Appunti di Ingegneria della Conoscenza

Corso di Laurea Triennale in Informatica

Nicolas Pinto

Anno Accademico: 2022-2023

Indice

Capitolo 1

Sistemi basati su conoscenza

1.1 Introduzione

"Typically information is defined in terms of data, knowledge in terms of information, and wisdom in terms of knowledge"

1.1.1 Sistemi Intelligenti

La conoscenza può essere rappresentata come una piramide in cui partendo dalla base si ha: dati, informazioni, conoscenza e sapere. Ci sono rappresentazioni alternative quali, concatenazioni o grafi.

I dati sono simboli che rappresentano stimoli o segnali, l'informazione è costituita da dati dotati di significato e la conoscenza rappresenta un'informazione elaborata, organizzata e applicata.

L'intelligenza Artificiale (AI) mira a studiare e comprendere i principi che rendono possibile un comportamento intelligente in sistemi artificiali. Chiaramente questo può avvenire solo se si assume che la computazione possa essere una forma di ragionamento, quindi di intelligenza.

1.1.2 Progettazione di sistemi basati su conoscenza

Durante la progettazione di un sistema basato su conoscenza si distinguono tre diversi momenti:

1. Elaborazione in fase di progetto, a cura del progettista.
2. Computazione *offline*, ovvero una fase di apprendimento di una conoscenza di fondo data o definita in precedenza dalla quale ricavare una *base di conoscenza* (KB) utilizzabile nel seguito.
3. Computazione *online*, ovvero una fase in cui si ricavano informazioni dalle *osservazioni* e vengono effettuate delle decisioni sulla base della KB.

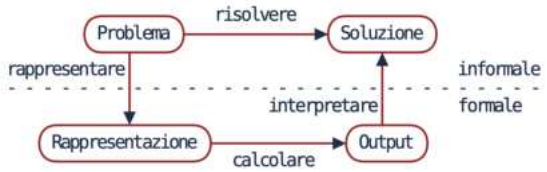


Figura 1.1

Nella AI è importante rappresentare *cosa* vada calcolato. Il *come* consiste in una ricerca di soluzioni possibili. Per questo motivo gioca un ruolo fondamentale *rappresentare* formalmente un problema a partire da una sua descrizione informale, mediante l'utilizzo di opportuni linguaggi di rappresentazione. Quindi al livello formale si otterrà un *output* come prodotto di una computazione, il quale può essere interpretato come *soluzione* a livello informale.

Come nell'ingegneria del software, il progettista deve definire cosa costituisca una soluzione. Le soluzioni possono essere classificate come segue:

- Soluzione ottimale: migliore secondo una misura di qualità
- Soluzione soddisfacente: buona secondo una specifica delle risposte adeguate, quando ottenere la soluzione migliore è troppo costoso.
- Soluzione approssimata: soluzione di qualità prossima a quella ottimale
- Soluzione probabile: soluzione di cui si è sicuri con una con un certo grado di certezza. Indica una forma approssimata di una soluzione precisa. Si possono distinguere i tassi di errore per falsi positivi e falsi negativi.

Come accennato in precedenza, nella progettazione di un sistema intelligente vi sono problematiche quali:

- come acquisire e rappresentare la conoscenza di un dominio
- come usarla per rispondere a domande o risolvere problemi

Secondo l'ipotesi di Newell e Simon, un sistema simbolico fisico è necessario e sufficiente alla generazione di azioni intelligenti.

La rappresentazione del mondo però, per risultare utile deve avere un certo livello di astrazione, evitando l'eccesso di dettagli. Occorre dunque trovare un giusto livello di astrazione, considerando che un alto livello di astrazione è più comprensibile dall'uomo ma distante dalla comprensione della macchina e viceversa per un basso livello di astrazione.

Ad esempio, per un robot, è sufficiente la planimetria di un edifizio come rappresentazione del mondo reale, ignorando distanze, grandezze, angoli di sterzata, ecc.

1.1.3 Complessità dei Sistemi

Le dimensioni della complessità nella progettazione definiscono uno spazio di progettazione e forniscono una decomposizione sommaria dello spazio.

Dimensione	Valori
Modularità	piatta, modulare, gerarchica
Schema di rappresentazione	stati / feature / relazioni e individui stati: n_s / 2^{n_f} / $2^{n_s^2} \cdot n_r$ (rel. binarie)
Incertezza sull'osservazione	totalmente / parzialmente osservabile
Incertezza sull'effetto	deterministico, stocastico
Preferenze	finalità, preferenze complesse
Apprendimento	conoscenza data, conoscenza appresa
Limiti alle risorse computazionali	razionalità perfetta, razionalità limitata
Orizzonte	senza pianificazione, a stage finite, indefinite, infinite
Numero di attori	singolo sistema/agente, sistemi/agenti multipli (distribuiti)

Figura 1.2

Capitolo 2 =DFS-DBSUND

Spazi di stati e ricerca di soluzioni

2.1 Risoluzione di problemi mediante ricerca

La *ricerca* è un caso semplice di *soluzione di un problema* caratterizzato, in assenza di incertezza, da un modello del mondo fatto di stati (rappresentazione piatta) in cui si deve raggiungere uno stato obiettivo a partire dallo stato corrente. Il goal consiste quindi nel trovare la sequenza di passi da effettuare per raggiungere lo stato finale.

L'astrazione del problema si riduce alla ricerca di un percorso in un grafo orientato da un nodo di partenza ad un nodo finale.

L'idea alla base consiste nel far costruire al sistema una serie di soluzioni parziali (percorsi). Per ognuna di esse si verifica che sia un percorso verso uno dei goal, in caso negativo, si costruiscono nuove soluzioni parziali estendendo quelle correnti.

2.1.1 Formalizzare un problema di ricerca

Un problema di ricerca si formalizza mediante:

- Uno *spazio di stati*, ovvero un insieme di stati in cui vi è il sottoinsieme di stati di partenza ed il sottoinsieme di stati obiettivo. Questi ultimi sono specificabili anche attraverso una funzione booleana, `goal(s)`, che risulta verificata se e solo se `s` è uno stato obiettivo.
- *insieme di azioni* per ogni stato
- *funzione d'azione*: (stato, azione) → nuovo stato
- *criterio* di qualità per soluzioni accettabili. La soluzione ottimale massimizza tale criterio

2.1.2 Ricerca su grafo

Per risolvere un problema di ricerca è necessario definire lo spazio di ricerca e applicarvi un algoritmo di ricerca. Molti compiti sono riconducibili al problema di trovare percorsi tra nodi di partenza e nodi obiettivo all'interno di un grafo (eventualmente orientato).

Nota: la complessità di un grafo è indicata da un fattore di ramificazione dei suoi nodi che può essere uscente se indica il numero di archi uscenti dal nodo, o entrante se indica il numero di archi entranti nel nodo.

Un algoritmo di ricerca generico, dato un grafo, esplora incrementalmente percorsi dai nodi di partenza verso nodi obiettivo memorizzando di volta in volta la *frontiera* dei percorsi già esplorati. Inizialmente la frontiera è composta dai soli nodi di partenza, successivamente vi è un'espansione dei percorsi verso i nodi non ancora esplorati, fino ad incontrare i nodi goal:

- Si seleziona un percorso, rimuovendolo dalla frontiera
- Si estende il percorso con ogni arco uscente dall'ultimo nodo
- Si aggiungono alla frontiera i percorsi ottenuti

2.1.3 Strategie di ricerca non informate

Il problema determina il grafo e l'obiettivo. La strategia di ricerca specifica il percorso da selezionare dalla frontiera e nello specifico, le strategie non informate non prendono in considerazione la posizione dell'obiettivo. Se gli archi del grafo hanno costo unitario, possono applicarsi algoritmi di ricerca in ampiezza, in profondità e iterative deepening. Se, invece, al grafo è associata una funzione di costo, allora si applica una procedura di ricerca a costo minimo.

Ricerca in ampiezza (BFS)

La ricerca in ampiezza (breadth-first- search) è un metodo di ricerca non informato e non euristico in cui la frontiera, cioè l'insieme dei percorsi già esplorati, viene implementata con una coda FIFO. Il suo obiettivo è quello di espandere il raggio d'azione al fine di esaminare tutti i nodi del grafo sistematicamente, fino a trovare il nodo cercato. In altre parole, se il nodo cercato non viene trovato, la ricerca procede in maniera esaustiva su tutti i nodi del grafo.

Il procedimento da seguire per metterlo in pratica è sintetizzato come segue:

1. Mettere in coda il nodo o i nodi radice.
2. Togliere dalla coda il primo nodo (nella prima iterazione il nodo sorgente) ed esaminarlo.
    - Se l'elemento cercato è trovato in questo nodo viene restituito il risultato e la ricerca si interrompe.
    - Se l'elemento cercato non era in questo nodo mettere in coda tutti i successori non ancora visitati del nodo in analisi.

3. Se la coda è vuota, ogni nodo nel grafo è stato visitato e l'elemento non è stato trovato perché non presente e quindi la ricerca si interrompe.

4. Se la coda non è vuota, ripetere il passo 2.

Complessità Sia b il fattore di ramificazione, se il primo percorso della frontiera ha n archi, ci sono almeno b^{n-1} elementi nella frontiera. Quindi complessità in spazio e tempo esponenziali al numero degli archi del percorso di soluzione di lunghezza minima.

Utilità Il BFS ci garantisce, quando esiste, il ritrovamento della soluzione (con il minor numero di archi) ed è utile quando:

- non si hanno problemi di spazio
- si cerca una soluzione con numero di archi minimale
- lo spazio presenta percorsi di lunghezza infinita, il BFS infatti esplora interamente lo spazio degli stati.

Il BFS non è utile invece quando

- tutte le soluzioni sono associate a percorsi lunghi
- è disponibile l'euristica
- il grafo viene generato dinamicamente, in quanto comporterebbe un'enorme crescita della complessità spaziale.

Ricerca in profondità (DFS)

Nella ricerca in profondità la frontiera viene gestita con una pila LIFO in cui gli elementi sono aggiunti uno alla volta ed il nodo da espandere è l'ultimo inserito.

Con uno stack la ricerca procede in profondità completando un singolo percorso prima di provare percorsi alternativi. Questo avviene applicando il *backtracking*, ovvero si seleziona una prima alternativa per ogni nodo, tornando indietro alla successiva solo dopo aver tentato tutti i completamenti.

Complessità Se b è il fattore di ramificazione e k la lunghezza del primo percorso della lista, ci sono al più altri $k(b-1)$ percorsi.

Nel caso ottimo la soluzione è già sul primo ramo, quindi la complessità è lineare nella lunghezza del percorso.

Nel caso pessimo l'algoritmo diverge. Questo avviene in presenza di grafi infiniti o che presentano cicli. Se il grafo è un albero finito, con fattore di ramificazione limitato da b e profondità k, il caso pessimo ha complessità esponenziale $O(b^k)$

Utilità L'algoritmo di ricerca DFS è utile in caso di spazio limitato, in presenza di molteplici soluzioni, anche se costituite da percorsi lunghi.

È inefficiente quando sono possibili percorsi infiniti oppure quando, pur esistendo soluzioni alternative poco profonde, esse sono localizzate sulla destra del grafo e quindi la ricerca si attarda su percorsi più lunghi alla sinistra del grafo.

Iterative Deepening =DFS + BOUND

L'obiettivo dell'algoritmo di ricerca Iterative Deepening è combinare l'efficienza in spazio del DFS con l'ottimalità del BFS. L'idea alla base consiste nel non memorizzare ma ricalcolare gli elementi della frontiera del BFS.

Quindi il DFS all'interno delle iterazioni del BFS ricerca fino a una profondità limitata, eliminando precedenti computazioni e ripartendo, se necessario. Si parte da una lunghezza unitaria come lunghezza massima di percorso, poi si incrementa di una unità ad ogni iterazione. Se esiste, una soluzione verrà trovata ed esplorando percorsi in ordine di lunghezza quello con meno archi sarà individuato per primo.

Per questo algoritmo si possono definire due momenti di *fallimento* della ricerca, il fallimento *innaturale*, in cui viene raggiunto il limite di profondità incrementando il quale, la ricerca riparte. Il fallimento *naturale*, in cui si esaurisce lo spazio di ricerca, dunque non esiste alcuna soluzione a nessun livello di profondità.

Ricerca a costo minimo (LCFS)

Per diversi domini, gli archi hanno costi non unitari. In questi casi agli archi del grafo sono associati dei costi (o pesi) e l'obiettivo consiste nel trovare il percorso con il minor costo totale. Questa rappresenta una differenza sostanziale con i tre algoritmi precedenti, i quali minimizzano il numero degli archi ma non necessariamente il costo totale. In effetti potrebbe esistere una soluzione alternativa con percorso più breve, ma di costo maggiore.

Il più semplice algoritmo che garantisce di trovare il percorso a costo minimo è il lowest-cost-first search, il cui funzionamento è simile al breadth-first search, ma invece di espandere un percorso con il minor numero di archi, esso seleziona un percorso con il costo inferiore. Questo è implementato utilizzando per la frontiera una coda con priorità ordinata sulla base del costo.

Complessità Dato un fattore di ramificazione finito e se i costi sono limitati inferiormente da una costante positiva, se esiste, viene garantita la soluzione ottimale. Infatti il primo percorso trovato termina in uno nodo obiettivo e procedendo in ordine di costo, se ne esiste uno migliore allora viene selezionato.

2.2 Strategie di ricerca informate

2.2.1 Ricerca Euristica

Contrariamente a quello algoritmico, il procedimento euristico è basato sull'intuizione e sullo stato temporaneo delle cose con l'obiettivo di generare una nuova conoscenza. In informatica l'euristica è una regola pratica dettata dall'esperienza passata. Per risolvere un problema mediante un algoritmo, infatti, si utilizzano conoscenze specifiche che vanno al di là della definizione del problema stesso. Quelli euristici sono procedimenti logici dominati dall'incertezza e quindi legati al probabile e al possibile. L'algoritmo è tipicamente sequenziale (step by step), l'euristica, invece, è fondamentalmente reticolare. Gli algoritmi euristici, sono algoritmi che non garantiscono di ottenere la soluzione ottima, ma in generale sono in grado di fornire una "buona" soluzione ammissibile per il problema.

