Implementazione Prolog dell'algoritmo Alternating Decision Tree

Scianatico Lorenzo

3 luglio 2012

Indice

Introduzione

In questo studio viene presentato un algoritmo di apprendimento noto come Alternating Decision Tree, o più semplicemente, ADTree. Questo algoritmo è noto in letteratura per via di alcune caratteristiche come la sua connessione al Boosting e la sua capacità di generalizzare altri algoritmi di apprendimento.
Questo lavoro quindi si pone nell'ambito della specifica sotto-area dell'Intelligenza Artificiale nota come Apprendimento Automatico. Di recente infatti sono stati molti i nuovi traguardi raggiunti nell'ambito di tale disciplina, producendo una grande quantità di algoritmi in grado di apprendere conoscenza da un insieme di dati, sfruttando anche approcci tra loro notevolmente differenti. In particolare si possono citare numerosi strumenti che fanno uso di tecniche sub-simboliche, come Reti Neurali e Support Vector Machine, che pur avendo diversi vantaggi, hanno la nota negativa di essere scarsamente interpretabili, o in alcuni casi non interpretabili.
Tra le tecniche simboliche invece vanno citati gli alberi di decisione, che al momento è uno strumento non solo efficace ma anche interpretabile. L'algoritmo che sarà discusso in questo caso di studio è di fatto un particolare algoritmo basato su alberi di decisione messo a punto da Yoav Freund e Llew Mason, e in seguito chiarito da Bernhard Pfahringer, Geoffrey Holmes and Richard Kirkby.
Il presente lavoro è stato svolto nell'ambito del corso di Intelligenza Artificiale, tenuto dalla professoressa Esposito Floriana e dal professor Ferilli Stefano dell'Università degli studi di Bari "Aldo Moro".

Capitolo 1

Alberi di decisione

1.1 Introduzione agli alberi di decisione

Con il nome di apprendimento di alberi di decisione si identificano i metodi per l'apprendimento di funzioni a valore discreto in cui la funzione è rappresentata (anche graficamente) da un albero di decisione. Questi metodi sono inclusi tra i più efficienti ed efficaci algoritmi di inferenza induttiva. Sono stati applicati con successo ad una vasta gamma di casi, a partire dai casi clinici sino a casi di tipo economico-finanziario, o anche alla classificazione di giornate ideali per giocare a tennis.

Oltre che per via grafica, gli alberi di decisione possono essere rappresentati anche come un insieme di regole if-then, che consente una interpretazione relativamente semplice.
L'interpretabilità degli alberi di decisione è data dalla loro struttura. Si parte da un nodo principale, detto radice (o anche *root*) dell'albero. Tutti gli altri

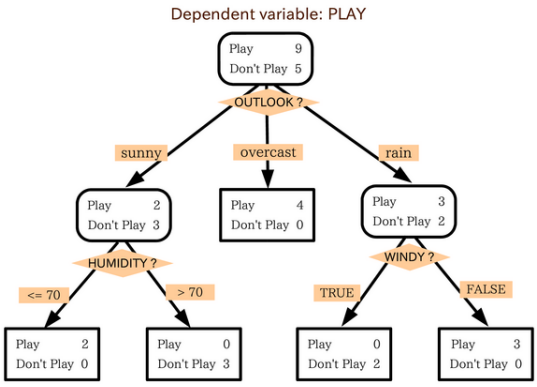

Figura 1.1: Albero di decisione

nodi sono posti nell'albero nei vari sotto-alberi del nodo principale, sino a giungere ai nodi senza figli, detti foglie dell'albero. In ogni nodo viene memorizzata una regola. Nel momento in cui si decide di classificare un'istanza utilizzando l'albero di decisione un singolo attributo dell'istanza viene confrontato con la regola contenuta nel nodo. In base alla risposta si procede nella visita dell'albero, oppure ci si ferma se si è raggiunta una foglia, portando in output una risposta. L'istanza quindi, a partire dal nodo radice effettua una visita di un solo path all'interno dell'albero.
In generale un albero di decisione può essere considerato come una disgiunzione di congiunzioni di vincoli sui valori degli attributi di una certa istanza da classificare. Ogni path è una congiunzione di test sugli attributi, mentre tutto l'albero è una disgiunzione dei vari path.
Noti algoritmi di alberi di decisione sono C4.5, C5.0, ID3 e CART.

1.2 Problemi appropriati per gli alberi di decisione

Generalmente è possibile tracciare delle caratteristiche per stabilire per quali problemi è appropriato l'uso di alberi di decisione. Le principali sono:

- Le istanze sono rappresentate come coppia di attributo e valore, e sono descritte da un insieme fissato di attributi e il corrispettivo valore. La situazione più semplice da gestire è quella in cui ogni attributo copre un piccolo numero di valori disgiunti. Esistono tuttavia delle estensioni per gestire anche attributi a valore reale.

- La funzione da apprendere ha valori discreti. Ad esempio, la funzione da apprendere classifica in modo binario gli esempi. Esiste tuttavia la possibilità di estendere le funzioni a più di due valori di output possibili.

- Possono essere richieste descrizioni disgiuntive. Come precisato in precedenza, gli alberi di decisione rappresentano per loro natura espressioni in disgiunzione tra loro.

- Il training set può contenere errori. I metodi basati sugli alberi di decisione si mostrano robusti agli errori, sia quelli di classificazione negli esempi per il training che quelli dei singoli attributi che descrivono gli esempi.

- Il training set contiene valori mancanti. Gli alberi di decisione possono comunque operare anche in presenza di dati mancanti.

