

Previsione di Sintomatologie Post-Dialisi

Studio e realizzazione di un sistema supervisionato di estrazione regole

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

ABSTRACT

Implementazione Prolog dell'algoritmo di Intelligenza Artificiale C4.5, con k-fold test e confronto con altri sistemi pre-esistenti di classificazione.

1. OBIETTIVO

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

In questo modo, il medico può confermare la possibilità di occorrenza di una o più 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, anonimizzandole, dalla base di dati originale. Ai fini del processo di estrazione delle regole, è opportuno considerare il **sex** del paziente e la sua **età** al momento della seduta di dialisi in analisi¹.

¹Ciò non esclude la possibilità di considerare altri dati relati-

2.2 Parametri della seduta di dialisi

I parametri più rilevanti di una seduta di dialisi, al fine di prevedere eventuali sintomatologie successive, sono divisi in più categorie [3] [4].

A. L'efficienza della rimozione dei prodotti di scarto è 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 è 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

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 vi al paziente; l'algoritmo da realizzare potrebbe essere este-

fra la perdita di peso programmata e quella effettiva, a sua volta calcolata come differenza fra peso iniziale e peso finale.

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

Table 4: Parametri derivati

2.4 Sintomatologie

Il sistema verrà addestrato con istanze di esempio pre-classificate. La classificazione consiste nell'assegnazione, ad ogni esempio, di una o più categorie di sintomi, 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, è 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 ricca² e sono stati sottoposti ad una serie di passaggi³.

3.1 Creazione dei valori derivati

Per poter istanziare i valori degli attributi definiti in 2.3, è 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:

- **SESSION_ID**, l'ID della seduta di dialisi, utile per identificare la seduta in ogni momento
- **SESSION_DATE**, la data di esecuzione
- **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

so andando a considerare anche i dati relativi alle malattie pregresse del paziente ed eventuali comorbidità registrate.

²Circa dal 1999 ai primi mesi del 2014.

³Tutte le trasformazioni e selezioni di dati descritte in que-

- **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, è 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 singola 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; ciò si è 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).

In questa sezione sono codificate nel `scripts/01-sql-server-tables.sql`.

3.4 Migrazione dei dati

Per una gestione più libera dei dati, si è scelto di migrare le tabelle create da Microsoft SQL Server a MySQL⁴, anche in ottica futura (cfr. 9).

4. PULIZIA DEI DATI

Una volta spostati i dati su un database MySQL, si è scelto di eliminare alcuni record e mantenerne altri più rilevanti⁵. La base dati originaria, infatti, contiene 185476 record.

4.1 Calcolo dello score

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

Fissato il numero degli attributi (di dialisi) a 15, un record con *score* più elevato sarà selezionato con più probabilità per avviare il processo di apprendimento.

4.2 Pazienti rilevanti

Inoltre, lo *score* è stato utilizzato anche per poter eliminare, dai record già selezionati, tutti quelli che appartengono a pazienti che hanno meno di 5 rilevazioni di sintomi con uno *score* percentuale più 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 quindi i valori nulli); in un passo successivo, sono stati scansionati tutti i record e, qualora fosse rilevato un valore nullo, è stato inserito il valore medio relativo al paziente associato.

Tutto ciò, 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 è pertanto assunto che l'utente avesse erroneamente cancellato (dall'interfaccia del sistema) la dicitura "asintomatico", aggiungendo comunque una sintomatologia nulla (con ID uguale a 1).

5. APPRENDIMENTO DI REGOLE

L'obiettivo del sistema è 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.

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

⁵Gli script rilevanti sono contenuti nel file `scripts/04-`

5.1 Alberi di decisione

Nei sistemi TDIDT, il concetto è rappresentato in termini di un albero di decisione costruito in modalità top-down con una tecnica model driven: il sistema seleziona un attributo che caratterizza correttamente 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
- A insieme di attributi
- $\Lambda_A = \{a_1, \dots, a_r\}$ valori discreti che un attributo $A \in A$ può assumere

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

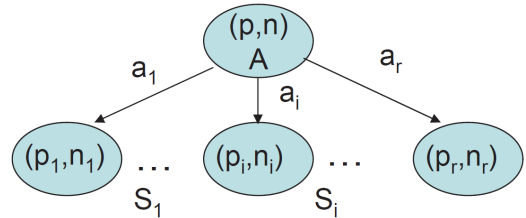


Figure 1: Esempificazione di un albero di decisione

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, 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 valore (o range di valori) dell'osservazione da classificare.