L'algoritmo di *ricerca euristica* prende in considerazione informazioni sull'obiettivo nella selezione dei nodi attraverso una funzione euristica h la quale ad ogni nodo n associa un numero reale non negativo che rappresenta la stima del costo minimale d'un percorso da n fino a un nodo goal.

Si dice che h è *ammissibile* se *sottostima* il costo, ovvero se $h(n)$ è minore o uguale rispetto al costo minimale effettivo del percorso. Solitamente questa richiesta è soddisfatta se si considera come stima la soluzione di un problema simile ma più semplice, utilizzando i costi relativi come euristica per risolvere il problema originale.

Nell'esempio del robot visto a lezione, una euristica ammissibile per la ricerca del percorso dalla posizione o103 alla posizione r123 è rappresentata dalla distanza euclidea, quindi una sottostima, tra le varie posizioni.

2.2.2 DFS Euristica

È chiaro come si possa estendere la stessa funzione euristica h, dai singoli nodi, al caso dei percorsi (non vuoti) semplicemente applicando h all'ultimo nodo del percorso stesso:

$$h(< n_0, \ldots, n_k >) = h(n_k)$$

Tale h estesa può essere applicata nell'algoritmo di DFS Euristica ove viene utilizzata per ordinare i vicini aggiunti alla pila (o frontiera) in modo che il nodo con la migliore euristica vada in cima alla pila. Si tratta di una scelta *locale* in quanto vengono esplorati i percorsi che estendono quello selezionato prima di tentarne altri e presenta gli stessi problemi della DFS.

2.2.3 Greedy Best First Search (GBFS)

Al contrario della DFS Euristica, in questo algoritmo la scelta effettuata sulla base della euristica è *globale* in quanto il percorso con il minimo valore $h(n)$ viene selezionato dell'intera frontiera.

Generalmente funziona bene, ma a volte potrebbe intraprendere percorsi apparentemente promettenti per h, ma con costi che possono aumentare rapidamente.

Utilità Nel problema della ricerca ad esempio, l'euristica semplice non è sufficiente ad evitare loop dovuti alla presenza di cicli nel grafo. Si consideri il grafo in Figura 2.1

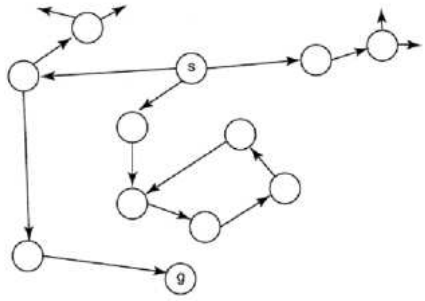


Figura 2.1

Supponendo di voler raggiungere il nodo g partendo dal nodo s percorrendo il percorso minimo, di considerare la lunghezza di ogni arco come il relativo costo e di avere come funzione euristica h la distanza Euclidea da g:

- La DFS Euristica seleziona il nodo sotto s divergendo
- GBFS, analogamente, dato che tutti i nodi sotto s sembrano buoni, itererà su di essi, non provando mai strade alternative.

2.2.4 Ricerca A*

A* è un algoritmo di ricerca su grafi che individua un percorso da un dato nodo iniziale verso un dato nodo goal.

Nella selezione del percorso da espandere si considerano sia il costo per percorso parziale sia una stima euristica. Quindi per ogni percorso $p =< s, \ldots, n >$ della frontiera, A* usa una stima del costo di un percorso completo (fino a un goal g) che lo estenda:

$$f(p) = cost(p) + h(p)$$

ove:

- $cost(p)$ è il costo di p partendo da un nodo iniziale s
- $h(p)$ è la stima del costo del cammino successivo da n a g

L'algoritmo A* è anche un esempio di ricerca best-first, ovvero, ad ogni passo, tra tutti i nodi possibili da espandere l'algoritmo sceglie il nodo con la funzione di valutazione (h) più bassa (da cui best-first).

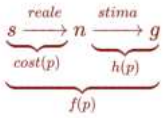


Figura 2.2

Implementazione di A* L'implementazione di A* segue lo schema di algoritmo di ricerca generico implementando una frontiera memorizzata in una coda con priorità che viene ordinata da $f(p)$. Nello specifico, il percorso parziale nella frontiera che dà $f(p)$ minimo viene posizionato in cima. Inoltre se $h(n)$ è ammissibile, allora $f(p)$ non sovrastima il costo di un percorso completo che includa p.

A* può seguire molti percorsi, per cui potrebbe esserci bisogno di qualche backtrack occasionale, perché magari raggiungo un nodo dal quale non posso procedere, ma intuitivamente questa è una strategia che ha buone chance di trovare l'obiettivo velocemente. Inoltre, può essere provato che questa strategia troverà in ogni caso la strada migliore possibile, cioè la soluzione ottimale, come farebbe la ricerca breadth-first.

Ammissibilità degli algoritmi

Un'euristica si definisce *ammissibile* se è una funzione euristica che non sovrastima mai il costo effettivamente necessario per raggiungere l'obiettivo. Intuitivamente, una funzione ammissibile è "ottimistica", in quanto sottostima sempre il costo effettivo. Da questo ne deriva che un algoritmo di ricerca è ammissibile quando, se esistono soluzioi, ne troverà sempre una ottimale (di costo minimo) anche se lo spazio degli stati è infinito.

A* è un algoritmo ammissibile se il fattore di ramificazione è finito, i costi degli archi sono maggiori di un certo $\epsilon > 0$ e h è un'euristica ammissibile, cioè, non sovrastima il costo minimale per percorsi da n a un nodo goal.

Questo algoritmo garantisce che la prima soluzione sarà ottimale anche in presenza di cicli, ma non assicura che ciascun nodo intermedio selezionato dalla frontiera sia un cammino ottimo.

Importanza dell'euristica in A* A* è un algoritmo che fa apprezzare l'efficienza di una sottostima abbastanza vicina alla realtà. Infatti possiamo osservare che: dato c il costo del percorso di costo minimo, se h è ammissibile, A* espande ogni percorso dal nodo di partenza nell'insieme

$$\{p|cost(p) + h(p) < c\}$$

più alcuni percorsi nell'insieme

$$\{p|cost(p) + h(p) = c\}$$

Quindi per migliorare l'efficienza di A* andrebbe scelta una h che riduca la cardinalità del primo insieme.

2.2.5 IDA*

Iterative deepening A* (noto anche con l'acronimo IDA*) è in grado di trovare il cammino minimo fra un nodo indicato come iniziale e ciascun membro di un insieme di "nodi obiettivo" in un grafo pesato.

L'algoritmo è una variante dell'iterative deepening depth-first search usata per migliorare le prestazioni di A*.

Esattamente come nell'iterative deepening DFS, l'algoritmo viene ripetuto più volte con una soglia sempre più grande, finché non verrà raggiunta una soluzione. Tuttavia, in questo caso la soglia non dipenderà più dalla profondità rispetto al nodo radice, ma dal valore assunto dalla funzione di valutazione. Come in A* ed altri algoritmi euristici, viene usata come funzione di valutazione $f(n) = cost(n) + h(n)$ dove *cost* è il costo accumulato per raggiungere il nodo n, mentre h è una stima euristica del costo necessario per arrivare alla soluzione partendo da n.

L'algoritmo inizia impostando la soglia (threshold) $T \leftarrow f(s)$ dove s è il nodo iniziale e mette i figli di s in una coda (detta *fringe*). Successivamente, espande i nodi nel fringe per i quali $f(n) = T$ finché non raggiunge una soluzione. Se la ricerca fallisce in modo *innaturale*, cioè nel fringe non è presente nessun n tale che $f(n) = T$ allora l'algoritmo imposta $T \leftarrow \min_{n \in fringe}(cost(n) + h(n))$, ovvero aggiorna la soglia in base al minimo valore assunto dalla funzione di valutazione fra tutti i nodi del fringe.

Il vantaggio principale di IDA* è l'uso lineare della memoria, al contrario di A* che ha bisogno, nel caso peggiore, di uno spazio esponenziale. D'altro canto, questo algoritmo usa fin troppa poca memoria, che potrebbe essere invece sfruttata per migliorare le prestazioni in termini di tempo.

2.2.6 Potatura

Potatura dei cicli

Abbiamo visto come A* non si faccia ingannare dalla presenza di cicli all'interno di un grafo. È possibile però adottare strategie di *pruning* (o potatura) che permettano di riconoscere cicli o percorsi multipli sullo stesso nodo ed evitarli.

Per grafi finiti una strategia consiste nel *non* prendere in considerazione i vicini di un nodo che sono già contenuti nel percorso. Quindi viene effettuato un controllo (preventivo o successivo) che, dato un nodo da aggiungere al percorso, ne testa la presenza all'interno della sequenza di nodi che costituiscono il percorso stesso e lo aggiunge solo nel caso in cui non è presente.

La complessità della potatura dipende dal metodo di ricerca usato. Essa sarà costante per metodi con un solo percorso di lavoro, memorizzato come insieme. Si utilizza in tal caso una funzione hash o si associa ad ogni nodo un bit che è acceso quando viene aggiunto a un percorso e spento in caso di backtracking, sarà quindi sufficiente non espandere i nodi con il bit acceso. Nel caso di metodi che lavorano su più percorsi la complessità sarà lineare in quanto, per evitare i cicli, si evita di aggiungere al percorso parziale un nodo già presente.

Multiple-Path Pruning (MPP)

Spesso, i cicli non sono l'unico problema, infatti può accadere che più di un percorso porti allo stesso nodo. Una strategia risolutiva potrebbe consistere nel potare dalla frontiera qualsiasi altro percorso porti a un nodo per il quale ne esiste già un altro.

Il Multiple-Path Pruning (o MPP) è implementato gestendo una lista di nodi finali di percorsi già espansi, detta closed list o explored set. Questa lista è inizialmente vuota, durante la computazione, selezionando un percorso $< n_0, \ldots, n_k >$ se n_k è già nella lista esso può essere scartato, altrimenti si aggiunge n_k alla lista e si prosegue.

Questo approccio non garantisce necessariamente che il percorso di costo minimo non venga scartato. Ma talvolta necessitiamo di tale garanzia, per cui si può applicare una alternativa più sofisticata:

- Assicurare che il primo percorso trovato per un dato nodo sia ottimale, andando a scartare i successivi
- se nella frontiera vi è il percorso $p =< s, \ldots, n, \ldots, m >$ e viene trovato un percorso $p' =< s, \ldots, n >$ meno costoso della parte fino a n in p, si può o eliminare p o sostituire in p tale parte con p'.

Nell'algoritmo LCFS (o ricerca a costo minimo), il primo percorso verso un dato nodo è quello di costo minimo, quindi potare percorsi successivi assicura comunque di poter trovare una soluzione ottimale.

A* non garantisce che quando si selezione un percorso verso un nodo per la prima volta, questo sia quello di costo minimo. Si noti che il teorema dell'ammissibilità lo garantisce per ogni percorso che porta ad un nodo goal, ma non per percorsi che portano ad un qualsiasi nodo.

Comunque questa caratteristica può essere ottenuta se alla funzione euristica si garantisce, oltre alla ammissibilità, anche la *consistenza*.

Consistenza e monotonicità Una euristica consistente è una funzione $h(n)$ non negativa sul nodo n che soddisfa il vincolo

$$h(n) \leq cost(n, n') + h(n')$$

per ogni coppia di nodi n e n' dove $cost(n, n')$ è il costo del percorso a costo minimo da n ad n'. Si noti che se $h(g) = 0$ per ogni nodo goal g, allora una euristica consistente non sovrastimerà mai il costo dei percorsi da un nodo verso un obiettivo.

La consistenza può essere garantita se la funzione euristica soddisfa la *restrizione di monotonicità*:

$$h(n) \leq cost(n, n') + h(n')$$

per ogni arco $< n, n' >$.

In questo modo è più facile controllare la restrizione di monotonicità dal momento che dipende dagli archi piuttosto che da ogni coppia di nodi.

MPP vs CP

Il MPP è una tecnica più generale della potatura dei cicli (essendo i cicli una sottoclasse di multi-path) ed è preferibile con metodi in ampiezza. La potatura dei cicli (CP) è preferibile con metodi in profondità.

Complessità MPP può essere eseguita:

- in tempo costante su grafi espliciti, con un bit per nodo per il quale sia stato trovato un percorso, o con funzione hash
- in tempo logaritmico se il grafo viene generato dinamicamente, salvando la lista dei nodi espansi.

Strategia	Selezione da Frontiera	Soluzione Garantita	Complessità in Spazio
DEPTH-FIRST	ultimo nodo aggiunto	No	lineare
BREADTH-FIRST	primo nodo aggiunto	con meno archi	esponenziale
ITERATIVE DEEPENING	–	con meno archi	lineare
GREEDY BEST-FIRST	$h(p)$ minimale	No	esponenziale
LOWEST-COST-FIRST	$cost(p)$ minimale	costo minimo	esponenziale
A*	$cost(p) + h(p)$ minimale	costo minimo	esponenziale
IDA*	–	costo minimo	lineare

Figura 2.3: Tabella di sintesi delle strategie di ricerca

Completezza Un algoritmo di ricerca si definisce *completo* se garantisce la soluzione quando esiste. Le strategie che trovano percorsi con un numero di archi o costo minimo sono complete. Nel caso pessimo si ha tempo esponenziale nel numero di archi dei percorsi esplorati.