A:	10	15	21	28	32	40	50
Class:	No	Yes	Yes	No	Yes	Yes	No
Threshold:		12.5		24.5	30		45

Figura 1.2: Esempio di ricerca delle soglie

Leggendo queste caratteristiche risulta chiaro come, di fatto, gli alberi di decisione possano essere applicati a moltissimi problemi di natura pratica. In particolare, si evince come si possano applicare a problemi di classificazione. Tuttavia bisogna fare molta attenzione nella costruzione di un albero, poiché gli alberi di decisione (di per sé NP-completi) talvolta possono creare modelli troppo complessi che perdono la capacità di generalizzare correttamente, ovvero problema di overfitting. Per risolvere tale problema si ricorre a tecniche di potatura.

1.3 Incorporare valori continui

Come descritto in precedenza, gli alberi sono definiti per operare su valori discreti. In primo luogo, il valore che viene predetto deve essere discreto. In secondo luogo, l'attributo testato nel nodo dell'albero deve essere discreto. questa seconda restrizione può essere rimossa, in modo da poter fare uso di valori continui. L'obiettivo si può ottenere definendo dinamicamente nuovi valori discreti che partizionano l'attributo a valore continuo in un insieme di valori discreti. Nel dettaglio, per un attributo A a valore continuo l'algoritmo deve creare dinamicamente una nuova condizione booleana, tale che la condizione sia vera per $A < c$ e falsa negli altri casi. Il problema diventa quindi quello di selezionare la migliore soglia per il valore c.
Per trovare queste soglie è necessario ordinare gli esempi rispetto all'attributo continuo, calcolando la media dei valori continui in ogni cambio di label degli esempi, come mostrato in figura 2. Questa procedura fornisce in output tutte le soglie candidate, tra cui scegliere quella più appropriata per la creazione del nodo. Il criterio per tale scelta varia in base all'algoritmo per indurre l'albero. Ad esempio, l'algoritmo di C4.5 fa uso della variazione dell'entropia per stabilire quale nodo aggiungere all'albero di decisione in tutti i casi, non solo per attributi continui, mentre CART fa uso dell'indice di Gini. A questi si aggiungono altri metodi di ottimizzazione e di potatura dei nodi superflui. Nel caso dell'Alternating Decision Tree il criterio è differente e sarà discusso in seguito.

1.4 Vantaggi e svantaggi

Riassumendo, i vantaggi degli alberi di decisione sono:

- Semplicità di interpretazione, che a volte non richiede neanche particolari spiegazioni;
- Ridotta preparazione dei dati;
- Possibilità di gestire sia dati numerici che categorici;
- Utilizzo di un modello white-box;
- Possibilità di validazione dei modelli tramite test statistici;
- Robustezza;
- Rapidità di analisi anche per grandi quantità di dati.

D'altra parte, gli svantaggi sono:

- Np-completezza dell'apprendimento di un modello ottimale. Di conseguenza, gli alberi di decisione fanno uso di algoritmi greedy e strategie euristiche localmente ottimali ma che non garantiscono un albero che risulti globalmente ottimale.
- Facilità di overfitting. Per questo esistono diverse tecniche di potatura degli alberi e criteri di stop.
- Esistono concetti che non vengono espressi facilmente dagli alberi di decisione, e che quindi risultano difficili da apprendere, come ad esempio lo XOR. Per questi concetti è bene fare uso di altre tecniche più adatte, come tecniche di apprendimento statistico o programmazione logica induttiva.
- Per dati che includono variabili categoriche con differenti livelli il guadagno di informazione può essere distorto.

Capitolo 2

Boosting

2.1 L'idea di base

Si pensi ad uno scommettitore inesperto di corse ippiche, o calcistiche, o di un evento sportivo generico. Per aumentare le sue possibilità di vittoria potrebbe pensare di rivolgersi ad uno scommettitore più esperto di lui, chiedendogli quali potrebbero essere delle linee guida per poter scegliere su chi scommettere ottenendo il massimo dei profitti. Tuttavia, difficilmente uno scommettitore esperto potrebbe dare delle regole con cui scommettere sempre, a meno di non avere dei dati più o meno recenti alla mano. In tal caso, può dare dei buoni suggerimenti in base al recente andamento delle ultime competizioni. Di conseguenza il problema diventa in primo luogo quello di scegliere i dati da presentare, e in secondo luogo quello di capire come combinare le regole fornite volta per volta dallo scommettitore esperto in modo da avere un insieme unico di regole su cui basarsi.
Lo scopo del Boosting è proprio questo: ottenere, dato un insieme di regole di predizione non molto accurate, un insieme di regole molto accurate e precise, proprio come suggerito dall'esempio dello scommettitore. Di fatto il proposito degli algoritmi di Boosting è quello di prendere un insieme di weak learner per creare uno strong leaner. I primi algoritmi di Boosting sono stati implementati per essere sperimentati sugli OCR.
Tra gli algoritmi più importanti va citato AdaBoost.

2.2 AdaBoost

L'algoritmo AdaBoost (contrazione di Adaptive Boosting)è uno degli algoritmi più importanti di Boosting, ed è stato realizzato da Yoav Freund e Robert Shapire. Nel 2003 entrambi sono stati premiati con il premio Gödel per il

loro lavoro su tale algoritmo.
Introdotto nel 1995, AdaBoost ha risolto numerosi problemi dei primi algoritmi di boosting.

Siano dati: $(x_1, y_1), \ldots, (x_m, y_m)$ con $x_i \in X, y_i \in Y = \{-1, +1\}$

Inizializzare $D_1(i) = 1/m$.

