bayesdb/learn_data_import.py
```

Lo script esegue, a catena, i seguenti comandi:

1. elimina un'eventuale tabella precedentemente creata
2. importa i dati dal `csv` nella tabella `dialysisai`
3. specifica come `continuous` gli attributi `PROG_DURATION` e `BLOOD_VOLUME`, e ignora `REAL_SYMPTOM_ID`
4. crea 20 modelli per la tabella creata
5. analizza la tabella per 100 iterazioni[24]

A questo punto, il modello bayesiano é stato costruito, ed é possibile iniziare a fare inferenze[25].

Stime delle probabilitá di dipendenza. Si sono volute valutare le probabilitá di dipendenza fra gli attributi del dataset caricato; pertanto é stata eseguita la *query* (per conoscere come eseguire *query* su BayesDB, consultare C.4):

```
ESTIMATE DEPENDENCE PROBABILITIES FROM dialysisai;
```

che ha generato l'immagine in figura 7, dalla quale sono ricavabili tutte le dipendenze.

Essendo interessati esclusivamente alle dipendenze fra la sintomatologia e tutti gli altri attributi, si raffina la *query* precedente:

```
ESTIMATE DEPENDENCE PROBABILITIES FROM dialysisai
REFERENCING symptom_id WITH CONFIDENCE 0.9;
```

che restituisce l'immagine in figura 8; questa mostra solo gli attributi con una evidente una dipendenza della sintomatologia, quali differenza del flusso del sangue, età e sesso (ignorando la banale dipendenza con `real_symptom_id`).

[22] Il file di migrazione é disponibile in `bayesdb/learn_data.csv`.
[23] Lo script completo é contenuto in `bayesdb/learn_data_export.sql`
[24] Questa operazione potrebbe impiegare molti minuti per completare.
[25] Il modello é stato pre-costruito ed esportato in

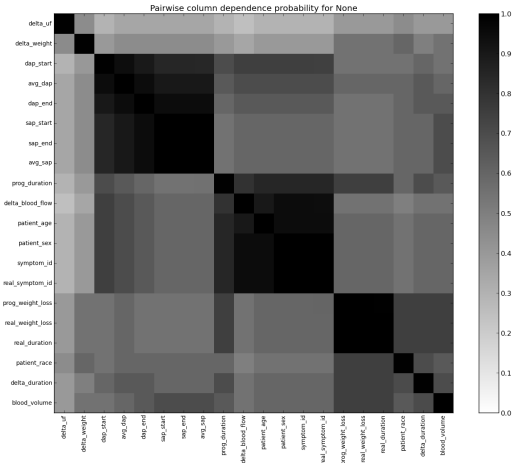

Figure 7: Probabilitá di dipendenza fra gli attributi

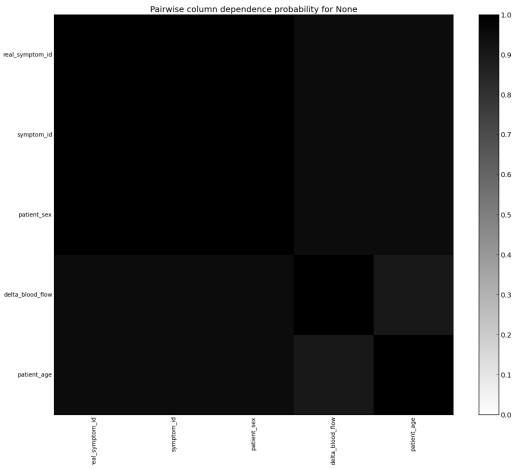

Figure 8: Probabilitá di dipendenza fra sintomo e gli altri attributi

Predizione del sintomo. Per predire la sintomatologia per ogni record non classificato, si puó eseguire la *query*:

```
INFER symptom_id
FROM dialysisai
WITH CONFIDENCE 0.9;
```

Il risultato restituito a video é la lista di tutte le predizioni dei valori di sintomatologia: dove il valore é stato predetto, esso é preceduto da un asterisco `*`, altrimenti viene mostrato il valore pre-esistente.

Test e risultati. Il Bayesian Query Language (BQL), linguaggio alla base di BayesDB, non é ancora del tutto completo[26], pertanto non permette una veloce aggregazione dei risultati (conteggi, medie, ecc.)[27].
Per questa ragione, é stato creato uno script Python[28] che, una volta avviato, stampa a video e su file[29] il risultato dei test sulla predizione effettuata.

La *query* eseguita seleziona il sintomo vero, che non verrá modificato poiché sempre presente, e il sintomo da predire, che sará inferito[30] dal sistema con un livello di confidenza di 0.9.