2.3 Strategie di ricerca più sofisticate

Un certo numero di perfezionamenti possono essere fatti per le strategie precedenti. In primo luogo, presentiamo la ricerca *branch-and-bound depth-first*, che è garantita per trovare una soluzione ottimale, e può sfruttare una funzione euristica, come la ricerca A*, ma con i vantaggi spaziali della ricerca depth-first. Presentiamo anche *metodi di riduzione* dei problemi per suddividere un problema di ricerca in una serie di problemi di ricerca più piccoli, ognuno dei quali

può essere molto più facile da risolvere. Infine, mostriamo come la *programmazione dinamica* può essere utilizzata per trovare percorsi da qualsiasi luogo a un obiettivo e per costruire funzioni euristiche.

2.3.1 Branch and Bound

L'algoritmo di ricerca Branch and Bound combina l'efficienza in spazio delle strategie in profondità con l'informazione euristica per la ricerca del percorso ottimale. È particolarmente efficace quando esistono molteplici cammini verso un nodo goal. Come nella ricerca A*, la funzione euristica $h(n)$ è non negativa e inferiore o uguale al costo minimo del percorso da n a nodo goal, in altri termini la si assume ammissibile.

L'idea alla base della ricerca branch-and-bound consiste nel memorizzare il percorso di costo minimo trovato verso un nodo goal e settare il relativo costo come *bound*. Se la ricerca trova un percorso p tale che $cost(p) + h(p) \geq bound$, allora p può essere eliminato (potato). Se l'algoritmo trova un percorso completo migliore del percorso già memorizzato allora lo rimpiazza aggiornando anche il *bound*. Quindi la ricerca procede per trovare una soluzione migliore. Per avviare l'algoritmo il bound viene settato ad un valore molto alto. Questo algoritmo, quindi, genera una sequenza di soluzioni via via migliori fino a quella ottimale.

2.3.2 Direzione della ricerca

La dimensione dello spazio di ricerca di un generico algoritmo di ricerca dipende dalla lunghezza del percorso e dal fattore di ramificazione. Tutto ciò che può essere fatto per ridurre queste caratteristiche può potenzialmente dare grandi risparmi.

Se sono rispettate le seguenti condizioni:

- il numero di nodi goal è finito
- per ogni nodo n possono essere generati i vicini $\{n' :< n, n' >\in A\}$ nel *grafo inverso*

l'algoritmo di ricerca dei grafici può iniziare con il nodo iniziale e cercare in avanti un nodo obiettivo, oppure iniziare con un nodo obiettivo e cercare all'indietro il nodo iniziale.

In molte applicazioni l'insieme dei nodi obiettivo o il grafico inverso non possono essere facilmente generati rendendo infattibile la ricerca all'indietro. È anche vero però che a volte lo scopo della ricerca è solo quello di trovare un nodo obiettivo e non il percorso per raggiungerlo.

Nella *ricerca in avanti* si procede da un nodo di partenza e si raggiungono i nodi goal, nella *ricerca all'indietro* si parte da un nodo goal ed usando il grafo inverso si termina in un nodo di partenza.

Ricerca bidirezionale

L'idea della *ricerca bidirezionale* è quella di cercare in avanti dall'inizio e indietro dall'obiettivo contemporaneamente. Quando le due frontiere di ricerca si intersecano, l'algoritmo deve costruire un unico percorso che si estende dal nodo di partenza attraverso l'intersezione di frontiera a un nodo obiettivo. Un nuovo problema si pone nel corso di una ricerca bidirezionale, vale a dire garantire che le due frontiere di ricerca si incontrino effettivamente. Per esempio, per una ricerca in profondità in entrambe le direzioni è improbabile che l'algoritmo funzioni a meno che uno sia estremamente fortunato in quanto le frontiere di ricerca hanno dimensioni decisamente ridotte e possono facilmente "sorpassarsi" senza incontrarsi. Al contrario, un ricerca in ampiezza garantisce l'incontro delle due frontiere. Una combinazione di ricerca in profondità in una direzione e ricerca in ampiezza nell'altra garantirebbe l'intersezione tra le frontiere ma la scelta di quale applicare ed in quale direzione potrebbe essere difficile.

Ricerca basata su isole

Uno dei modi in cui la ricerca può essere resa più efficiente è quello di identificare un numero limitato di luoghi in cui la ricerca in avanti e la ricerca all'indietro potrebbero incontrarsi. Ad esempio, nella ricerca di un percorso da due camere su piani diversi, può essere opportuno vincolare la ricerca di andare prima all'ascensore su un livello, andare al livello appropriato e poi andare dall'ascensore alla stanza obiettivo. Intuitivamente, queste posizioni designate sono isole nel grafico di ricerca, che sono vincolate ad essere su un percorso di soluzione da un nodo iniziale a un nodo obiettivo.

Quando vengono specificate le isole, un agente può scomporre il problema della ricerca in diversi problemi di ricerca ottenendo tre problemi più semplici da risolvere. Avere problemi più piccoli aiuta a ridurre l'esplosione combinatoria di grandi ricerche ed è un esempio di come la conoscenza supplementare su un problema è usata per migliorare l'efficienza di ricerca.

Per trovare un cammino tra s e g usando le isole si deve:

- Identificare un insieme di isole $i_0 \ldots, i_k$
- trovare un cammino da s a i_0, da i_{j-1} a i_j e da i_k a g.

Ognuno di questi problemi di ricerca corrisponde ad un sotto problema più semplice del problema generale e, quindi, più semplice da risolvere.

Ricerca in una gerarchia di astrazioni

La nozione di isole può essere utilizzata per definire strategie di problem solving che funzionano a più livelli di astrazione. L'idea di cercare in una gerarchia di astrazioni comporta in primo luogo astrarre il problema, tralasciando quanti più dettagli possibili. Una soluzione al problema astratto può essere vista come una soluzione parziale al problema originale, infatti ci si aspetta che un'adeguata

astrazione risolva il problema in grandi linee, lasciando solo problemi minori da risolvere.

L'efficacia della ricerca in una gerarchia di astrazioni dipende da come si scompone e si astrae il problema da risolvere. Una volta che i problemi sono astratti e scomposti, uno qualsiasi dei metodi di ricerca potrebbe essere utilizzato per risolverli. Non è facile, tuttavia, riconoscere astrazioni o scomposizioni utili per un dato problema.

Programmazione dinamica

La programmazione dinamica è un metodo per l'ottimizzazione che consiste nel memorizzare le soluzioni parziali ai problemi. Le soluzioni che sono già state trovate possono essere recuperate piuttosto che essere ricalcolate. La programmazione dinamica può essere utilizzata per trovare percorsi in grafici finiti costruendo una funzione $cost\_to\_goal$ che fornisca il costo esatto di un percorso a costo minimo dal nodo a un obiettivo.

Sia $cost\_to\_goal(n)$ il costo effettivo del percorso a costo minimo dal nodo n ad un nodo goal, $cost\_to\_goal(n)$ può essere definita come segue:

$$cost\_to\_goal(n) = \begin{cases} 0 & \text{se } goal(n) \\ min_{<n,m>\in A}(cost(< n, m >) + cost\_to\_goal(m)) & \text{altrimenti} \end{cases}$$

L'idea generale consiste nel creare una tabella offline di $cost\_to\_goal(n)$ per ogni nodo n. Questo può essere possibile effettuando una ricerca *lowest-cost-first-search*, con una *multiple-path-pruning*, partendo dai nodi obiettivo nel grafo inverso (ovvero con tutti gli archi invertiti).

L'algoritmo di programmazione dinamica, invece di avere un obiettivo da cercare, memorizza i valori di $cost\_to\_goal$ per ogni nodo trovato. In sostanza, la programmazione dinamica funziona a ritroso rispetto all'obiettivo, costruendo i percorsi a costo minimo per l'obiettivo da ciascun nodo nel grafico.

Una *policy* è una specifica di quale arco prendere da ogni nodo. Data la funzione $cost\_to\_goal$, che è calcolata offline, un criterio può essere calcolato come segue: dal nodo n dovrebbe andare ad un nodo vicino m che minimizza

$$ost(< m, n >) + cost\_to\_goal(m)$$

Questa policy porterà l'agente da un qualsiasi nodo a un nodo goal seguendo un percorso a costo minimo.

Utilità La programmazione dinamica può essere usata per costruire funzioni euristiche che possono essere usate per ricerche A* e branch-and-bound. Come detto in precedenza, un modo per costruire una funzione euristica è quello di semplificare il problema (ad esempio, omettendo alcuni dettagli) fino a quando il problema semplificato è abbastanza piccolo. La programmazione dinamica può essere usata per trovare il costo di un percorso ottimale ad un obiettivo nel

problema semplificato. Queste informazioni formano un database di pattern che può essere usato come euristico per il problema originale.

La programmazione dinamica è utile quando:

- i nodi obiettivo sono espliciti
- è necessario un percorso di costo minimo
- il grafo è finito e piccolo abbastanza da rende possibile la memorizzazione del valore di $cost\_to\_goal$ per ogni nodo
- i nodi obiettivo non cambia molto spesso

I problemi principali con la programmazione dinamica sono:

- funziona solo per grafi finiti e tabelle sufficientemente piccole
- un agente deve ricalcolare una policy per ogni singolo nodo.

Capitolo 3

Ragionamento con vincoli

3.1 Mondi possibili, Variabili e Vincoli

Invece di ragionare esplicitamente in termini di stati, è in genere meglio descrivere gli stati in termini di *features* (caratteristiche) e di ragionare in termini di tali features. Le features sono descritte utilizzando delle *variabili* che spesso non sono indipendenti e si stabiliscono degli *hard constraints* (vincoli rigidi) che specificano le combinazioni legali di assegnazioni di valori alle variabili. Le preferenze rispetto alle assegnazioni sono specificate in termini di *soft constraint* (vincoli flessibili). In tale contesto il ragionamento è inteso come la generazione di assegnazioni che soddisfino i vincoli rigidi e ottimizzino i vincoli flessibili.

3.1.1 Variabili e Mondi

I problemi di soddisfazione dei vincoli (CSP) sono descritti in termini di variabili e mondi possibili. Un *mondo possibile* è un modo possibile in cui il mondo (il mondo reale o un mondo immaginario) potrebbe essere. I mondi possibili sono descritti da variabili algebriche. Una variabile algebrica è un simbolo usato per indicare le caratteristiche dei mondi possibili. I nomi delle variabili algebriche iniziano con una lettera maiuscola. Ogni variabile algebrica X ha un dominio associato, $dom(X)$, che è l'insieme di valori che la variabile può assumere.

Le variabili si distinguono in *variabili discrete* e *variabili continue*. Le variabili discrete hanno un dominio finito o infinito numerabile. Un esempio di variabile discreta è la variabile booleana. Le variabili continue sono variabili con dominio infinito non numerabile, ad esempio, il dominio può corrispondere all'insieme dei numeri reali o ad un suo intervallo.

Dato un insieme di variabili, un *assegnazione* su tale insieme è una funzione dalle variabili ai rispettivi domini. Scriveremo un assegnazione sull'insieme $\{X_1, X_2, \ldots, X_k\}$ come $\{X_1 = v_1, X_2 = v_2, \ldots, X_k = v_k\}$, dove $v_i \in dom(X_i)$ per ogni i. Ad ogni variabile può essere assegnato un solo valore e se tutte le variabili sono avvalorate si parla di assegnazione totale, parziale altrimenti.

Quindi possiamo definire un *mondo possibile* come un assegnamento totale. Se il mondo w è l'assegnazione $\{X_1 = v_1, X_2 = v_2, \ldots, X_k = v_k\}$, diremo che la variabile X_i ha il valore v_i nel mondo w.

Se abbiamo n variabili, ognuna delle quali ha dominio di cardinalità d allora ci sono d^n mondi possibili. Uno dei vantaggi principali di ragionare in termini di variabili è la possibilità di descrivere molti mondi con poche variabili, infatti con sole 10 variabili in dominio con cardinalità 2 si possono descrivere $2^{10} = 1024$ possibili mondi.

3.1.2 Vincoli

In molti domini, non tutte le possibili assegnazioni sono ammissibili. Un vincolo rigido distingue le assegnazioni lecite da quelle illegali.

Uno *scope* (o ambito) è il sottoinsieme delle variabili coinvolte. Una *relazione* su uno scope S è una funzione booleana che va dall'assegnazione su S in $\{true, false\}$. Cioè, specifica se ogni assegnazione è lecita. Un constraint c è costituito da uno scope S e una relazione su S. Si dice che un constraint coinvolga ciascuna delle variabili nel suo ambito.

Un constraint può essere valutato su ogni assegnamento che estenda il suo scope. Considerando il constraint c su S, l'assegnazione A su S', con $S \subseteq S'$, soddisfa c se A, ristretto a S, è $true$ per la relazione, altrimenti A viola c.

Un mondo possibile w soddisfa un insieme di constraints se ogni constraint è soddisfatto dai valori assegnati in w alle variabili nello scope del constraint. In questo caso diremo che il mondo possibile è un *modello* dei vincoli. Cioè, un modello è un mondo possibile che soddisfa tutti i vincoli.

I vincoli possono essere definiti *intensionalmente*, in termini di formule, oppure *estensionalmente*, elencando tutto le assegnazioni lecite. I vincoli possono essere unari ($B \leq 3$), binari ($B \geq A$) o n-ari.

3.1.3 Constraint Satisfaction Problems

Un constraint satisfaction problem (CSP) consiste in:

- Un insieme di variabili
- Un dominio per ogni variabile
- Un insieme di vincoli.

Un CSP è finito se ha un numero finito di variabili di dominio finito o infinito numerabile, ma possono essere definiti anche CSP infiniti.

Dato un CSP, è possibile effettuare svariati compiti:

- Determinare se esista o meno un modello
- Trovare un modello
- Constare il numero di modelli

- Enumerare i modelli
- Trovare il modello migliore, data una misura di qualità
- Determinare se alcuni enunciati siano veri in tutti i modelli.