Per $t = 1, \ldots, T$:

- Allenare ogni weak learner usando la distribuzione D_t.
- Considerare tutte le ipotesi deboli $h_t : X \to \{-1, +1\}$ con un errore pari a:
$$\epsilon_t = Pr_{i \sim D_t} [h_t(x_i) \neq y_i]$$
- Scegliere $\alpha = \frac{1}{2} ln \left(\frac{1 - \epsilon_t}{\epsilon_t} \right)$.
- Aggiornare:
$$D_{t+1}(i) = \frac{D_t(i)}{Z_t} \times \begin{cases} \exp^{-\alpha_t}, & \text{se } h_t(x_i) = y_i \\ \exp^{\alpha_t}, & \text{se } h_t(x_i) \neq y_i \end{cases} = \frac{D_t(i) exp(-\alpha_t y_i h_t(x_i))}{Z_t}$$
dove Z_t è un fattore di normalizzazione, tale che D_{t+1} sia una distribuzione.

L'ipotesi finale:
$$H(x) = sign \left(\sum_{t=1}^{T} \alpha_t h_t(x) \right).$$

L'algoritmo considera un training set $(x_1, y_1), \ldots, (x_m, y_m)$ dove ogni x è un'istanza, ogni y è una label (nel caso di classificazioni binarie $+1$ o -1) ed m è la numerosità del campione. Iterativamente, per un numero T di ripetizioni, l'algoritmo impara delle regole più semplici, ovvero le cosiddette ipotesi deboli. L'idea principale è di mantenere una distribuzione dei pesi sul training set. I pesi vengono denotati con $D_t(i)$ e vengono inizializzati tutti allo stesso modo, in base alla numerosità del campione o alternativamente ad uno, e ad ogni iterazione i pesi degli esempi mal classificati vengono incrementati in modo da cambiare il focus dei weak learner. Il compito è quello quindi di trovare una ipotesi debole appropriata per la distribuzione D_t. La bontà di ogni ipotesi viene misurata con il suo errore ϵ_t.
Il passo successivo è quello di scegliere un parametro α_t che, intuitivamente,

misura l'importanza dell'ipotesi h_t. Dalla formula si evince come questo parametro risulta essere maggiore nel momento in cui l'errore ϵ_t è più piccolo.
Seguendo la regola della procedura riportata, la distribuzione D_t viene aggiornata. L'effetto è di far crescere il peso degli esempi non classificati correttamente da h_t e di ridurre il peso degli esempi ben classificati.
L'ipotesi finale H è un voto pesato a maggioranza delle T ipotesi deboli, dove α_t è il peso assegnato ad ogni h_t. Il Boosting con AdaBoost può essere visto come una minimizzazione di una funzione a perdita convessa su un insieme convesso di funzioni. In particolare, la perdita che viene minimizzata è la perdita esponenziale:

$$\sum_i e^{-y_i f(x_i)}$$

cercando una funzione:

$$f(x) = \sum_t \alpha_t h_t(x)$$

Si dimostra che l'errore generalizzato per AdaBoost è:

$$\hat{Pr}[margin(x,y) \leq \theta] + \tilde{O}\left(\sqrt{\frac{d}{m\theta^2}}\right)$$

con il margine che viene calcolato per ogni esempio (x, y) come:

$$\frac{y \sum_t \alpha_t h_t(x)}{\sum_t \alpha_t}$$

L'algoritmo è generalizzabile a classificazioni multiclasse.

2.3 Relazione con le SVM

Si supponga di aver trovato le ipotesi debole che si desidera combinare, e di essere interessati solo ai coefficienti α_t. Un approccio ragionevole suggerito da AdaBoost è di scegliere i coefficienti in modo che l'errore ne risulti minimizzato. Si supponga quindi pari a zero il primo termine e si ponga l'attenzione sul secondo termine, dato che si sta tentando di massimizzare il margine minimo di ogni esempio. Si denoti il vettore di ipotesi deboli con

$\mathbf{h}(x) \doteq \langle h_1(x), h_2(x), \ldots, h_N(x)\rangle$ che sarà detto *vettore di istanze* e il vettore di coefficienti $\boldsymbol{\alpha} \doteq \langle \alpha_1, \alpha_2, \ldots, \alpha_N\rangle$ che sarà detto *vettore dei pesi*. A questo punto l'obiettivo di massimizzazione del minimo margine diventa:

$$\max_{\boldsymbol{\alpha}} \min_{i} \frac{(\boldsymbol{\alpha} \cdot \mathbf{h}(x_i))y_i}{\|\boldsymbol{\alpha}\|\|\mathbf{h}(x_i)\|}$$

Nel caso del boosting, le norme sono definite come:

$$\|\boldsymbol{\alpha}\|_1 \doteq \sum_t |\alpha_t|, \quad \|\mathbf{h}(x)\|_\infty \doteq \max_t |h_t(x)|$$

Nel caso della classificazione binaria la seconda norma è sempre uguale ad 1. Questo obiettivo somiglia molto a quello delle SVM (Support Vector Machine), dove le norme sono tutte euclidee:

$$\|\boldsymbol{\alpha}\|_1 \doteq \sqrt{\sum_t \alpha_t^2}, \quad \|\mathbf{h}(x)\|_\infty \doteq \sqrt{\sum_t h_t(x)^2}$$

Questa descrizione fa sembrare AdaBoost e le SVM molto simili tra loro. In realtà ci sono molte differenze importanti tra le due tecniche:

- *Norme differenti forniscono margini differenti*:La differenza delle norme può essere molto poco significativa per spazi dimensionali ridotti. Tuttavia, sia per AdaBoost che per le SVM le dimensioni sono molto ampie. La conseguenza è che le norme conducono a risultati differenti, specie se solo alcune delle variabili sono rilevanti. Ad esempio, si supponga che le ipotesi deboli abbiano tutte range binario, e che le label y sugli esempi possano essere tutte calcolate a votazione di maggioranza con k delle ipotesi deboli. Si può dimostrare che tali ipotesi sono solo una frazione ridotta del numero totale delle ipotesi, e che quindi il margine associato ad AdaBoost sarà più grande del margine associato dalle macchine a vettori di supporto.