`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
 - rapporto di guadagno
 - guadagno di informazione normalizzato
 - 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 , \dots)

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_-):

$$\text{entropia}(S) = -p_+ \log_2 p_+ - p_- \log_2 p_-$$

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

- 0 se c'è minima entropia, ovvero se tutti gli esempi sono positivi
- 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 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:

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

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:

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

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

$\text{Gain}(S, A)$ 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

È 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.

Tuttavia, 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 è scelto di implementare il funzionamento base dell'algoritmo C4.5 di Ross Quinlan [6] [7], estensione del precedente ID3[5]. 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 è 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 stabilità del sistema, si è 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 stabile⁶, tuttavia, si è dovuto ricorrere ad altre tecniche per realizzare un classificatore che possa essere ritenuto valido.

6.2 K-Fold Cross-Validation

La **cross-validation** è 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 è 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 è diviso in 6 moduli, ognuno dei quali si occupa di parti differenti del programma⁷.

Si è scelto di utilizzare SWI-Prolog[1] come ambiente Prolog.

7.1 Utility

Sono stati realizzati 2 moduli che realizzano funzioni di utilità.

7.1.1 util.pl

util.pl contiene brevi regole che implementano:

⁶Al momento è in corso un procedimento per raccogliere dati da più centri di dialisi in tutta Italia, in modo tale da disporre di un insieme di esempi più 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.

⁷Ogni regola definita nei diversi moduli è stata documentata. La documentazione è 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

- **timer**, 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 più comune (**list_most_common**) e dell'indice di un elemento specifico (**index_of**), oltre che funzionalità di aggiunta 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).

7.2 Avvio del programma

Il programma principale è definito nel modulo **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 è il file di configurazione con il quale si vuole accedere al database.⁸. Vedi 7.3 per il l'accesso al database. Si è 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** 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 modalità **ask**) e **make_doc/0** (che genera la documentazione HTML).

Il sistema di logging implementato è, di default, molto verboso. Se non si vuole avere in output il dettaglio completo di ciò che sta avvenendo nel programma, è possibile utilizzare le regole messe a disposizione dal modulo di logging (vedi 7.1) per settare un livello di logging meno verboso:

```
?- log_level(info).
```

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

collegamenti ai moduli, dai quali si può comunque accedere alla documentazione delle regole.

⁸Alcuni file di configurazione di esempio sono presenti nella

```

% 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

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

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 modalit  `ask`, il file di configurazione verr  chiesto all'utente, e nel caso in cui non

cartella `prolog/config`.

⁹Per questa ragione, é necessario che sulla macchina client siano installati i driver di connessione ODBC al database target. Poich  i parametri sono impostati dal programma Prolog, non é necessario creare nessuna connessione sulla macchina.

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 propriet  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, salvando in memoria e stampando a video 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.

Poich  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 ottenere¹⁰. 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.

  stata realizzata anche la regola

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

dove `Type` pu  essere `positive` o `negative`. Tramite questa modalit  é possibile accedere a tutti gli esempi prelevati dalla base di dati. Sono presenti, inoltre, regole di conteggio e di verifica di esistenza (cfr. documentazione HTML).

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 quindi la tipologia per ogni attributo.

¹⁰La regola `get_records/2` prepara lo *statement* all'esecu-

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 già automaticamente partizionati, per cui una classe di tipo categorico avrà i *range* corrispondenti a tutti i valori di categoria
- ogni attributo di tipo `number` é stato suddiviso in 10 *range* di dimensioni uguali¹¹

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.

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

ziona.

¹¹L’approccio utilizzato é semplicistico, quindi passibile di

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*.

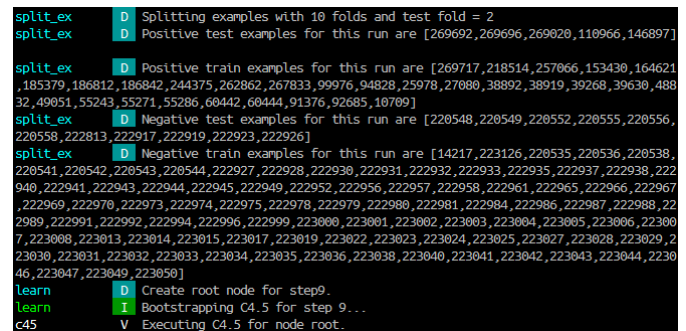
Nei primi test del programma, si é notato un bassissimo livello di distribuzione degli esempi nei diversi fold di ogni esecuzione, per cui i primi *run* avevano un bassissimo livello di *error rate* (spesso 0), che cresceva con l’aumentare dell’indice del passo.

Ciò 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.