Osservazione Alcuni metodi di risoluzione di un CSP possono determinare se esiste un modello, ma, se non esiste, non possono determinare che non esiste. Determinare se esiste un modello per un CSP con domini finiti è NP-completo e non esistono algoritmi noti per risolvere tali problemi che non usano il tempo esponenziale nel peggiore dei casi. Tuttavia, solo perché un problema è NP-completo non significa che tutte le istanze siano difficili da risolvere.

3.2 Algoritmo Generate-and-Test

Un CSP finito potrebbe essere risolto da un algoritmo esaustivo come il *generate and test*. Lo spazio delle assegnazioni totali si indica con D e l'algoritmo ne controlla ogni elemento e restituisce la prima assegnazione che soddisfa tutti i vincoli oppure tutte le assegnazioni soddisfacenti.

Si tratta di un algoritmo abbastanza stupido e poco utile, infatti se si hanno n domini di dimensione d, allora D ha d^n elementi. Al crescere di n D crescerebbe esponenzialmente rendendo il problema intrattabile.

3.3 Risoluzione di CSP tramite ricerca

Gli algoritmi generate-and-test assegnano valori a tutte le variabili prima di controllare che i vincoli sia rispettati. Poiché i vincoli individuali coinvolgono solo un sottoinsieme delle variabili, alcuni vincoli possono essere testati prima che a tutte le variabili siano stati assegnati valori. Se un'assegnazione parziale è incoerente con un vincolo, anche qualsiasi assegnazione totale che estende l'assegnazione parziale sarà incoerente.

Un'alternativa al generate-and-test consiste nel costruire uno spazio di ricerca per le strategie di ricerca a grafo definito come segue:

- I nodi sono le assegnazioni parziali $n = \{X_1 = v_1, \ldots, X_k = v_k\}$
- I vicini di un nodo n sono tutti quei nodi per cui $\{X_1 = v_1, \ldots, X_k = v_k, Y = y_i\}$ per ogni $y_i \in Dom(Y)$ e $Y \notin (X_1, \ldots, X_k)$
- Il nodo di partenza è l'assegnazione vuota
- I nodi goal rappresentano assegnazioni totali (che possono esistere solo che tutti i vincoli sono rispettati).
- La soluzione è rappresentata dai nodi rappresentanti assegnazioni consistenti.

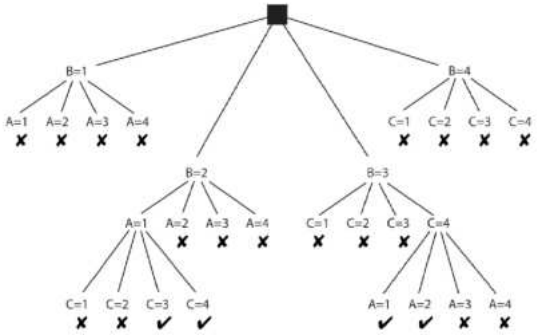


Figura 3.1: Possibile albero di ricerca per il CSP

È un caso di problema di ricerca in cui siamo interessati ai nodi obiettivo e non al percorso effettuato per raggiungerli.

La ricerca in questo albero con una ricerca depth-first search, tipicamente chiamata backtracking, può essere molto più efficiente del generate-and-test. Quest'ultimo è equivalente a controllare i vincoli solo dopo aver raggiunto le foglie. Al contrario, anticipare il controllo dei vincoli permette di potare grandi sottoalberi con un notevole guadagno in termini di prestazioni.

3.4 Algoritmi basati su consistenza

Anche se la ricerca backtracking sullo spazio di ricerca delle assegnazioni è di solito un miglioramento sostanziale rispetto al generate-and-test, presenta ancora varie inefficienze dovute al fatto che durante il backtrack possono essere riscoperte violazioni di vincoli già trovate, le quali si possono facilmente risolvere una volta per tutte eliminando il valore che provoca tutte le inconsistenze. Questa è l'idea alla base degli algoritmi basati su consistenza.

Gli algoritmi basati su coerenza sono meglio pensati per lavorare sulla rete di vincoli (constraint network) indotta dal CSP, ove:

- vi è un nodo circolare per ogni variabile
- vi è un nodo rettangolare per ogni vincolo
- vi è un arco $< X, c >$ per ogni vincolo c ed ogni variabile X nell'ambito di c.

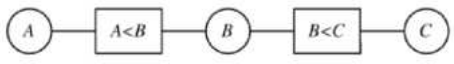


Figura 3.2: Esempio di constraint network con variabili A, B, C e vincoli $A < B$ e $B < C$

Quando un vincolo ha solo una variabile nel suo ambito, l'arco $< X, c >$ si dice *consistente rispetto al dominio* se ogni possibile valore di X soddisfa c.

Ad esempio il vincolo $B \neq 3$ ha ambito $\{B\}$. Con questo vincolo, e con $D_B = \{1, 2, 3, 4\}$, l'arco $< B, B \neq 3 >$ non è consistente rispetto al dominio perché $B = 3$ viola il vincolo. Se il valore 3 venisse eliminato dal dominio di B, allora l'arco diverrebbe consistente rispetto al dominio.

Ora supponiamo che il vincolo c abbia ambito $\{X, Y_1, \ldots, Y_k\}$. L'arco $< X, c >$ si dice *arco consistente* se, per ogni valore $x \in D_X$, esistono valori $y_1, \ldots, y_k$ con $y_i \in D_{Y_i}$, tale che il vincolo $c(X = x, Y_1 = y_1, \ldots, Y_k = y_k)$ sia soddisfatto.

Una rete di archi si dice consistente se tutti gli archi che la compongono sono consistenti. Nel rendere consistente una rete è necessario fare attenzione che l'eliminazione di un elemento dal dominio di una variabile non renda inconsistenti archi che primo lo erano.

3.4.1 Algoritmo basato sulla consistenza degli archi (GAC)

L'idea alla base dell'algoritmo della *consistenza generalizzata degli archi* è rendere consistente la rete restringendo i domini delle variabili. Questo algoritmo considera un insieme *to_ do* di archi potenzialmente inconsistenti che inizialmente contiene tutti gli archi della rete. Finché l'insieme non è vuoto, viene rimosso un arco $< X, c >$ e se si tratta di un arco inconsistente viene reso consistente riducendo il dominio di X. Ora, tutti gli archi resi inconsistenti dalla restrizione del dominio di X vengono nuovamente aggiunti all'insieme *to_ do*.

Indipendentemente dall'ordine in cui gli archi sono considerati, l'algoritmo terminerà sempre con una rete coerente e domini ridotti. Le terminazioni possibili sono tre:

1. un dominio diventa vuoto, quindi il CSP non ha soluzioni
2. tutti i domini sono ridotti ad un solo valore, quindi esiste un'unica soluzione
3. si ottiene una rete semplificata con variabili con domini multivalore su cui applicare altri algoritmi (come quello che segue).

3.5 Separazione dei domini

Un altro metodo per semplificare la rete è domain splitting. L'idea è quella di dividere un problema in una serie di casi disgiunti da risolvere separatamente. La soluzione al problema iniziale è l'unione delle soluzioni a ogni caso.

Nel caso più semplice, consideriamo una variabile binaria X il cui dominio sia $\{t, f\}$. Tutte le soluzioni hanno o $X = t$ o $X = f$, quindi per ottenere tutte le soluzioni è sufficiente trovare prima le soluzioni con $X = t$ e poi quelle con $X = f$. Se siamo interessati ad una sola soluzione, allora possiamo controllare il secondo caso solo se il primo non ha soluzione.

Nel caso in cui, invece, il dominio delle variabili contiene più di due valori, allora lo si può splittare in diversi modi. Suddividere il dominio in tante parti quanti sono i valori contenuti porterebbe ad ottenere grandi progressi con un solo split, ma suddividere l'insieme in sottoinsiemi più numerosi permetterebbe di potare più rami in un colpo, senza dover considerare singolarmente ogni valore. Bisognerebbe trovare un giusto compromesso tra le due opzioni.

Un modo effettivo di risolvere un CSP è di integrare l'approccio basato sulla consistenza in un algoritmo ricorsivo, ovvero:

- selezionare il problema in input tramite GAC
- se non è risolto direttamente
  - selezionare una variabile con dominio almeno binario
  - partizionare il dominio in un certo numero di casi (sotto-problemi)
  - risolvere ricorsivamente tali casi

3.6 Eliminazione di variabili

Come abbiamo visto, il GAC semplifica la rete rimuovendo le variabili, un metodo analogo è il *variable elimination* (VE).

L'idea di VE è quella di rimuovere le variabili una per una. Quando si rimuove una variabile X, VE crea un nuovo vincolo sulle variabili rimanenti che rifletta gli effetti di X su tutte le altre variabili. Questo nuovo vincolo sostituisce tutti i vincoli che coinvolgono X, formando una rete ridotta che non coinvolge X. Il nuovo vincolo è costruito in modo che qualsiasi soluzione al CSP ridotto possa essere estesa a una soluzione del CSP che contiene X. Oltre alla creazione di un nuovo vincolo, VE fornisce un modo per costruire una soluzione al CSP iniziale partendo da una soluzione al CSP ridotto.

Eliminazione di una variabile

Il seguente algoritmo è descritto utilizzando le operazioni di join e proiezione dell'algebra relazionale. Data una variabile X da eliminare, l'influenza di X sulle restanti variabili è data dalle *relazioni* dei vincoli che la coinvolgono.

Prima l'algoritmo raccoglie tutti i vincoli che coinvolgono X e li rappresenta nella relazione $r_x(X, \bar{Y})$, dove $\bar{Y}$ è l'insieme delle variabili nello scope di r_x, ovvero le vicine di X nel grafo dei vincoli. Quindi l'algoritmo proietta r_x su $\bar{Y}$ ottenendo una relazione che sostituisce tutte le relazioni che coinvolgono X. A questo punto si ottiene un CSP ridotto che coinvolge una variabile in meno, da risolvere ricorsivamente.

Una volta ottenuta una soluzione per il CSP ridotto, l'algoritmo estende tale soluzione ad una soluzione per il CSP originale tramite join con r_x per aggiungere le assegnazioni a X.

Esempio Consideriamo un CSP che contenga le variabili A, B, C ognuna con dominio {1,2,3,4}. Supponiamo che i vincoli che coinvolgano B siano $A < B$ e $B < C$.

Per eliminare B, prima avviene il join tra le relazioni dei vincoli su B:

A	B
1	2
1	3
1	4
2	3
2	4
3	4

$\bowtie$

B	C
1	2
1	3
1	4
2	3
2	4
3	4

$=$

A	B	C
1	2	3
1	2	4
1	3	4
2	3	4

La proiezione di questa ultima relazione sulle colonne A e C induce una nuova relazione senza B:

A	C
1	3
1	4
2	4

Questa relazione su A e C sostituisce i vincoli imposti su B. I vincoli originali su B sono rimpiazzati con i nuovi vincoli su A e C. VE quindi risolve la nuova rete che è ora più semplice perché non coinvolge la variabile B. Per generare una o tutte le soluzioni, l'algoritmo memorizza la relazione di join su A, B e C per costruire una soluzione che coinvolga B a partire dalla soluzione di un problema ridotto.

L'algoritmo VE può essere combinato con algoritmi basati su consistenza usati per semplificare il problema quando si elimina una variabile, in questo modo si otterrebbero tabelle intermedie più piccole.

3.7 Ricerca locale

Gli algoritmi precedenti effettuano una ricerca sistematica dell'intero spazio delle assegnazioni (di valori alle variabili). Se lo spazio è finito, possono o trovare una soluzione o oppure provare che non ne esiste una. Molti spazi, però, sono troppo grandi o addirittura infiniti per potervi applicare una ricerca sistematica. Gli algoritmi che vedremo in questa sezione lavorano in infiniti, non applicano la ricerca sistematica, ma si tratta di metodi mediamente efficienti. Non effettuando una ricerca esaustiva non possono garantire che una soluzione venga trovata, anche quando essa esiste. Non possono garantire, pertanto, che non esista alcuna soluzione. Solitamente trovano applicazione in contesti per i quali si prevede che esistano delle soluzioni.

Il metodo della ricerca locale, o *local search* inizia con un assegnazione totale di un valore ad ogni variabile e prova a migliorare tale assegnazione iterativamente per passi di miglioramento, per passi casuali o tramite ripartenze con assegnazioni differenti.

```
procedure Local_search(Vs,dom,Cs)
   Input
      Vs: insieme di variabili
      dom: funzione che restituisce il dominio di una
      variabile
      Cs: insieme di vincoli da soddisfare
   Output
      assegnazione totale che soddisfa i vincoli
   Local
      A array di valori indicizzato sulle variabili in Vs
      (assegnazione)
   repeat // try
      for each X ∈ Vs do
         A[X] ← valore casuale da dom(X)
      // walk
      while not stop_walk() and A non soddisfa Cs do
         Selezionare Y ∈ Vs e un valore w ∈ dom(Y)
         A[Y] ← w
      if A soddisfa Cs then
         return A
   until terminazione
```

Figura 3.3: Ricerca locale per trovare una soluzione ad un CSP

Un algoritmo di ricerca locale per problemi CSP è dato in Figura 3.3. L'array A specifica l'assegnazione di una valore ad ogni variabile. Il primo for each assegna un valore ad ogni variabile. La prima volta in cui viene eseguito è chiamata *random inizialization*. Ogni iterata del ciclo più esperto è detta *try*. La seconda e le successive assegnazioni di valori causali ad ogni variabile è una cosiddetta *random restart*. Un'alternativa al random inizialization potrebbe ricorrere all'utilizzo di una euristica o di una conoscenza a priori.

Il ciclo while effettua una *ricerca locale*, o walk, nello spazio delle assegnazioni. Considera un insieme di successori per l'assegnazione totale A e ne seleziona uno che vada a diventare la prossima assegnazione totale. Per possibili successori si intendono quelle assegnazioni che differiscono dalla attuale assegnazione totale per il valore assegnato ad una sola variabile.