- *I requisiti computazionali sono differenti*: i calcoli coinvolti in entrambi i casi sono massimizzazioni di espressioni matematiche dato un insieme

di ineguaglianze. Mentre le SVM corrispondono alla programmazione quadratica, AdaBoost corrisponde alla *programmazione lineare*. Di fatto questo connette AdaBoost anche alla teoria dei giochi.

- *Approcci differenti per la ricerca efficiente in spazi dimensionali ampi*: Parte della ragione dell'efficienza delle SVM e di AdaBoost risiede nel fatto che cercano entrambi classificazioni lineari per spazi dimensionali estremamente ampi, potenzialmente infiniti. Le prime risolvono il problema attraverso dei metodi *kernel* che consentono di fare calcoli in spazi dimensionali ridotti equivalenti a prodotti scalari di grandi dimensioni. AdaBoost invece procede con una procedura *greedy*: il leaner debole funge da oracolo per trovare le coordinate giuste per le ipotesi deboli, mentre l'aggiornamento dei pesi guida il leaner verso altre coordinate correlate. La maggior parte del lavoro per migliorare questi metodi risiede nel selezionare la funzione kernel più appropriata per le SVM e nell'uso dell'algoritmo di apprendimento debole per AdaBoost. Questi algoritmi sono tra loro differenti sia nel loro funzionamento che nell'operare in spazi tra loro differenti, e di conseguenza anche i classificatori generati sono molto diversi tra loro.

Capitolo 3

Alternating Decision Tree

Introdotti brevemente la teoria degli alberi di decisione e del Boosting ora è possibile introdurre l'algoritmo di apprendimento di Altrenating Decision Tree. Tale algoritmo è stato messo a punto da Yoav Freund e da Llew Mason, e combina AdaBoost con i tradizionali alberi di decisione, generalizzando differenti tipi di algoritmi di apprendimento.

3.1 Introduzione

In seguito alla sua presentazione, l'algoritmo AdaBoost ha conosciuto un notevole successo, soprattutto combinato con gli alberi di decisione e con i decision stumps. Questi ultimi sono casi particolari di alberi di decisione composti da una sola regola.
Un altro ottimo pacchetto software di questo genere è C5.0, che combina una variante di AdaBoost con C4.5, ottenendo un classificatore molto preciso, ma molto spesso ampio, pesante, e di difficile interpretazione. Per questi motivi Freund e Mason hanno messo a punto questo algoritmo, l'ADTree, che combina in un nuovo modo gli alberi di decisione, i decision stump e il boosting in un nuovo tipo di classificatore, molto più piccolo e più semplice da interpretare. Tale algoritmo ha dimostrato di riuscire, in molti casi, ad eguagliare le capacità di C5.0.

3.2 ADTree: definizione e algoritmo

I Decision Stump sopra accennati non sono altro che semplici alberi di decisione di un solo livello, come riportato nell'immagine 3.1, eventualmente con più nodi figli.

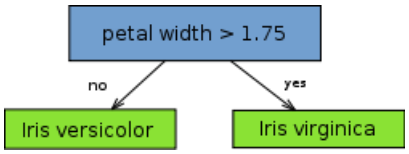

Figura 3.1: Decision stump

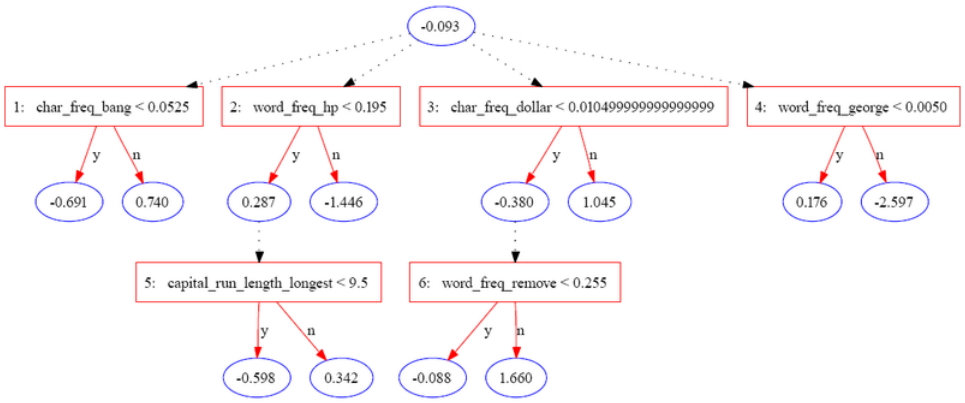

Figura 3.2: ADTree

Questi semplici alberi ad un livello sono particolarmente adatti ad essere combinati con AdaBoost in quanto costituiscono un insieme perfetto di learner deboli di coi necessita il boosting per funzionare. Il sistema di riconoscimento facciale Viola-Jones ad esempio utilizza la combinazione AdaBoost-Decision Stump con ottimi risultati. Tale framework è stato implementato in diversi linguaggi con successo.
L'ADTree è una generalizzazione di decision stump e alberi di decisione tradizionali. Anziché avere un nodo con due figli ogni nodo può avere un numero arbitrario di figli. Inoltre ogni nodo è costituito da due parti: una parte detta di *split* e una detta di *predizione*. Mentre uno split corrisponde ad un normale nodo di decisione contenente una regola, i nodi di predizione, due per ogni split, contengono un valore reale. Il nodo radice è costituito invece dal solo nodo di predizione. Come per un albero di decisione classico, l'ADTree costituisce una mappatura di istante in una di due classi, -1 e $+1$, negativo e positivo. Tuttavia, a differenza di un tradizionale albero di decisione, dove la classificazione è associata ad un percorso dell'albero, qui la classificazione è associata a più percorsi e alla somma di tutti i valori presenti nei percorsi che l'istanza ha percorso nell'albero. Inoltre l'istanza non attraversa solo un percorso all'interno dell'albero, ma può attraversare più percorsi contemporaneamente, come si evince dalla figura 3.2.