```
split_ex D: Splitting examples with 10 folds and test fold = 2
split_ex D: Positive test examples for this run are [269692,269696,269820,118966,146897]
split_ex D: Positive train examples for this run are [269717,218514,257866,153438,164621,185379,186812,186842,244375,262862,267833,99976,94828,25978,27080,38892,38919,39268,39630,48832,49651,55243,55271,55286,60442,60444,91376,92685,10789]
split_ex D: Negative test examples for this run are [228548,228549,228552,228555,228556,228558,222813,222917,222919,222923,222926]
split_ex D: Negative train examples for this run are [14217,223126,228535,228536,228538,228541,228542,228543,228544,222927,222928,222930,222931,222932,222933,222935,222937,222938,222940,222941,222943,222944,222945,222949,222952,222956,222957,222958,222961,222965,222966,222967,222969,222970,222973,222974,222975,222978,222979,222980,222981,222984,222986,222987,222988,222989,222991,222992,222994,222996,222999,223000,223001,223002,223003,223004,223005,223006,223007,223008,223013,223014,223015,223017,223019,223022,223023,223024,223025,223027,223028,223029,223030,223031,223032,223033,223034,223035,223036,223038,223040,223041,223042,223043,223044,223046,223047,223049,223050]
learn D: Create root node for step9.
learn I: Bootstrapping C4.5 for step 9...
c45 V: Executing C4.5 for node root.
```

Figure 3: Suddivisione in 10 fold

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 così composti:

```
nnode(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.

miglioramento, vedi 9.

- **Parent** rappresenta il **node/4** genitore; per questa ragione, ogni nodo può 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** é 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 già 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 dovrà 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**:

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

```
c45-test V Non null different training targets left: [1,6]
c45 D Looking for the best attribute to split [ SAPStart : range(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 DAPDuration is 0.08844760214785355
info_gain V Info gain for SAPEnd is 0.1010777847191066
info_gain V Info gain for SAPAverage is 0.07794434474845499
info_gain V Info gain for DAPStart is 0.1010777847191066
info_gain V Info gain for DAPEnd is 0.17106214692568505
info_gain V Info gain for DAPAverag is 0.1010777847191066
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), cioè 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) ]
```


- **Node** il nodo appena creato
- **ParentNode** il nodo correntemente in analisi e in ingresso al passo corrente
- **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(-7.6,4.8000000000000001) ]
  ✓ [ DeltaBloodFlow : range(4.8000000000000001,17.200000000000003) ]
  ✓ [ 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(-7.6,4.8000000000000001) ]
      ✗ [ 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(4.8000000000000001,17.200000000000003) ]
        ✗ [ 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 la regola `print_le_tree/0`, una rappresentazione grafica e formattata dell'albero di de-

cisione prodotto, in cui ogni elemento foglia (positivo) avrà 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 stesso¹².

La regola principale che si occupa della generazione delle regole é `gen_all_the_rules/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`) 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 quantità di negativi non classificati (ovvero classificati come negativi, poiché il sistema genera regole per la sola classe positiva)
- **fn(FalseNegatives)**, cioè i positivi erroneamente classificati come negativi
- **tp(TruePositives)**, i positivi correttamente classificati
- **fp(FalsePositives)**, cioè 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 anche la *F₁-Measure*, equivalente a:

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

¹²Vengono generate regole per i soli nodi foglia positivi.

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 riasserte come `final_positive(ID)`, dove `ListaPassi` è la lista degli indici dei passi che hanno generato quella regola.

Il metodo che si occupa di unire le regole in un insieme singolo è `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 è 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 sul livello `info`:

- una ricapitolazione di tutte le informazioni asserite ad ogni esecuzione
- la media delle singole esecuzioni¹³
- l'esecuzione del test finale

7.6.4 Salvataggio log e regole

Al termine di ogni processo di apprendimento viene creato un file, `runs/log_{ID}.csv`¹⁴ contenente, per ogni esecuzione: il sintomo, il passo, il numero di regole generate e l'error rate; infine, il file riporta il tempo di esecuzione in secondi, il numero totale di esempi positivi presenti, il numero totale di regole generate e l'errore medio complessivo.

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,
```

¹³Come già detto, la media potrebbe essere poco indicativa per dataset molto piccoli.

¹⁴Per la creazione dei file di log, è necessario che la cartella `runs` esista già.

¹⁵Per la verifica delle regole è necessario che sia caricato in memoria Prolog anche la regola `check_condition_list/2`,