Questa ricerca termina quando viene trovata una soluzione (un'assegnazione totale soddisfacente) o viene soddisfatto il criterio di stop_walk(), quando ad esempio viene raggiunto un numero massimo di cicli. Questo algoritmo non garantisce la fermata. Se il criterio di stop fosse falso, potrebbe ciclare all'infinito anche se non esistono soluzioni. Anche se una soluzione dovesse esistere, però, potrebbe rimanere bloccato in una regione dello spazio. La completezza[1] dell'algoritmo di ricerca locale dipende dai criteri di selezione e di stop.

Una versione alternativa e completa di questo algoritmo prende il nome di *random sampling*. In questa versione il criterio di stop_walk() è sempre true così da non eseguire mai il ciclo while. In questo modo continua a provare assegnazioni casuali che possano soddisfare tutti i vincoli. Il random sampling

[1] Un algoritmo è completo se, quando una soluzione esiste, garantisce di trovarla.

è completo nel senso che, con abbastanza tempo a disposizione, garantisce di trovare una soluzione. Il difetto consiste nella sua lentezza.

Un'altra versione del local search è detta *random walk*. In questa versione il criterio di stop_walk() è sempre false e pertanto non vi sono random restart. Nel while, il random walk, seleziona casualmente una variabile ed le assegna un valore casuale. Si esce dal ciclo while solo quando è stata trovata un'assegnazione totale soddisfacente. Anche questa versione dell'algoritmo è completa e dal momento che riassegna una sola variabile per volta, richiede passi più brevi. Il numero dei passi da effettuare, però, dipende dalla distribuzione delle soluzioni.

3.7.1 Iterative Best Improvement

Iterative Best Improvement è un algoritmo di ricerca locale che seleziona il successore di un'assegnazione corrente che più migliora una certa *funzione di valutazione*. Se vi sono più successori adatti, se ne sceglie uno casualmente. Quando l'obiettivo è minimizzare la funzione, questo algoritmo prende il nome di *greedy descent*, se l'obiettivo è massimizzarla, allora è detto *greedy ascent*. Se si implementa uno dei due metodi, l'altro può essere ottenuto adottando la funzione di segno opposto.

Questo algoritmo richiede un metodo per valutare ogni assegnazione totale. Per i problemi di soddisfacimento dei vincoli, una funzione di valutazione comune è data dal numero di vincoli violati. Un vincolo violato si definisce *conflitto*. Quando la funzione di valutazione assume il valore nullo si è giunti alla soluzione, cioè ad un'assegnazione totale che soddisfi tutti i vincoli. A volte di preferisce raffinare tale funzione pesando i vincoli in maniera differente.

Un *ottimo locale* è un'assegnazione totale che non ha alcun successore che ne migliori la valutazione. Un *ottimo globale* è un'assegnazione con valutazione massima fra tutte le assegnazioni possibili. Un ottimo globale è anche un ottimo locale, ma ci sono ottimi locali che non lo sono globalmente.

Se la funzione è basata sul numero dei conflitti, un CSP è soddisfacibile se ha come minimo globale un numero di conflitti pari a zero, mentre un CSP non soddisfacibile prevede un minimo globale con valore positivo.

3.7.2 Randomized Algorithms

L'itertive best improvement prende casualmente un possibile successore dell'assegnazione corrente, ma può bloccarsi in un minimo locale che non sia globale.

La casualità può essere utilizzata per sfuggire a tali minimi locali in due modi:

- *Random restart*, in cui vengono scelti valori casuali per tutte le variabili dell'assegnazione. È una mossa casuale globale, quindi costosa, ma permette di ripartire da regioni completamente diverse dello spazio.

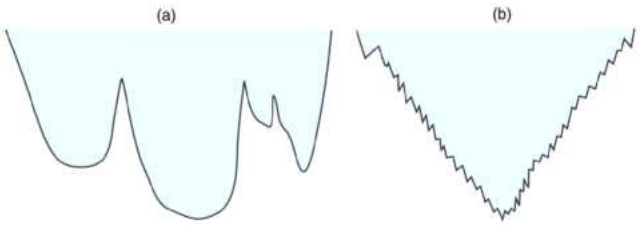


Figura 3.4: Due spazi di ricerca per la ricerca del minimo.

- *Random walk*, in cui le mosse casuali sono alternate da passi di ottimizzazione. È una mossa casuale locale che con il greedy ascent permette passi in direzione opposta per sfuggire a minimi locali.

Integrando massimo miglioramento iterativo e mosse random si ottiene un algoritmo di *ricerca locale stocastica*.

Consideriamo gli spazi di ricerca bidimensionali in Figura 3.4 in cui l'obiettivo è trovare il valore minimo e supponiamo che un possibile successore può essere trovato effettuando piccoli passi o verso destra o verso sinistra rispetto alla posizione attuale.

Il metodo più rapido per trovare il minimo globale nello spazio di ricerca *(a)* ci si aspetta che sia l'unione del *greedy descent* con il *random restart*. Il primo permette di trovare facilmente minimi locali, il secondo permette di raggiungere la parte centrale, ovvero la vallata più profonda, dove si converge rapidamente verso un minimo globale. Il *random walk* non funzionerebbe bene in quanto richiederebbe molti piccoli passi casuali per uscire da un minimo locale.

Al contrario, nello spazio di ricerca *(b)*, il *random restart* rimane bloccato a cercare tra numerosi minimi locali mentre il *random walk* con *greedy descent* potrebbe evitare tali minimi locali, inoltre pochi passi casuali spesso sono sufficienti.

3.7.3 Local Search Variants

Tabu search

Così come presentato, l'algoritmo di local search non ha memoria. Non ricorda nulla su come procede la ricerca. Un semplice modo di usare la memoria per ottimizzare la ricerca locale è applicare la *Tabu Search* che evita la modifica di assegnazioni introdotte di recente. In altre parole, quando si seleziona una variabile da riassegnare, si evita di scegliere variabili che siano state riassegnate negli ultimi t (tabu tenure) passi. Se t è piccolo si potrebbe memorizzare una lista delle variabili modificate di recente, se t è grande si può memorizzare per ogni variabile il passo in cui è avvenuto l'ultimo cambiamento.

Best Improving Step

Il metodo *Best Improving Step* (o passo di massimo miglioramento) seleziona una coppia variabile-valore che porta al miglioramento di valutazione massimale (scegliendo casualmente in caso di più coppie con lo stesso miglioramento).

Una implementazione naive di questo metodo è, data una assegnazione totale corrente, per ogni X e ogni $v \in dom(X)$ diverso dal valore corrente di X, confrontare l'assegnazione corrente con quella in cui $X = v$. Quindi si seleziona una coppia di massimo miglioramento, anche in caso di differenze negative (peggioramenti).

Una implementazione alternativa e più sofisticata prevede l'utilizzo di una coda con priorità di coppie variabile-valore pesate. Per ogni variabile X ed ogni valore $v \in dom(X)$ tale che v non è assegnata a X nell'assegnazione corrente, la coppia $< X, v >$ dovrebbe essere nella coda di priorità. Il peso w della coppia $< X, v >$ è la differenza tra la valutazione dell'assegnazione totale corrente e la valutazione dell'assegnazione con $X = v$. A ogni iterata si seleziona la coppia variabile-valore con peso minimo, ovvero il successore di massimo miglioramento. Una volta riassegnata X è necessario ricalcolare i pesi di tutte le coppie variabile-valore coinvolte in vincoli il cui soddisfacimento è mutato.

Two Stage Choice

Un'alternativa è di selezionare prima una variabile e quindi scegliere un valore per tale variabile. L'algoritmo *Two-Stage choice* (o scelta a due fasi) mantiene una coda con priorità di variabili con peso pari al numero di conflitti in cui sono coinvolte. Ad ogni passo si seleziona la variabile che partecipa a più conflitti e le si assegna un nuovo valore o casuale o che minimizzi il numero di conflitti. Anche in questo caso va ricalcolato il peso delle variabili coinvolte in vincoli il cui soddisfacimento è mutato.

Any Conflict Algorithm

Invece di scegliere la miglior variabile da riassegnare, un'alternativa ancora più semplice consiste nel selezionare ogni *variabile conflittuale*. L'algoritmo *Any-Conflict*, ad ogni passo, seleziona una delle variabili conflittuali casualmente (non necessariamente quella con più conflitti) e le assegna un valore che minimizzi il numero di conflitti oppure un valore casuale. L'assegnazione casuale fornisce ad ogni variabile la stessa probabilità di essere scelta, altrimenti la probabilità dipenderà dal numero di conflitti in cui è coinvolta.

Simulated Annealing

L'algoritmo di *Simulated Annealing* è un metodo di ottimizzazione tipicamente descritto in termini di termodinamica. Il movimento randomico corrisponde alle alte temperature, viceversa, una bassa randomicità corrisponde a basse temperature. Tale algoritmo non prevede strutture dati ausiliari in quanto seleziona

casualmente una variabile e un valore del suo dominio per poi accettarne o rigettarne l'assegnazione risultante.

Quindi il simulated annealing è un algoritmo stocastico di ricerca locale dove le temperature sono abbassate lentamente partendo da un random walk ad alta temperatura che eventualmente diventa un greedy descent a temperature prossime allo zero.

Ad ogni passo, data l'assegnazione corrente A, sceglie casualmente una variabile e le assegna valore ottenendo una nuova assegnazione A'. Se A' non peggiora aumenta il numero di conflitti, quindi non peggiora l'euristica, la sostituisce alla corrente assegnazione. Altrimenti la sostituisce con una probabilità che dipende dalla temperatura e dal peggioramento che comporta.

Per capire se accettare i passi peggiorativi, si considera un $T \in \mathbb{R}^+$, cioè un valore positivo reale che indichiamo rappresenti la temperatura. Supponiamo che A sia l'assegnazione totale corrente e supponiamo che $h(A)$ sia l'euristica da minimizzare. Tipicamente h è il numero di conflitti. Il simulated annealing seleziona casualmente un possibile successore che costituisce l'assegnazione totale A'. Se $h(A') \leq h(A)$ allora $A \leftarrow A'$, altrimenti si può accettare secondo la distribuzione di probabilità di Gibbs/Boltzmann:

$$e^{-(h(A')-h(A))/T} \tag{3.1}$$

Poiché tale probabilità è considerata solo quando $h(A') > h(A)$, l'esponente è sempre negativo. Se $h(A')$ è vicino a $h(A)$, l'assegnazione è più probabile che sia accettata. Allo stesso tempo, se la temperatura è alta, l'esponente sarà prossimo a zero e, quindi, la probabilità vicina a uno. Al decrescere della temperatura, invece, l'esponente tende a $-\infty$ e dunque la probabilità tenderà a zero.

3.7.4 Random Restart

La tecnica del *random restart* può migliorare le prestazioni di un algoritmo casuale debole (che ha successo poche volte) semplicemente eseguendolo più volte. Le performance del random restart possono anche essere predette in quanto le esecuzioni sono indipendenti l'una dall'altra. Infatti, un algoritmo con una probabilità di successo p che viene eseguito n volte, potrebbe trovare una soluzione con probabilità $1-(1-p)^n$. esso fallisce solo se falliscono tutti i tentativi ed ogni tentativo è indipendente.

3.8 Population-Based Methods

Gli algoritmi precedenti gestiscono una singola assegnazione totale (un *individuo*). Gli algoritmi basati su popolazioni gestiscono insiemi di assegnazioni, un insieme di assegnazioni è infatti detto *popolazione*.

Beam Search Beam Search è un algoritmo simile al iterative best improvement, ma conserva fino a k assegnazioni anziché una sola e riporta successo

trova un assegnazione soddisfacente. Ad ogni passo, seleziona i migliori k possibili successori dell'individuo corrente (o li seleziona tutti se non ce ne sono a sufficienza) ed in caso di parità li seleziona casualmente. Quindi ripete con il nuovo set di k assegnazioni.

Questo algoritmo è utile in caso di memoria limitata in quanto permette di scegliere k in base alla memoria disponibile.

Stochastic Beam Search Stochastic Beam Search è una variante di Beam Search in cui, al posto di scegliere i migliori k individui, sceglie k individui randomicamente, consapevoli che gli individui con valutazione migliore hanno più probabilità di essere selezionati. Questo è possibile ottenendo la probabilità di essere scelti in funzione della funzione di valutazione (la scelta dipende dall'euristica). Un'implementazione standard calcola la probabilità di scegliere un individuo applicando la distribuzione di Gibbs/Boltzmann, ottenendo che:

$$e^{-h(A)/T} \tag{3.2}$$

dove $h(A)$ è la funzione di valutazione euristica e T è la temperatura.

Rispetto al Beam Search normale, questa versione tende a consentire più diversità nei k individui che costituiscono la popolazione. Infatti, h riflette l'adattamento (*fitness*) di un individuo e come in biologia, maggiore è il fitness di un individuo, tanto maggiore è la probabilità di passare i propri geni alle generazioni future. Può anche accadere che uno stesso individuo venga scelto più volte.

Genetic Algorithms Gli algoritmi genetici sono simili al Stochastic Beam Search, ma ogni individuo della popolazione è una combinazione di una coppia di individui genitori. Nello specifico, gli algoritmi genetici utilizzano una operazione conosciuta come *crossover* in cui si seleziona un paio di individui e quindi si crea un nuovo figlio (offspirng) copiando parte delle assegnazioni alle variabili da un genitore e il resto dall'altro.

Un algoritmo genetico gestisce una popolazione di k individui (con k pari). Ad ogni passo, viene creata una nuova generazione di k individui come segue:

- Seleziona casualmente coppie di individui, dove le più adatte vengono scelte con probabilità maggiore e la probabilità dipende dall'incremento del fitness e dalla temperatura.
- Per ogni coppia effettua un crossover
- Fa mutare casualmente alcuni valori randomici per variabili scelte a caso. È un passo di random walk.

Una volta creati i k individui, si passa alla generazione successiva.