In modo abbastanza semplice è possibile trasformare un qualunque albero di decisione in un ADTree, aggiungendo i nodi di predizione. Naturalmente i nodi di predizione e il valore contenuto vanno aggiunti con un certo criterio. I tradizionali alberi di decisione definiscono una partizione dello spazio delle istanze in regioni disgiunte, solitamente con partizioni binarie iterative. Ogni parte viene divisa una volta. questo corrisponde allo split di nodi foglia. In generale invece per un ADTree ogni parte può essere suddivisa più volte. In seguito un'istanza percorre l'albero in tutti i path possibili, fino alle foglie, raccogliendo tutti i valori dei nodi di predizione interessati. Il segno della sommatoria di tutti questi valori dà l'output dell'albero.
Da questo si capisce come gli ADTree siano una generalizzazione degli alberi di decisione. Supponiamo di avere un ADTree con tre livelli: radice, set di nodi di decisione, e i nodi di predizione. Questo mostra come gli ADTree siano una generalizzazione dei decision stump.

Con una semplice regola si può esprimere ogni nodo dell'ADTree:

if precondizione **then**
  if condizione **then**
    output p1
  else
    output p2
  end if
end if

Prima di definire la procedura per la creazione di un ADTree è necessario definire alcune funzioni ausiliarie:

- $W_+(c)$ restituisce il peso di tutti gli esempi marcati come positivi che soddisfano la condizione c.
- $W_-(c)$ restituisce il peso di tutti gli esempi marcati come negativi che soddisfano la condizione c.
- $W(c) = W_+(c) + W_-(c)$.

La procedura è la seguente:

Inizializzazione: il peso di ogni istanza viene settato ad uno. Si inizializza l'insieme di regole $\mathcal{R}$ con una regole con condizione e precondizione entrambe vere. Il valore di predizione di tale regola è:

$$a = \frac{1}{2} ln \frac{W_+(c)}{W_-(c)}$$

con valore c uguale a True.

Preadattamento: si aggiorna il peso di ogni istanza secondo la formula:

$$w_{i,1} = w_{i,0} e^{-a y_t}$$

con y_t uguale ad 1 o -1 in base alla classificazione.

Iterando per $t = 1, 2, \ldots, T$:

- Si genera il set di condizioni $\mathcal{C}$.
- Per ogni precondizione $c_1 \in \mathcal{P}_t$ e ogni condizione $c_2 \in \mathcal{C}$ calcolare: $Z_t(c_1, c_2) = 2\left(\sqrt{W_+(c_1 \wedge c_2) W_-(c_1 \wedge c_2)} + \sqrt{W_+(c_1 \wedge \neg c_2) W_-(c_1 \wedge \neg c_2)}\right) + W(\neg c_1)$
- Selezionare c_1, c_2 tali che minimizzino $Z_t(c_1, c_2)$ e settare $\mathcal{R}_{t+1}$ come $\mathcal{R}_t$ con l'aggiunta di r_t la cui precondizione sia c_1 e condizione sia c_2 e i cui valori di predizione sono pari a:

$$a = \frac{1}{2} ln \frac{W_+(c_1 \wedge c_2) + 1}{W_-(c_1 \wedge c_2) + 1} \qquad b = \frac{1}{2} ln \frac{W_+(c_1 \wedge \neg c_2) + 1}{W_-(c_1 \wedge \neg c_2) + 1}$$

- Settare $\mathcal{P}_{t+1}$ come $\mathcal{P}_t$ con l'aggiunta di $c_1 \wedge c_2$ e $c_1 \wedge \neg c_2$.
- Aggiornare i pesi delle istanze secondo la formula:

$$w_{i,t+1} = w_{i,t} e^{-r_t(x_i) y_t}$$

Output: l'output della classificazione è, considerando tutte le regole di $\mathcal{R}_{T+1}$:

$$class(x) = sign\left(\sum_{t=1}^{T} r_t(x)\right)$$

L'articolo originale di Freund e Mason conteneva alcuni errori di battitura che compromettono il corretto funzionamento dell'algoritmo. La versione riportata è quella che in seguito riportarono Bernhard Pfahringer, Geoffrey Holmes and Richard Kirkby nel loro articolo, che risulta corretta e che ha un leggero miglioramento nell'aggiunta dell'uno, sia a denominatore che a numeratore, al calcolo dei due valori di predizione, al fine di evitare presenza di zeri nelle frazioni. La fase di preadattamento nell'articolo originale invece

era considerata come implicita.

Il training set è denotato come $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)$ con $\mathbf{x}_i \in R^d$ e $y_i \in \{-1, +1\}$. L'insieme delle precondizioni $\mathcal{P}_t$ cresce di due elementi per iterazione, mentre quello delle regole $\mathcal{R}_t$ cresce di una regola per volta.
Si noti anche come i nodi possano essere aggiunti in un punto arbitrario dell'albero e non solo ai nodi foglia, e come il criterio per la scelta non sia il guadagno di informazione (C4.5) i l'indice di Gini (CART) ma il minimo di una funzione.