```
report I Learning algorithm finished in 2m 12s 0ms.
report I Symptom: 8
report I Positive examples: 100
report I Negative examples: 100
report I Total runs: 10
report I Runs recap:
report I - Run 1 | Rules : 22 | TP Rate : 1 | FP Rate : 0.09090909090909091 | Precision :
0.9166666666666666 | Recall : 1 | F-Measure : 0.9565217391304348
report I - Run 2 | Rules : 22 | TP Rate : 1 | FP Rate : 0.09090909090909091 | Precision :
0.9166666666666666 | Recall : 1 | F-Measure : 0.9565217391304348
report I - Run 3 | Rules : 23 | TP Rate : 0.9090909090909091 | FP Rate : 0 | Precision :
1 | Recall : 0.9090909090909091 | F-Measure : 0.9523809523809523
report I - Run 4 | Rules : 22 | TP Rate : 1 | FP Rate : 0 | Precision : 1 | Recall : 1 |
F-Measure : 1
report I - Run 5 | Rules : 21 | TP Rate : 1 | FP Rate : 0.09090909090909091 | Precision :
0.9166666666666666 | Recall : 1 | F-Measure : 0.9565217391304348
report I - Run 6 | Rules : 22 | TP Rate : 1 | FP Rate : 0 | Precision : 1 | Recall : 1 |
F-Measure : 1
report I - Run 7 | Rules : 21 | TP Rate : 0.5454545454545454 | FP Rate : 0 | Precision :
1 | Recall : 0.5454545454545454 | F-Measure : 0.7058823529411764
report I - Run 8 | Rules : 21 | TP Rate : 0.9090909090909091 | FP Rate : 0 | Precision :
1 | Recall : 0.9090909090909091 | F-Measure : 0.9523809523809523
report I - Run 9 | Rules : 20 | TP Rate : 0.7272727272727273 | FP Rate : 0 | Precision :
1 | Recall : 0.7272727272727273 | F-Measure : 0.8421052631578948
report I - Run 10 | Rules : 12 | TP Rate : 0.7 | FP Rate : 0.1 | Precision : 0.875 | Rea
ll : 0.7 | F-Measure : 0.7777777777777777
report I AVERAGES:
report I - TP Rate: 0.8790909090909091
report I - FP Rate: 0.03727272727272727
report I - Precision: 0.9625
report I - Recall: 0.8790909090909091
report I - F-Measure: 0.9100092516030059
test I FINAL:
test I - TP: 100
test I - TN: 97
test I - FP: 3
test I - FN: 0
test I - TP Rate: 1
test I - TN Rate: 0.97
test I - FP Rate: 0.03
test I - FN Rate: 0
test I - Precision: 0.970873786407767
test I - Recall: 1
test I - F-Measure: 0.9852216748768473
test I - Rules: 44
report D Report printed.
save_log D Tests written to file.
save_rules D Rules written to file.
1 7-
```

Figure 6: Stampa del report di fine processo

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

8. RISULTATI

Il programma è stato eseguito su 9 sintomi target distinti e con un diverso numero di esempi positivi¹⁶.

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 tabella¹⁷ sono:

- ID del sintomo analizzato

che serve ad eseguire il *matching* dei range descritti dalla regole generate.

¹⁶La 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

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

- 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, però, si è 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 è 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 più generali che coprano almeno gli stessi esempi di una o più regola specifica, senza introdurre grosse variazioni di precisione¹⁸.

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 allo 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.

¹⁸In tal senso, sarebbe utile estendere il programma con una funzionalità 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 Altri sistemi

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

8.3.1 Weka

Il tool di Data Mining utilizzato è Weka[2], 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.

Una volta importati i dati del dataset di esempio in Weka Explorer (si veda A 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 modalità di test e avviando la classificazione per il sintomo target 8, sono stati ottenuti i risultati mostrati in tabella 7.

L'output di esecuzione restituito¹⁹ è 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)
```

¹⁹L'output non è completo; per l'output completo, vedere il file **weka/**

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*[8], 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 modalità di testing con 10-fold cross-validation, l'output principale ottenuto é il seguente:

=== 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 più 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:

=== 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).

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

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 a

```
(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 tutti gli esempi che la generavano si trovavano, evidentemente, nel sottoinsieme di testing.

8.3.2 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 cardinalità di ogni range non sarebbe piú fissa, ma varierebbe a seconda della vicinanza dei valori degli esempi o secondo altri criteri.

Come già 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 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). In questo modo sarebbe anche possibile definire un ordine di priorità per le regole generate, da quella "piú certa" a quella con la piú bassa probabilità.

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 già in studio una migrazione su Google Compute Engine
- apprendimento multi-threading sui diversi split dei *k-fold*
- 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 all'*error rate* del passo in cui sono state apprese (diretta proporzionalità tra ordine regola e ordine *error rate*) e alla loro semplicità (minor numero di clausole, maggiore rilevanza)

10. REFERENCES

- [1] Swi-prolog, Apr. 2014.
- [2] Swi-prolog, Apr. 2014.
- [3] 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.
- [4] 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.
- [5] T. M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.
- [6] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.
- [7] 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.

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

APPENDIX

A. WEKA