L'operazione di crossover che avviene negli algoritmi genetici può essere *uniforme* se considera due genitori e genera due figli. Nei figli, per ogni variabile, il valore viene scelto casualmente copiandolo da uno dei genitori. Invece, il crossover *one-point* assume un ordinamento tra tra le variabili. Viene selezionato

un indice i casuale. Un figlio viene generato selezionando i valori per le variabili fino a i da un genitore e per le successive dall'altro genitore. L'altro figlio viene generato in maniera complementare

3.9 Ottimizzazione

Invece che avere possibili mondi che soddisfano i vincoli oppure no, spesso può capitare di avere una relazione di preferenza sui possibili mondi e vorremmo il miglior mondo possibile secondo tale preferenza.

Un *problema di ottimizzazione* è dato da:

- un insieme di variabili con un dominio associato
- una *funzione-obiettivo* che mappi le assegnazioni totali a numeri reali
- un *criterio di ottimalità*, che solitamente consiste nel trovare l'assegnazione totale che minimizzi o massimizzi la funzione obiettivo.

Un *problema di ottimizzazione vincolato* include vincoli rigidi che specificano le assegnazioni possibili ammissibili. L'obiettivo è trovare un'assegnazione ottimale che soddisfi i vincoli rigidi. In un problema di ottimizzazione vincolato, la funzione-obiettivo è fattorizzata in un insieme di vincoli flessibili. Un *vincolo flessibile* ha uno *scope* che è un insieme di variabili. Il vincolo flessibile è una funzione che va dal dominio delle variabili nel suo scope ad un numero reale detto *costo*. Un criterio di ottimalità tipico è la scelta dell'assegnazione totale che minimizza la somma dei costi dei vincoli flessibili.

I vincoli flessibili possono essere sommati puntualmente. La somma di due vincoli flessibili è un vincolo flessibile con scope che è l'unione dei due scope iniziali. Il costo di ogni assegnazione alle variabili nello scope è la somma dei costi delle assegnazioni nei vincoli flessibili.

I problemi di ottimizzazione presentano una difficoltà, è difficile sapere quando un'assegnazione è ottimale. Se nei CSP un algoritmo può controllare se un'assegnazione è una soluzione semplicemente considerando la soddisfazione di tutti i vincoli (hard), nei problemi di ottimizzazione un algoritmo può determinare se un assegnazione è ottimale solo confrontandolo con altre assegnazioni.

3.9.1 Metodi sistematici per l'ottimizzazione (Cenni)

Un modo di trovare l'assegnazione minimale, corrispondente all'algoritmo *generate and test* per i CSP, consiste nel calcolare la somma dei vincoli flessibili e selezionare l'assegnazione con il costo minimo. Proprio come il generate and test per i CSP, si tratta di un metodo che ha senso solo per problemi semplici in quanto la sua complessità lo renderebbe intrattabile per problemi più grandi.

3.9.2 Ricerca Locale per l'Ottimizzazione

La ricerca locale è direttamente applicabile a problemi di ottimizzazione, dove viene utilizzata per minimizzare la funzione obiettivo, invece che trovare una soluzione.

Questi algoritmi di ricerca locale vengono eseguiti per un certo tempo (anche con random restar per cercare in altre parti di spazio), mantenendo sempre l'assegnazione migliore e restituendolo al termine.

La ricerca locale per l'ottimizzazione ha una ulteriore complicazione che non si verifica quando ci sono solo vincoli rigidi. Infatti, in presenza di soli vincoli rigidi, l'algoritmo sa di aver trovato una soluzione quando non ci sono conflitti, ma in presenza di vincoli flessibili diventa difficile determinare se un'assegnazione totale trovata sia la migliore soluzione secondo il criterio di ottimalità. Un *ottimo locale* è un'assegnazione totale migliore o almeno non peggiore, secondo il criterio di ottimalità, di ogni suo possibile successore. Un *ottimo globale* è un'assegnazione totale migliore o almeno non peggiore di tutte le assegnazioni totali possibili. Senza una ricerca sistematica, l'algoritmo non saprebbe se la soluzione trovata migliore localmente sia un ottimo globale o se ne esista una migliore altrove nello spazio di ricerca.

Quando si usa la ricerca locale per problemi di ottimizzazione con vincoli sia rigidi che flessibili, può essere utile consentire la violazione dei vincoli rigidi per arrivare a una soluzione ottimale. Lo si può fare rendendo i costi di violazione dei vincoli rigidi alti ma finiti.

Domini Continui

Per l'ottimizzazione con domini continui, una ricerca locale diventa più complicata perché non è ovvio come definire il possibile successore di un'assegnazione totale.

Per l'ottimizzazione, dove la funzione di valutazione è continua e derivabile, può essere utilizzato un algoritmo di *gradient descent* per trovare il minimo valore (o il *gradient ascent* per trovare il massimo). Il gradient descent è come una camminata su di una collina in cui, per scendere velocemente, si cammina sui punti più ripidi. L'idea generale è che il successore di un'assegnazione totale è un passo, in discesa, proporzionale alla pendenza della funzione di valutazione h. Ovvero, il gradient descent fa passi in ogni direzione proporzionalmente al negativo della derivata parziale in quella direzione.

Se X è una variabile a valore reale con valore corrente v, il prossimo valore dovrebbe essere

$$v - \eta \frac{dh}{dX}(v) \tag{3.3}$$

dove:

- η, la *dimensione del passo*, è la costante di proporzionalità che determina quanto velocemente il gradiente scende verso il minimo. Se η è troppo grande l'algoritmo potrebbe oltrepassare il minimo, se troppo piccolo, il progresso potrebbe risultare troppo lento.

- $\frac{dh}{dX}$, è la derivata di h rispetto a X, ed è una funzione di X valutata per $X = v$. Ovvero: $\lim_{\epsilon \to 0} \frac{(h(X=v+\epsilon) - h(X=v))}{\epsilon}$

Per l'ottimizzazione multidimensionale, ovvero in presenza di più variabili, il gradient descent fa passi in tutte le dimensioni proporzionali a ogni derivata parziale. Se $< X_1, \ldots, X_n >$ sono le variabili alle quali devono essere assegnati i valori, un assegnazione totale corrisponde ad una tupla di valori $< v_1, \ldots, v_n >$. Il successore di tale assegnazione $< v_1, \ldots, v_n >$ si ottiene muovendosi in ogni direzione in proporzione alla pendenza di h in quella direzione. Il nuovo valore per X_i è:

$$v_i \leftarrow v_i - \eta \frac{\partial h}{\partial X_i}(v_1, \ldots, v_n) \tag{3.4}$$

dove η è la dimensione del passo.

Capitolo 4

Rappresentazione e Ragionamento Proposizionale

Questo capitolo considera una forma semplice di knowledge base in cui vengono descritti fatti e regole che governano il mondo. Un agente può usare tale base di conoscenza, insieme alle sue osservazioni, per determinare cos'altro deve essere vero nel mondo. Quando esso viene interrogato su ciò che deve essere vero data una base di conoscenza, risponde alla query senza generare tutti i mondi possibili.

4.1 Proposizioni

Le proposizioni sono enunciati sul mondo che forniscono vincoli su ciò che potrebbe essere vero. I vincoli potrebbero essere specificati *estensionalmente*, come tabelle di assegnazioni legali alle variabili, o *intensionalmente* in termini di formule.

Ci sono una serie di ragioni per usare proposizioni per specificare vincoli e domande:

- Le formule forniscono relazioni tra variabili più concise e leggibili dell'equivalente estensionale, ricavandone anche una maggiore efficienza del ragionamento.
- Le formule sono modulari, quindi piccole modifiche al problema si traducono in piccole modifiche alla base di conoscenza. Ne segue un debug più facile da eseguire.

4.1.1 Sintassi del Calcolo Proposizionale

Una *proposizione* è una frase, scritta in una lingua, che ha un valore di verità (cioè, è vera o falsa) in un *mondo*. Una proposizione è costruita da proposizioni atomiche usando connettivi logici.

Una proposizione atomica, o solo un atomo, è un simbolo. Per convenzione le proposizioni consistono di lettere, cifre, underscore (_) e iniziamo con una lettera minuscola (es. accesa_l1). Le proposizioni possono essere costruite da proposizioni più semplici usando connettivi logici. Una proposizione o formula logica è

- o una *proposizione atomica*
- o una *proposizione composta* del tipo: $\neg p$, $p \wedge q$, $p \vee q$, $p \rightarrow q$, ..., con p e q proposizioni.

4.1.2 Semantica del Calcolo Proposizionale

La semantica definisce il significato delle frasi di un linguaggio, ovvero specifica come mettere in corrispondenza i simboli della lingua con il mondo.

Nella semantica del calcolo proposizionale gli atomi hanno significato per qualcuno e sono o vere o false nelle interpretazioni. Dalla verità degli atomi si ricava la verità delle proposizioni nelle interpretazioni.

Un'*interpretazione* consiste in una funzione π che mappa gli atomi in $\{true, false\}$. Se $\pi(a) = true$, l'atomo a è vero nell'interpretazione π, se $\pi(a) = false$, l'atomo a è falso nell'interpretazione π. Se una proposizione composta è vera in un'interpretazione viene dedotto usando le tabella di verità dei connettivi logici presenti nella proposizione e dai valori di verità dei componenti della proposizione. Si noti che i valori di verità sono definiti solo rispetto alle interpretazioni; stesse proposizioni possono avere diversi valori di verità in diverse interpretazioni.

Una *base di conoscenza* è un insieme di proposizioni dichiarate vere. Un elemento della base di conoscenza è un *assioma*. Un *modello* di base di conoscenza KB è un'interpretazione in cui tutte le proposizioni in KB sono vere. Se KB è una base di conoscenza e g è una proposizione, diremo che g è una conseguenza logica della KB se g è vera in ogni modello della KB e scriviamo

$$KB \vDash g$$

Ovvero, $KB \vDash g$ significa che non esistono modelli della KB in cui g è falsa.

La prospettiva dell'ingegnere della conoscenza

L'idea di base dietro l'uso della logica è che, quando un progettista di base di conoscenza ha un mondo particolare da caratterizzare, può scegliere quel mondo come interpretazione intesa, scegliere significati per i simboli rispetto a quel mondo, e scrivere proposizioni su ciò che è vero in quel mondo (*assiomatizzazione*). Quando il sistema calcola una conseguenza logica di una base di conoscenze, il progettista può interpretare questa risposta rispetto all'interpretazione desiderata. Un designer dovrebbe comunicare questo significato ad altri designer e utenti in modo che possano anche interpretare la risposta rispetto al significato dei simboli. La specifica del significato dei simboli viene chiamata *ontologia*.

La prospettiva della macchina

L'ingegnere della base di conoscenza che fornisce informazioni al sistema ha un'interpretazione intesa e interpreta i simboli in base a tale interpretazione. Egli afferma la conoscenza, in termini di proposizioni, di ciò che è vero nell'interpretazione intesa. Il computer, però, non ha accesso all'interpretazione prevista, ma solo alle proposizioni nella base di conoscenza. Il computer è in grado di dire se qualche affermazione è una conseguenza logica della base di conoscenza.

Se il progettista della base di conoscenza mente, cioè se alcuni assiomi sono falsi nell'interpretazione intesa, non vi è garanzia che le risposte del computer siano vere nell'interpretazione intesa.

È molto importante capire che il computer non conosce il significato dei simboli. È l'umano che dà significato ai simboli. Tutto ciò che il computer conosce del mondo è ciò che gli viene detto del mondo. Tuttavia, poiché il calcolatore può fornire le conseguenze logiche della base di conoscenza, può fornire le conclusioni che sono vere nel mondo.

4.2 Vincoli Proposizionali

La classe dei *CSP proposizionali* sono caratterizzati da:

- Variabili booleane
- Vincoli clausali (o clausole), cioè espressioni logiche della forma: $l_1 \vee l_2 \vee \cdots \vee l_k$, in cui ogni l_i è un *letterale*. un letterale è un atomo o la negazione di un atomo. Una clausola è *soddisfatta* in un possibile mondo se e solo se almeno un letterale è vero in tale mondo.

In termini di calcolo proposizionale, un insieme di clausole è una forma limitata di formule logiche. Qualsiasi formula proposizionale può essere convertita in forma clausale.

In termini di vincoli, una clausola è un vincolo su un insieme di variabili booleane che esclude una delle assegnazioni dalla considerazione - l'assegnazione che rende tutti i letterali falsi.

4.3 Clausole Definite Proposizionali

Il linguaggio delle clausole definite proposizionali è un sottolinguaggio del calcolo proposizionale che non ammette ambiguità o incertezza. In questo linguaggio, le proposizioni hanno la stessa semantica che hanno nel calcolo proposizionale, ma non tutte le proposizioni composte sono permesse in una base di conoscenza.

Infatti, una *clausola definita* è una disgiunzione di n letterali di cui soltanto uno è positivo. Ad esempio, dati tre letterali a, b, c, una clausola definita è composta da due letterali negativi $\neg a, \neg b$ e da un letterale positivo c separati tra loro da un operatore di disgiunzione (OR, $\vee$). Una clausola definitiva può

essere scritta nella seguente forma: $\neg a \vee \neg b \vee c$. La clausola definita può essere trasformata in un'implicazione composta dalla congiunzione (AND, $\wedge$) tra gli equivalenti letterali opposti positivi (a, b) che ha per conclusione il letterale positivo (c). Ciò è possibile applicando alla disgiunzione le regole di inferenza della legge di De Morgan. La forma dell'implicazione equivalente della clausola definita è la seguente

$$(a \wedge b) \Rightarrow c$$

La formula espressa sotto forma di implicazione è più intuitiva per l'uomo, pertanto la presenza di clausole definite rende più agevole la programmazione della base di conoscenza nei sistemi esperti e degli algoritmi di intelligenza artificiale.