In ultimo: è assente un criterio di stop. Dagli esperimenti condotti da chi ha testato l'algoritmo è emerso che il modello va sicuramente in overfitting se il numero di iterazioni del processo di boosting è relativamente alto rispetto al dataset. In alcuni casi possono essere sufficienti anche poco più di dieci iterazioni per mandare il modello in overfitting. Al momento non si dispone di un criterio di stop appropriato. Non si dispone ancora di una spiegazione per questo comportamento, e per ora l'unico consiglio è quello di stabilire empiricamente il numero di iterazioni per la generazione di regole.

3.3 Interpretare gli ADtree

L'interpretazione di un albero di decisione classico è un'operazione relativamente banale. Uno dei vantaggi infatti degli alberi di decisione è l'estrema semplicità di lettura, che richiede al più un minimo di dimestichezza con diagrammi, eventualmente senza particolari competenze. Ancora più semplice nel caso di decision stump che altro non sono che dei piccoli alberi.
Il discorso invece cambia per gli ADTree, che inizialmente possono risultare un po' più complicati rispetto ai tradizionali alberi di decisione. Per questo motivo vengono fornite delle semplici linee guida per interpretare gli ADTree, mostrando come nonostante la struttura notevolmente differente del modello si tratti di un compito relativamente semplice.
L'articolo originale di Freund e Mason riporta come il pacchetto software C5.0 e ADTree generino, sullo stesso dataset, il primo un albero di ben 446 nodi e il secondo un albero di soli 6 nodi! Per cui un ADTree tende comunque a fornire in output un modello più semplice e quindi più leggibile.
In primo luogo, il contributo di ogni singolo nodo può essere interpretato singolarmente. Infatti è sufficiente verificare i valori di predizione per capire come le singole caratteristiche possano essere influenti sulla classificazione.
Un ulteriore interpretazione può essere effettuata considerando la posizione dei nodi. dato che i nodi si aggiungono in un punto qualunque dell'albero e che i nodi possono avere più di un figlio a destra o a sinistra, l'effetto che si

ottiene è quello di più nodi su di uno stesso livello. Inoltre i nodi vengono etichettati anche con un numero al momento dell'aggiunta, numero che indica non solo l'iterazione in cui sono stati aggiunti ma anche l'importanza della decisione. Se i nodi sono su di uno stesso livello allora l'interazione tra le caratteristiche coinvolte è minima, se non nulla. Di fatto ognuno di questi nodi condurrà a strade differenti nell'analisi di un'istanza da classificare. Questo tipo di informazione è rappresentabile anche con un albero di decisione classico, ma con un numero di nodi maggiore. In netto contrasto con quanto appena detto, la significatività dei nodi ai livelli inferiori va valutata in relazione a quella dei nodi padre. Questa relazione infatti mette in evidenza una forte relazione tra le caratteristiche coinvolte. Questo genere di informazione viene regolarmente rappresentata nei tradizionali alberi di decisione ma è assente nei decision stump.
Il nodo radice contiene una classificazione preliminare per il modello. Il segno del nodo dice, in considerazione degli esempi, quale delle due categorie sia predominante e quale sia la classificazione a priori che darà alle istanze, ma con un basso livello di confidenza.
Infine, l'albero può essere interpretato anche nel complesso. Man mano che il modello viene visitato, si accumula evidenza per una o l'altra classificazione, sommando volta per volta il punteggio. Idealmente, se il punteggio parziale dovesse diventare molto alto in valore assoluto, tanto da non poter cambiare di segno, si potrebbe anche fare in modo di fermare la visita dell'albero. Lo stesso valore assoluto della somma può essere considerato come un valore ulteriore per valutare l'affidabilità della classificazione.
Infine va anche considerato che i nodi hanno nel complesso una influenza limitata. Questo aggiunge robustezza al modello, in quanto se si dovesse interrogare l'albero con un'istanza contenente missing values l'affidabilità della classificazione non viene compromessa. Se un valore dovesse mancare tutti i nodi che analizzano quel valore (ed eventuali figli) vengono tralasciati. Naturalmente questo riduce leggermente l'affidabilità della classificazione, ma non la pregiudica, a meno di un eccessivo numero di valori mancanti.

3.4 Implementazione

Il linguaggio scelto come da specifiche per l'esame è il Prolog. Quello della programmazione logica è particolarmente adatto come paradigma ai problemi di apprendimento, in particolare sugli alberi di decisione. L'intenzione infatti è quella di apprendere delle regole e di mantenerle sotto forma di albero. Per questi motivi, l'ambiente completamente interpretato e dinamico offerto da Prolog è particolarmente adatto a tali problemi. Naturalmente questo