```
INFER real_symptom_id, symptom_id
FROM dialysisai
WITH CONFIDENCE 0.9;
```

Quindi, lo script procede ad analizzare i dati restituiti dalla query, calcolando *true positive rate*, *false positive rate*, *precision*, *recall* e F-Measure, mostrate in tabella 10.

BayesDB	
+	100
TPR	0.90
FPR	0.01
Precision	0.98
Recall	0.90
F-Measure	0.94

Table 10: Risultato di esecuzione di BayesDB

9. SVILUPPI FUTURI

L'implementazione base del C4.5 realizzata é migliorabile in molti punti.

Un primo miglioramento potrebbe riguardare la generazione dinamica dei range degli attributi: la cardinalitá di ogni range non sarebbe piú fissa, ma varierebbe a seconda della vicinanza dei valori degli esempi o secondo altri criteri.

Come giá detto in 8.2, sarebbe molto utile eliminare, a fine processo, tutte quelle regole che risultano superflue nel senso che non vanno a includere molti esempi positivi in piú rispetto alle altre; le regole, quindi, potrebbero risultare anche piú compatte.

Inoltre, si potrebbe confrontare la precisione del programma nel caso i valori nulli non vengano riempiti "a monte" del processo (ovvero nel database), e vengano quindi ignorati completamente.

Ad oggi, il programma gestisce gli esempi come strettamente positivi o negativi. Avendo a disposizione circa 60 sintomi, tuttavia, potrebbe risultare utile riadattare alcune parti dell'algoritmo con il fine di classificare, in un unico albero di

`bayesdb/dialysisai.pkl.gz`; per importarlo, eseguire `IMPORT SAMPLES dialysisai INTO dialysisai`.

[26] La versione utilizzata é la `0.1.0 alpha`.

[27] Tale funzionalitá, assieme alla possibilitá di eseguire *query* annidate, é stata annunciata per la prossima release.

[28] Lo script é `bayesdb/learn_data_test.py`.

[29] Il log avrá il nome `learn_data_test.log`.

[30] Il valore inferito viene solo caricato e mostrato a video, la base di dati non viene modificata.

decisione, tutti i sintomi. Un possibile svantaggio si potrebbe avere relativamente a sintomi che hanno pochi esempi che li soddisfano, e che potrebbero non essere rappresentati da nessuna regola generata. Ci sarebbe, in questo caso, anche da valutare il costo dell'algoritmo, sicuramente piú complesso (si veda il calcolo dell'entropia).

Infine, vista la scarsa distribuzione delle sintomatologie per seduta di dialisi, potrebbe essere utile imporre una certa soglia di tolleranza, ad esempio andando a classificare come nodo foglia anche i nodi che contengono esempi positivi in una certa percentuale (95% anziché 100% come attualmente implementato), ma comunque con pochi controesempi. In questo modo sarebbe anche possibile definire un ordine di prioritá per le regole generate, da quella "piú certa" a quella con la piú bassa probabilitá.

Miglioramenti implementativi potrebbero essere:

- personalizzazione del numero dei *fold* da utilizzare nella *k-fold cross-validation*
- spostamento dell'intero processo di apprendimento su macchine virtuali nel cloud; é giá in studio una migrazione su Google Compute Engine
- apprendimento multi-threading sui diversi split dei *k-fold*
- calcolo parallelo del miglior *information gain*
- apprendimento multi-threading sui diversi *range* dell'attributo migliore ad ogni livello dell'albero
- eliminazione di regole troppo lunghe o che classificano pochi elementi (necessario un set esterno di validazione)
- ordinamento delle regole in base alla *F-Measure* del passo in cui sono state apprese (diretta proporzionalitá tra ordine regola e ordine *F-Measure*) e alla loro semplicitá (minor numero di clausole, maggiore rilevanza)

10. REFERENCES

[1] Bayesdb, Apr. 2014.

[2] Swi-prolog, Apr. 2014.

[3] Weka, Apr. 2014.

[4] R. Bellazzi, C. Larizza, P. Magni, R. Bellazzi, and S. Cetta. Intelligent data analysis techniques for quality assessment of hemodialysis services. In *Proc. of the Workshop on Intelligent Data Analysis and Pharmacology*, 2001.

[5] A. Kusiak, B. Dixon, and S. Shah. Predicting survival time for kidney dialysis patients: a data mining approach. *Comput. Biol. Med.*, 35(4):311–327, May 2005.

[6] T. M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[7] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[8] S. L. Salzberg. Book review: C4.5: Programs for machine learning by j. ross quinlan. morgan kaufmann publishers, inc., 1993. *Mach. Learn.*, 16(3):235–240, Sept. 1994.

[9] G. Webb. Decision tree grafting from the all-tests-but-one partition. San Francisco, CA, 1999. Morgan Kaufmann.

APPENDIX

A. IMPORTAZIONE DEI DATI

Per utilizzare il database come sorgente dei dati, occorre installare MySQL come motore di database. Quindi:

1. Avviare il server MySQL.
2. Tramite il tool "Import Data" di MySQL Workbench, selezionare il file `scripts/mysq-dump.sql` e avviare l'importazione dei dati.
3. Avviare lo script `scripts/mysql-scores.sql` per la creazione delle *view* utili.

B. WEKA

B.1 Setup di Weka

Per utilizzare Weka con i file ARFF, localizzati nella cartella `weka/`, non serve nessun pre-requisito particolare. Per la connessione al database, invece, serve specificare il file `DatabaseUtils.props` che contiene alcune proprietá per la connessione.

Un file di esempio é `weka/DatabaseUtils.props`, che permette di collegarsi a fonti dati MySQL. In particolare, le informazioni rilevanti da gestire sono:

```
# JDBC driver (comma-separated list)
jdbcDriver=com.mysql.jdbc.Driver,
  org.gjt.mm.mysql.Driver

# database URL
jdbcURL=jdbc:mysql://localhost:3306/DialysisAI
```

Successivamente serve includere, se non giá presente, la libreria `jar` per le connessioni JDBC ai database MySQL nel *classpath* caricati da Weka. Ad esempio, su Windows serve modificare il file `RunWeka.ini` (presente nel percorso di installazione di Weka) in modo tale che la variabile `cp` includa, all'ultima riga, il percorso completo del file `jar`:

```
cp=%CLASSPATH%;%MY_SQL_PATH%/Connector J 5.1.29/
  mysql-connector-java-5.1.29-bin.jar
```

B.2 Importazione dati in Weka

Per importare i dati di addestramento in Weka, é possibile operare in due modi: utilizzare il database MySQL oppure il file fornito.

B.2.1 Importazione da database

Per importare dal database, aprire Weka Explorer e collegarsi all'istanza del database MySQL desiderata. Quindi, eseguire la *query* `weka/weka_data_import.sql`, che non fa altro che selezionare i primi 100 esempi (con ID sintomo uguale a 8) e 100 controesempi.

Selezionare le 200 righe ottenute e, in ordine:

1. Scegliere se eliminare o meno gli attributi `KTV` e `QB`.
2. Convertire da tipo `numeric` a `category` gli attributi `PATIENT_SEX`, `PATIENT_RACE` e `SYMPTOM_ID`.

I dati sono quindi pronti per essere analizzati.

B.2.2 Importazione da ARFF

Anziché selezionare il database come fonte dati, Weka Explorer permette di importare dati pre-caricati da file in formato `arff`.

I file disponibili per l'importazione, e pre-caricati come descritto in B.2.1 sono:

- `weka/symptom-8-raw.arff` contiene i dati grezzi, come se appena recuperati dalla base di dati. I dati devono essere filtrati come descritto in B.2.1.
- `weka/symptom-8-filtered.arff` contiene i dati giá filtrati, pronti all'uso come descritto in 8.3.1.

C. BAYESDB

C.1 Installazione di BayesDB

Per avviare una versione funzionante di BayesDB, si suggerisce di utilizzare la VirtualBox VM messa a disposizione sul sito ufficiale, in quanto contiene una versione giá configurata e pronta all'uso.
All'avvio della VM é giá possibile utilizzare BayesDB, essendo avviato in automatico.

C.2 Collegamento alla VM

La macchina virtuale su cui BayesDB é preinstallato non supporta la configurazione di tastiera italiana; pertanto é conveniente accedere alla *shell* Ubuntu tramite protocollo SSH.

Su sistemi Linux/Mac OS X, digitare in una shell:

```
$ ssh -i vm_guest_id_rsa -p 2222
  -o StrictHostKeyChecking=no bayesdb@localhost
```

Su sistemi Windows, `ssh` é incluso nella distribuzione di Git e puó essere utilizzato allo stesso modo.

C.3 Trasferimento da e verso BayesDB

Per inviare file dalla macchina locale alla VM (e viceversa), utilizzare l'eseguibile `scp` (secure `cp`) su sistemi Linux/Mac OS X:

```
$ scp -r -i vm_guest_id_rsa -p 2222
  -o StrictHostKeyChecking=no
  path/to/file bayesdb@localhost:/home/bayesdb/file
```

Su Windows, installare WinSCP e configurarlo con i seguenti parametri:

- server: `localhost`
- port: `2222`
- username: `bayesdb`
- password: `bayesdb`
- key: utilizzare la chiave `vm_guest_id_rsa`, fornita nel pacchetto della VirtualBox VM scaricato dal sito di BayesDB (ignorare gli avvisi di incompatibilitá della chiave con Putty)

Quindi, copiare i file desiderati dalla GUI di WinSCP.

C.4 Esecuzione di query

Dopo aver fatto accesso alla VM in SSH, aprire una console Python digitando il comando `python`, e ottenere un'istanza del client BayesDB:

```
>>> from bayesdb.client import Client
>>> client = Client()
```

A questo punto, si puó utilizzare `client` per eseguire qualsiasi *query*, ad esempio:

```
>>> client('INFER real_symptom_id, symptom_id '
...     'FROM dialysisai '
...     'WITH CONFIDENCE 0.9;')
```