Esempio 1. *La proposizione a è "non sta piovendo", la proposizione b è "c'è molto vento", la proposizione c è "gioco a tennis". La clausola definita delle proposizioni è $\neg a \vee \neg b \vee c$, ossia "piove o c'è molto vento o gioco tennis". Il verificarsi di un evento esclude l'altro. Nella forma di implicazione $(a \wedge b) \Rightarrow c$ lo stesso concetto può essere letto "se non piove e non c'è molto vento, allora gioco a tennis".*

Formalizziamo la sintassi delle clausole definite proposizionali come segue:

- Una proposizione atomica (o atomo) è definita come per il calcolo proposizionale
- Una *clausola definita* è nella forma $h \leftarrow a_1 \wedge \cdots \wedge a_m$ dove h è un atomo, detto *testa* della clausola, e ogni a_i è un atomo. Si può leggere come "h *se* a_1 *and* ... *and* a_m"

  Se $m > 0$, la clausola è chiamata *regola*, dove $a_1 \wedge \cdots \wedge a_m$ è il *corpo* della clausola

  Se $m = 0$, la freccia può essere omessa e la clausola prende il nome di *clausola atomica* o *fatto*. Il corpo è vuoto.
- Una *base di conoscenza* è l'insieme delle clausole definite.

Le seguenti non sono clausole definite:

$\neg apple\_is\_eaten$ manca la testa
$apple\_is\_eaten \wedge bird\_eats\_apple$ testa non atomica
$Apple\_is\_eaten \leftarrow Bird\_eats\_apple$ nomi degli atomi in maiuscolo non ammessi
$happy \vee sad \vee \neg alive$ testa non atomica, equivale a $happy \vee sad \leftarrow alive$

Una clausola definita $h \leftarrow a_1 \wedge \cdots \wedge a_m$ è falsa nell'interpretazione I se $a_1 \ldots a_m$ sono tutte vere in I e h è falsa in I, altrimenti la clausola definita è vera in I.

4.3.1 Domande e Risposte

Costruire la descrizione di un mondo ci permette di determinare cos'altro sia vero in questo mondo. Dopo aver fornito ad un computer una base di conoscenza riguardo uno specifico dominio, un utente potrebbe voler porre al sistema domande sul dominio. Il computer può rispondere decidendo se una proposizione è conseguenza logica della KB oppure no.

Una *query* è un modo per chiedere se una proposizione è conseguenza logica di una data knowledge base. Una query è nella forma

$$\mathsf{ask}\ b$$

dove b è un atomo o una congiunzione di atomi (cioè il corpo di una regola in forma di clausola definita).

Una query è una domanda che ha come riposte "*yes*" se il corpo è una conseguenza logica della KB, oppure "*no*" se il corpo non è conseguenza logica della KB. Non significa che b sia falso, ma che per la conoscenza disponibile è impossibile determinare la verità.

4.3.2 Dimostrazioni

Finora abbiamo specificato cos'è una risposta, ma non come può essere calcolata. La definizione di $\models$ specifica quali proposizioni dovrebbero essere conseguenze logiche di una base di conoscenza, ma non come calcolarle. Il problema della *deduzione* è determinare se una proposizione è una conseguenza logica di una base di conoscenza. La deduzione è una forma specifica di *inferenza*.

Una *dimostrazione* è una dimostrazione meccanicamente derivabile che una proposizione deriva logicamente da una base di conoscenza. Un *teorema* è una proposizione dimostrabile. Una *procedura di dimostrazione* è un - forse non deterministico - algoritmo per derivare conseguenze di una base di conoscenza.

Data una procedura di dimostrazione, $KB \vdash g$ significa che g può essere dimostrato o derivato dalla base di conoscenza KB[1]. Una procedura di dimostrazione è *consistente* (o *sound*) rispetto a una semantica se tutto ciò che può essere derivato da una base di conoscenza è una conseguenza logica della base di conoscenza. Cioè, se $KB \vdash g$, allora $KB \models g$.

Una procedura di prova è *completa* rispetto a una semantica se c'è una dimostrazione per ogni conseguenza logica della base di conoscenza. Cioè, se $KB \models g$, allora $KB \vdash g$.

Vediamo due modi per costruire dimostrazioni per le proposizioni definite: una procedura bottom-up e una procedura top-down.

Dimostrazioni Bottom-Up

Una *procedura di dimostrazione bottom-up*[2] può essere utilizzata per derivare tutte le conseguenze logiche di una base di conoscenze. La procedura di prova

[1] $\vdash$ rappresenta l'asserzione logica, nota anche come *sequente*

[2] Si chiama bottom-up come analogia alla costruzione di una casa, dove ogni parte della casa è costruita sulla struttura già completata.

bottom-up si basa su atomi che sono già stati stabiliti. Al contrario, un approccio top-down parte da una query e cerca di trovare clausole definite che supportano la query.

L'idea generale si basa su una *regola di derivazione*, che è una forma generalizzata della regola di inferenza chiamata *modus ponens*[3]:

Definizione 1. Se "$h \leftarrow a_1 \wedge \cdots \wedge a_m$" è definita nella KB, ed ogni a_i è già stato provato, allora si può derivare h.

Ogni clausola atomica ($m = 0$) si considera immediatamente provata.

La procedura bottom-up è implementabile calcolando un *insieme delle conseguenze* C di un insieme di clausole definite. Sotto questa procedura, se g è un atomo, $KB \vdash g$ se $g \in C$. Per le congiunzioni di atomi, $KB \vdash g_1 \wedge \cdots \wedge g_k$, se $\{g_1, \ldots, g_k\} \in C$.

Questa procedura presenta le seguenti caratteristiche:

- *Consistenza* (o *soundness*), ogni atomo in C è conseguenza logica della KB, ovvero, se $KB \vdash g$ allora $KB \models g$.
- *Punto fisso*, il C finale generato dall'algoritmo è un *punto fisso minimale* perché ogni altra applicazione della regola di derivazione non cambia C e non esistono punti fissi più piccoli. Sia I l'interpretazione in cui ogni atomo del punto fisso minimo è vero ed ogni atomo non incluso nel punto fisso minimo è falso, allora I è detto *modello minimale* nel senso che contiene il più minor numero di proposizioni vere.
- *Completezza*, supposto $KB \models g$, allora g è vera in ogni modello di KB, quindi anche nel modello *minimo*, ovvero C. Pertanto $KB \vdash g$.

Esempio 2. *Supponiamo che al sistema sia data la base di conoscenza:*

$a \leftarrow b \wedge c.$
$b \leftarrow d \wedge e.$
$b \leftarrow g \wedge e.$
$c \leftarrow e.$
$d.$
$e.$
$f \leftarrow a \wedge g.$

Una sequenza dei valori assegnati a C nella procedura bottom-up è

$\{\}$
$\{d\}$
$\{e, d\}$
$\{c, e, d\}$
$\{b, c, e, d\}$
$\{a, b, c, e, d\}$

[3]Nella logica, il modus ponens è una semplice e valida regola d'inferenza, che afferma "Se p *implica* q è una proposizione vera, e anche la premessa p è vera, allora la conseguenza q è vera, o in notazione con operatori logici: $[(p \rightarrow q) \wedge p] \vdash q$

L'algoritmo termina con $C = \{a, b, c, e, d\}$. Quindi, $KB \vdash a$, $KB \vdash b$, e così via.

L'ultima regola nella KB non viene mai usata. La procedura di prova bottom-up non può derivare f o g. Ne deduciamo che esiste un modello della base di conoscenza in cui f e g sono entrambi falsi.

Dimostrazioni Top-Down

Una procedura dimostrativa alternativa consiste nello stabilire in maniera *top-down* se una query è conseguenza logica delle clausole definite date. Questa procedura è chiamata risoluzione SLD (*S*elezione di un atomo utilizzando una strategia *L*ineare su clausole *D*efinite proposizionali).

Questa procedura è una versione proposizionale del più generale metodo di *risoluzione*[4]ed è definita come segue:

$$[(a \leftarrow a_1 \wedge \cdots \wedge a_i \wedge \cdots \wedge a_r) \wedge (a_i \leftarrow b_1 \wedge \cdots \wedge b_m)] \vdash a \leftarrow a_1 \wedge \cdots \wedge b_1 \wedge \cdots \wedge b_1 \wedge \cdots \wedge b_m \wedge \cdots \wedge a_r$$

La procedura di dimostrazione top-down può essere intesa in termini di clausole di risposta. Una *clausola di risposta* è della forma

$$yes \leftarrow a_1 \wedge a_2 \wedge \cdots \wedge a_m$$

dove *yes* è un atomo speciale che deve essere *true* quando la risposta alla query è "yes".

Se la query è

$$\mathsf{ask}\ q_1 \wedge \cdots \wedge q_m$$

la clausola di risposta iniziale è

$$yes \leftarrow q_1 \wedge \cdots \wedge q_m$$

Ora, data una clausola di risposta, l'algoritmo seleziona un atomo, detto *sub-goal*, dal corpo della clausola di risposta, supponiamo a_1, e cerca di dimostrarlo scegliendo una clausola definita nella KB che abbia come testa l'atomo selezionato (a_1). Se non ne esistono, l'algoritmo fallisce[5].

Se la dimostrazione di un qualsiasi atomo nel corpo della clausola di risposta fallisce, diventa inutile procedere con la dimostrazione degli atomi restanti.

Il *risolvente* della suddetta clausola di risposta sulla selezione a_1 con la clausola definita

$$a_1 \leftarrow b_1 \wedge \cdots \wedge b_p$$

è la clausola di risposta

[4] Inferenza generica su clausole: $[(l_1 \vee \cdots \vee a \vee \cdots \vee l_r) \wedge (l'_1 \vee \cdots \vee \neg a \vee \cdots \vee l'_m)] \vdash l_1 \vee \cdots \vee l_r \vee l'_1 \vee \cdots \vee l'_m$

[5] Si noti l'uso dei termini "seleziona" e "sceglie", il primo indica una sistematicità nell'operazione di selezione dell'atomo da dimostrare, il secondo indica un non determinismo in fase di scelta della clausola da utilizzare nella dimostrazione.

$$yes \leftarrow b_1 \wedge \cdots \wedge b_p \wedge a_2 \wedge \cdots \wedge a_m$$

Cioè, il subgoal selezionato nella clausola di risposta viene sostituito con il corpo della clausola definita scelta.

Una *risposta* è una clausola di risposta con il corpo vuoto ($m = 0$), cioè $yes \leftarrow$.

Una *derivazione LSD* di una query ask $q_1 \wedge \cdots \wedge q_k$ da una base di conoscenza KB è una sequenza di clausole di risposta $\gamma_0, \gamma_1, \ldots, \gamma_n$ tale che

- γ_0 è al query originaria $q_1 \wedge \cdots \wedge q_k$
- γ_i è il risolvente di γ_{i-1} con una clausola definita nella KB
- γ_n è una risposta

Procedura top-down alternativa Una procedura alternativa per la dimostrazione top-down consiste nell'inizializzare un insieme G con tutti gli atomi da dimostrare presenti nel corpo della query (G contiene tutti i sub-goal). Anche in questo caso la clausola $a_1 \leftarrow b_1 \wedge \cdots \wedge b_p$ indica che il sub-goal a può essere sostituito dai sub-goal $b_1, \ldots, b_p$. Analogamente a quanto già visto, qualsiasi atomo del corpo può essere selezionato (tutti, prima o poi) e se una selezione non porta a terminare la dimostrazione, non serve tentare di selezionare un altro sub-goal. La procedura della scelta della clausola risolutiva non è deterministica. Pertanto, se ci sono scelte che portano a G vuoto, l'algoritmo ha successo, altrimenti l'algoritmo fallisce (e può rispondere "no").

Esempio 3. *Supponiamo che al sistema sia data la base di conoscenza:*

$a \leftarrow b \wedge c.$
$b \leftarrow d \wedge e.$
$b \leftarrow g \wedge e.$
$c \leftarrow e.$
$d.$
$e.$
$f \leftarrow a \wedge g.$

e viene chiesta la query:

ask $a.$

Vediamo prima derivazione vincente che corrisponde ad una sequenza di assegnazioni in cui viene scelto sempre l'atomo più a sinistra nel corpo:

$yes \leftarrow a$
$yes \leftarrow b \wedge c$
$yes \leftarrow d \wedge e \wedge c$
$yes \leftarrow e \wedge c$
$yes \leftarrow c$
$yes \leftarrow e$
$yes \leftarrow$

Vediamo ora una derivazione per la quale, nella sequenza di assegnazioni, viene scelto il corpo della seconda clausola definita avente come testa b:

$yes \leftarrow a$
$yes \leftarrow b \wedge c$
$yes \leftarrow g \wedge e \wedge c$

Questo tentativo di dimostrazione fallisce in quanto per g non ci sono regole che possano essere scelte.

Il non determinismo dell'algoritmo top-down, insieme con la strategia di selezione conduce alla creazione di un *grafo di ricerca*. I nodi vicini di un nodo $yes \leftarrow a_1 \wedge \cdots \wedge a_m$, dove a_1 è l'atomo selezionato, rappresentano tutte le possibili clausole di risposta ottenute risolvendo a_1. Vi è un vicino per ogni clausola definita avente a_1 in testa. I nodi goal della ricerca sono della forma $yes \leftarrow$. Dal momento che ci basta un qualsiasi path verso un nodo goal, non è necessario trovare il percorso minimo.

Confronto tra procedure Quando una procedura top-down deriva una risposta, le regole usate nella derivazione possono essere utilizzate per la dimostrazione bottom-up e viceversa. Questa equivalenza può essere utilizzata per dimostrare la consistenza e la completezza della procedura top-down.

Per come è definita, la procedura top-down potrebbe spendere tempo extra per riprovare lo stesso atomo più volte, concentrandosi però sugli atomi rilevanti per la query. Al contrario, la procedura bottom-up prova ogni atomo una sola volta.

Infine, notiamo che è possibile, per la procedura top-down, finire in *cicli infiniti* (a meno che non sia prevista la potatura dei cicli).