richiede una certa familiarità sia con la logica che con il linguaggio. Costrutti tipici più di un approccio procedurale, quali iterazioni e ricerca di minimi, possono comunque essere ottenuti con i meccanismi di *cut* di Prolog e altri comandi, appoggiandosi alle liste, la struttura dati più gestibile con Prolog. L'albero è stato implementato come un insieme di fatti, in quanto ogni nodo può avere un numero arbitrario di figli. Ogni nodo contiene informazioni sul nodo padre, la regola di decisione contenuta, e i valori di predizione. L'unica eccezione è la radice che è costituita da una regola sempre vera e dal valore di predizione singolo.
L'algoritmo di ordinamento scelto per cercare le soglie numeriche è il Quicksort. La scelta è motivata dal fatto che tale algoritmo è ricorsivo per sua natura e la ricorsione è un concetto fondamentale in Prolog. Inoltre, per migliorare l'efficienza, può essere adottata la rappresentazione con liste differenza.
Sempre per ragioni di efficienza e per evitare un eccessivo spreco di memoria, il minimo della funzione zeta viene calcolato volta per volta considerando solo ogni valore in relazione al precedente e conservando solo il minimo tra i due, e non calcolando tutti i valori e poi cercando il minimo nell'intera lista.
Un potenziale errore che potrebbe compromettere il funzionamento dell'algoritmo può essere facilmente commesso nel valutare $W(\neg c_1)$. La condizione $\neg c_1$ infatti è una congiunzione di più condizioni, per cui ricordando le leggi de Morgan, riportate di seguito nel caso proposizionale per semplicità:

$$\neg(a \wedge b) \equiv \neg a \vee \neg b$$
$$\neg(a \vee b) \equiv \neg a \wedge \neg b$$

è necessario valutare $\neg c_1$ come una disgiunzione di singole negazioni, con un predicato specifico, ed evitare di cadere nella trappola di valutarla negando le singole condizioni e di considerarle ugualmente come se fossero in congiunzione. Questo potenziale errore emerge solo nelle successive iterazioni dell'algoritmo, in quanto alla prima iterazione, essendo c_1 una condizione sempre vera, $\neg c_1$ sarà una condizione sempre falsa, che messa in *and* con altre condizioni genererà comunque condizioni sempre false, dando a $W(\neg c_1)$ un risultato falsato.

L'interprete con cui è stato testato è l'ultima versione di SWI-Prolog.

3.5 Esempi di output

Si riportano di seguito tre esempi di output per via grafica. Il primo esempio è il risultato del "tennis dataset".

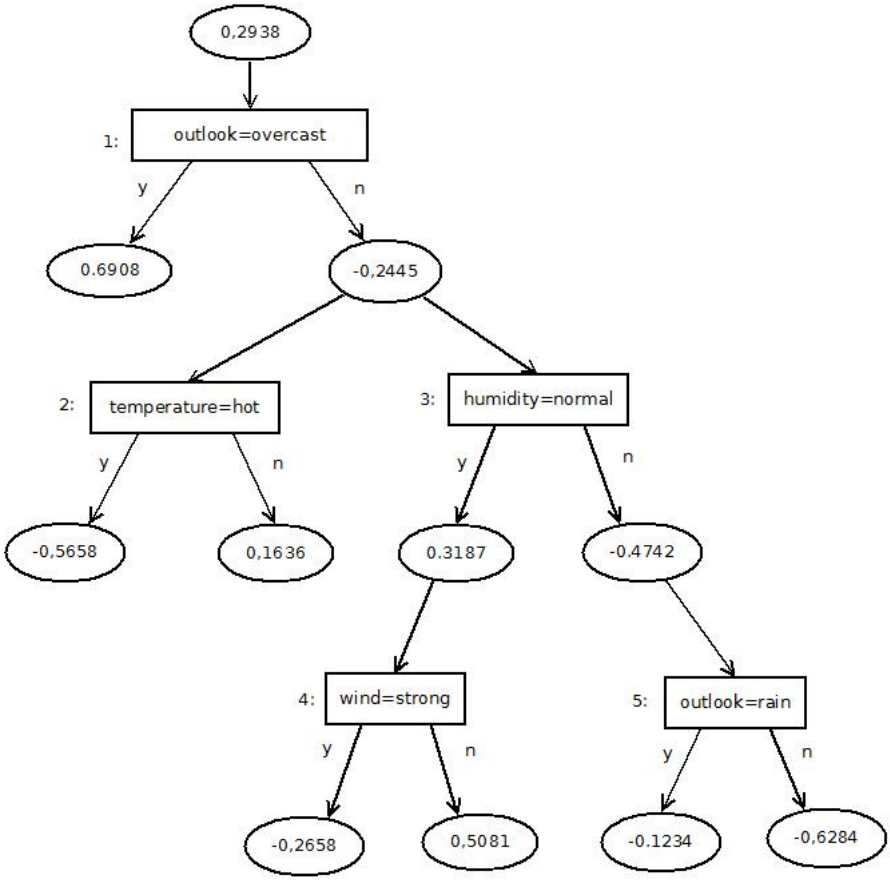

Figura 3.3: ADTree tennis

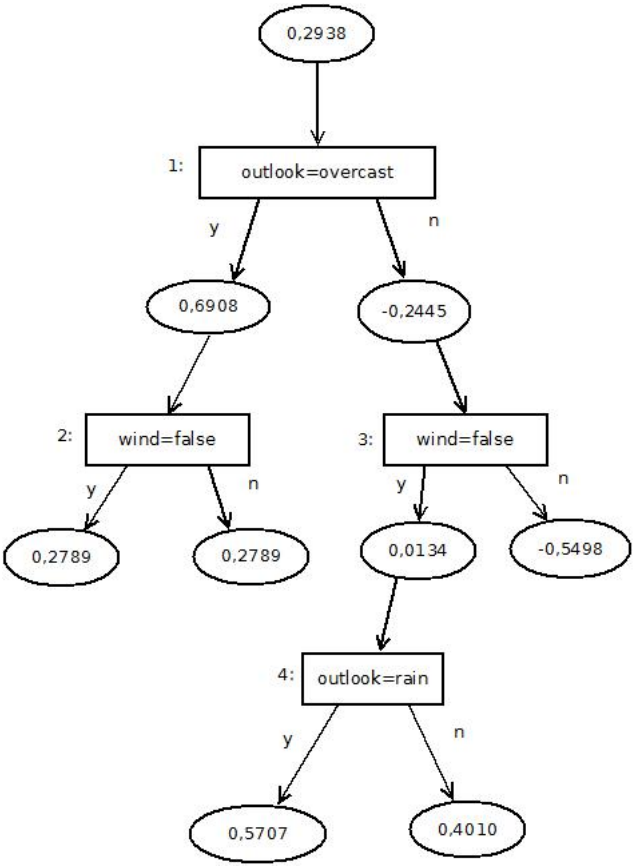

Figura 3.4: ADTree golf

Come si nota, secondo questi dati, un valore per l'attributo *outlook* uguale a *overcast* termina la classificazione, altrimenti viene preso un altro path. Si noti come temperatura e umidità vengano analizzate separatamente, come accade anche per vento e outlook per l'ultimo livello.

Per il dataset golf invece la situazione appare leggermente differente, in quanto per il modello sembrano contare solo due degli attributi per la classificazione, tanto da essere gli unici considerati, anche più volte.

Infine la terza immagine è relativa allo XOR. Per quanto il modello funzioni e tutte le combinazioni vengano classificate correttamente, l'output è un albero di difficile lettura e di scarsa comprensibilità, confermando che per tale funzione è più appropriato un altro tipo di algoritmo di apprendimento.

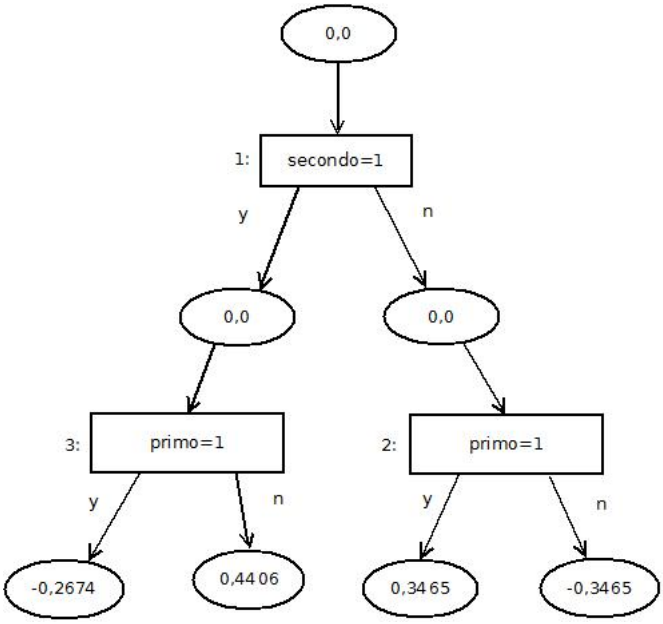

Figura 3.5: ADTree XOR

3.6 Guida all'utilizzo

Il sistema si apre facendo doppio click sul file "adtree.pl", o alternativamente aprendo l'interprete Prolog, spostandosi nella cartella del file e usando il comando *consult*. Il sistema porta con sé tre dataset, uno con il "tennis dataset", uno contenente il "golf dataset", entrambi facilmente reperibili sulla rete, e l'ultimo contenente un dataset per lo XOR.
L'avvio del sistema viene dato con il semplice comando:
?-avvio.
dato all'interprete. L'interazione è testuale, e viene portata avanti rispondendo alle domande sulla scelta del dataset, il numero di iterazioni, la visione dei nodi dell'albero, l'interrogazione del modello ottenuto dall'albero. Il formato con cui interrogare il modello deve essere simile al seguente esempio:
$[overcast, mild, high, strong]$
ed eventualmente riportando con "[ ]" un valore mancante. Il sistema risponderà con la classificazione dell'esempio.

Conclusioni

In conclusione, il linguaggio Prolog si è rivelato particolarmente adatto all'implementazione dell'algoritmo di apprendimento dell'Alternating Decision Tree, mostrando dei buoni risultati anche in termini di prestazioni. I possibili sviluppi futuri possono riguardare sia l'aspetto pratico ed implementativo dell'algoritmo, incorporando ottimizzazioni di vario genere, sia nella fase di learning che nella fase di visita dell'albero, sia gli aspetti più teorici dell'algoritmo.
AdaBoost possiede anche una versione per la classificazione multiclasse, e quindi la possibilità di gestire più classi potrebbe essere un'ottima miglioria per l'algoritmo. Inoltre potrebbe essere studiato un metodo per dare in modo automatico il giusto stop all'apprendimento o il test di tecniche di potatura che rimuovano nodi non necessari dall'albero, in modo da poter gestire il processo di boosting nel modo opportuno.

Bibliografia

- Tom Mitchell, "Machine Learning", 1th ed., McGraw-Hill, 1997
- Console L., Lamma E., Mello P., Milano M., "Programmazione logica e Prolog", 1th ed., Libreria UTET, 1997
- Freund Y., Mason L., "The alternating decision tree learning algorithm", Proceedings of the Sixteenth International Conference on Machine Learning, Morgan Kaufmann
- Freund Y., Shapire R., "A short introduction to Boosting", Journal of Japanese Society for Artificial Intelligence,September, 1999
- Pfhringer B., Holmes G., Kirby R., "Optimizing the induction of alternating decision trees", Proceedings of the Fifth Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining, 2001
- Iba W., Langley P., "Induction of One-Level Decision Trees", Proceedings of the Ninth International Conference on Machine Learning, Aberdeen, Scotland, 1–3 July 1992, San Francisco, Morgan Kaufmann
- Viola P., Michael J., "Robust Real-Time Face Detection", International Journal of Computer Vision, 2004