Esempio 4. *Data la knowledge base a la query seguenti:*

$g \leftarrow a.$
$a \leftarrow b.$
$b \leftarrow a.$
$g \leftarrow c.$
$c.$
ask $g.$

La procedura bottom-up termina con il punto fisso $\{c, g\}$, mentre l'algoritmo top-down con DFS semplice potrebbe continuare indefinitamente, a meno della potatura dei cicli.

La strategia di selezione dell'atomo per la risoluzione condiziona efficienza e terminazione (se la potatura non è prevista). La strategia di selezione migliore consiste nel selezionare l'atomo che più facilmente porti al fallimento. Una strategia comune, invece, consiste nell'ordinare prima gli atomi e selezionare il più a sinistra, perché questo permette di utilizzare una euristica circa la selezione degli atomi.

4.4 Questioni di Rappresentazione della Conoscenza

Un'*osservazione* è un'informazione ricevuta online da utenti, sensori o altre fonti di conoscenza. Non ci si può aspettare che gli utenti ci dicano tutto ciò che è vero. In primo luogo, non sanno ciò che è rilevante, e in secondo luogo, non sanno quale vocabolario usare. Un'ontologia che specifica il significato dei simboli e un'interfaccia utente grafica per consentire all'utente di fare clic su ciò che è vero, può aiutare a risolvere il problema del vocabolario. Tuttavia, molti problemi sono troppo grandi.

4.4.1 Interrogare l'utente

Un modo semplice per acquisire informazioni da un utente è quello di incorporare un meccanismo *ask-the-user* nella procedura di prova top-down. In un tale meccanismo, un atomo è *domandabile* (o *askable*) se l'utente può conoscere il valore della verità a run time. La procedura di prova top-down, quando ha selezionato un atomo da dimostrare, può utilizzare una clausola nella base di conoscenza per dimostrarlo, o, se l'atomo è askable, può chiedere all'utente se l'atomo è vero o meno. All'utente viene quindi chiesto solo degli atomi che sono rilevanti per la query. Ci sono tre classi di atomi che possono essere selezionate:

- gli atomi per i quali ci si aspetta che l'utente non sappia rispondere, i non askable
- atomi askable per i quali l'utente non ha ancora fornito una risposta
- atomi askable la cui risposta è già stata data e memorizzata

Una proceduta bottom-up potrebbe anche adattare la strategia ask-to-user, ma dovrebbe evitare di chiedere tutti gli atomi.

Invece di rispondere alle domande, a volte è preferibile per un utente essere in grado di specificare che c'è qualcosa di strano o insolito in corso. La normalità sarà un default che può essere sovrascritto con informazioni eccezionali (si esplorerà il concetto con l'assunzione di conoscenza completa che vedremo avanti).

4.4.2 Spiegazioni a Livello di Conoscenza

L'uso esplicito della semantica consente la spiegazione e il debug a livello di conoscenza. Per rendere un sistema utilizzabile dalle persone, il sistema non può semplicemente dare una risposta e aspettarsi che l'utente ci creda. Il sistema deve essere in grado di giustificare che la sua risposta è corretta. Lo stesso meccanismo può essere utilizzato per spiegare come il sistema ha trovato un risultato e per il debug della base di conoscenza.

Un utente può utilizzare una domanda *how* per farsi spiegare dal sistema come è stata dimostrata la risposta restituita. Il sistema fornisce in risposta

la clausola definita utilizzata per dedurre la risposta. Per qualsiasi atomo nel corpo della clausola definita, l'utente può chiedere come il sistema ha dimostrato quell'atomo.

L'utente può porre al sistema una domanda *why* in risposta a una domanda ricevuta dal sistema. Il sistema risponde dando la regola che ha prodotto la domanda. L'utente può quindi chiedere perché è stata dimostrata la testa di quella regola così da navigare l'intero albero di dimostrazione.

Infine, l'utente può porre al sistema la domanda *whynot* per chiedere perché un particolare atomo non è stato dimostrato.

4.4.3 Debugging a Livello di Conoscenza

Proprio come con altri software, le basi di conoscenza possono avere errori e omissioni. Gli esperti di dominio e gli ingegneri della conoscenza devono essere in grado di eseguire il debug di una base di conoscenze e aggiungere conoscenza. Nei sistemi basati sulla conoscenza, il debug è difficile perché gli esperti di dominio e gli utenti che hanno la conoscenza del dominio necessaria per rilevare un bug non sanno necessariamente tutto sul funzionamento interno del sistema. Chiunque stia eseguendo il debug del sistema è tenuto solo a conoscere il significato dei simboli e se specifici atomi sono veri o meno, e non ha bisogno di conoscere la procedura di prova.

Il *debug a livello di conoscenza* è un processo che consente di trovare errori nelle basi di conoscenza con riferimento solo a ciò che i simboli significano e ciò che è vero nel mondo.

Quattro tipi di errori non tattici sorgono in sistemi basati su regole.

- Viene prodotta una risposta errata; cioè, è derivato un atomo che è falso nell'interpretazione intesa.
- Una risposta che non è stata prodotta; cioè, la dimostrazione ha fallito su un particolare atomo vero quando invece avrebbe dovuto avere successo.
- Il programma entra in un ciclo infinito.
- Il sistema pone domande irrilevanti.

Premesso che, come già detto, le domande irrilevanti possono essere analizzate utilizzando le domande *why*, vediamo come debuggare gli altri tre tipi di problemi.

Risposte Errate Una risposta errata è una risposta falsa nell'interpretazione intesa. È anche chiamata errore *falso-positivo*.

Supponiamo che ci sia un atomo g che è stato dimostrato falso per l'interpretazione intesa. Allora ci deve essere una regola del tipo $g \leftarrow a_1 \wedge \cdots \wedge a_k$ nella base di conoscenza che è stata usata per provare g. Allora

- o almeno uno degli a_i è falso nell'interpretazione intesa, nel qual caso può si può risolvere ricorsivamente

- o tutti gli a_i sono veri nell'interpretazione intesa, quindi è sbagliata la regola $g \leftarrow a_1 \wedge \cdots \wedge a_k$.

In questo caso la procedura di debugging può essere effettuata mediante l'uso del comando *how*. Data una prova per un atomo g che è falso nell'interpretazione intesa, un utente può chiedere come g sia stato dimostrato. Questo restituirà la clausola definita utilizzata nella dimostrazione. Se la clausola fosse una regola, l'utente potrebbe usare *how* per chiedere circa un atomo nel corpo che era falso nell'interpretazione progettata. Ciò restituirà la regola che è stata usata per dimostrare quell'atomo. L'utente lo ripete fino a quando non viene trovata una clausola definita in cui tutti gli elementi del corpo sono veri (o non ci sono elementi nel corpo). Questa è la clausola definita errata.

L'utente o l'esperto di dominio può trovare la clausola definita *buggy* senza dover conoscere il funzionamento interno del sistema o come la prova è stata calcolata. Richiedono solo la conoscenza dell'interpretazione prevista e l'uso disciplinato di *how*.

Risposte Mancanti Il secondo tipo di errore si verifica quando una risposta attesa non viene prodotta. Un atomo g che è vero nel dominio, ma non è una conseguenza della base di conoscenza, è chiamato un errore *falso-negativo*. Per questo tipo di problema va capito perché non c'è una dimostrazione per g.

Supponiamo che g sia un atomo che dovrebbe avere una dimostrazione, ma che fallisce. Affinché la dimostrazione per g fallisca, i corpi di tutte le clausole definite con g nella testa devono fallire.

- Supponiamo che esista almeno una clausola definita per g utile per la sua dimostrazione; questo significa che tutti gli atomi nel corpo della clausola devono essere veri nell'interpretazione intesa. Poiché se g è un falso negativo il corpo ha fallito, ci deve essere almeno un atomo nel corpo che fallisce. Questo atomo è allora vero nell'interpretazione intesa, ma fallisce. Quindi possiamo fare un debug ricorsivo.

- Altrimenti, significa che non esiste nella KB una clausola definita utile per la dimostrazione di g, quindi l'utente deve aggiungere una clausola definita per g.

Cicli Infiniti Ci possono essere cicli infiniti solo se la base di conoscenza è ciclica, ovvero se contiene un atomo a tale che ci sia una sequenza di clausole definite della forma

$a \leftarrow \ldots a_1 \ldots$
$a_1 \leftarrow \ldots a_2 \ldots$
$\ldots$
$a_n \leftarrow \ldots a \ldots$

Una base di conoscenza è aciclica se c'è un'assegnazione di numeri naturali (interi non negativi) agli atomi in modo che agli atomi nel corpo di una clausola definita sia assegnato un numero più basso dell'atomo nella testa.

Se per la procedura bottom-up il problema non sussiste, in quanto una regola è selzionabile solo se al testa non è già stata derivata, il problema esiste per la procedura top-down.

Per rilevare una base di conoscenza ciclica, la procedura di dimostrazione top-down può essere modificata per mantenere l'insieme di tutti gli antenati per ogni atomo nella prova. Quindi, la dimostrazione fallisce se un atomo è nel suo insieme degli antenati.

4.5 Dimostrazione per contraddizione

Cosa si può concludere quando dalla KB deriva una proposizione contraria a quanto si osserva nella realtà? Si può ragionare ammettendo regole che producano *contraddizioni* al fine di *diagnosticare* malfunzionamenti nella KB tramite la specifica di casi *impossibili*.

4.5.1 Clausole di Horn

Il linguaggio delle clausole definite non consente di affermare una contraddizione. Tuttavia, una semplice espansione del linguaggio può consentire la dimostrazione per contraddizione.

Un *vincolo di integrità* è una clausola della forma

$$false \leftarrow a_1 \wedge \cdots \wedge a_k.$$

dove a_i sono atomi e $false$ è un atomo speciale che è falso in tutte le interpretazioni. Inoltre notiamo che $false \leftarrow a_1 \wedge \cdots \wedge a_k$ è logicamente equivalente a $\neg a_1 \vee \cdots \vee \neg a_k$

Una *clausola di Horn* è una clausola definita o un vincolo di integrità. Cioè, una clausola di Horn ha un atomo falso o normale come testa.

I vincoli di integrità permettono al sistema di dimostrare che una certa congiunzione di atomi è falsa in tutti i modelli di una base di conoscenza[6].

Esempio 5. *Consideriamo la KB:*

$false \leftarrow a \wedge b.$
$a \leftarrow c.$
$b \leftarrow d.$
$b \leftarrow e.$

O c o d è falso in ogni modello della KB. Se per assurdo fossero entrambi veri nello stesso modello I di KB, sia a che b dovrebbero essere veri in I, quindi la prima regola dovrebbe essere falsa in I, si giunge ad una contraddizione. Ragionamento analogo va fatto se o c o e sono falsi in ogni modello di KB. Quindi,

[6] Da una KB di clausole di Horn si possono derivare negazioni di atomi tramite *modus tollens*, una semplice e valida regola d'inferenza, che afferma "Se p *implica* q è una proposizione vera, e anche la conseguenza q è falsa, allora la premessa p è falsa, o in notazione con operatori logici: $[(p \rightarrow q) \wedge \neg q] \rightarrow \neg p$

$KB \models \neg c \vee \neg d$.
$KB \models \neg c \vee \neg e$.

Un insieme di clausole è *insoddisfacibile* se non ha modelli, ovvero se non esiste alcuna interpretazione in cui tutte le clausole sono vere. Una serie di clausole è *dimostrabilmente inconsistente* rispetto ad una procedura di dimostrazione se può essere derivato $false$. Se una procedura di dimostrazione è consistente e completa, una serie di clausole è dimostrabilmente incoerente se e solo se è insoddisfacente.

È sempre possibile trovare un modello per un insieme di clausole definite. L'interpretazione con tutti gli atomi true è un modello di qualsiasi insieme di clausole definite. Quindi, una base di conoscenza di clausola definita è sempre soddisfacibile. Tuttavia, un insieme di clausole di Horn può essere insoddisfacente.

Esempio 6. *L'insieme di clusole* $\{a, false \leftarrow a\}$ *è insoddisfacibile. Nessuna interpretazione soddisfa entrambe le clausole. Non possono essere vere entrambe in alcuna interpretazione.*

4.5.2 Assumibili e Conflitti

Consideriamo un sistema che ha una descrizione di come dovrebbe funzionare e alcune osservazioni. Se il sistema non funziona secondo le sue specifiche, un agente diagnostico dovrebbe identificare quali componenti potrebbero essere difettosi. Per svolgere questo compito è utile essere in grado di fare ipotesi che possono essere dimostrate false.

Un *assumibile* è un atomo che può essere assunto (come vero) in una dimostrazione per contraddizione. Una dimostrazione per contraddizione deriva da una disgiunzione della negazione degli assumibili. Con una base di conoscenza di clausole Horn e assumibili espliciti, il sistema può dimostrare una contraddizione partendo da alcune assunzioni ed estraendo quelle combinazioni di assunzioni che non possono essere tutte vere.

Se KB è un insieme di clausole di Horn, un *conflitto* della KB è un insieme di assumibili che, data la KB, implica $false$. Ovvero, $C = \{c_1, \ldots, c_r\}$ è un conflitto di KB se

$$KB \cup C \models false$$

In questo caso, una *risposta* è

$$KB \models \neg c_1, \ldots, \neg c_r$$

Un *conflitto minimale* è un conflitto tale che nessun sottoinsieme proprio sia anche un conflitto.

4.5.3 Diagnosi Basata sulla Consistenza

L'idea alla base della Diagnosi Basata sulla Consistenza (CBD) è determinare possibili guasti facendo assunzioni sul *funzionamento normale* e derivando quali

componenti potrebbero essere anormali, ovvero individuare i *conflitti* dai quali l'utente può diagnosticare il problema.

Dato un insieme di conflitti, una CBD è un insieme di assumibili con almeno un elemento in ogni conflitto.

4.5.4 Ragionamento su Clausole di Horn con Assunzioni

Vediamo una implementazione bottom-up ed una top-down per la ricerca di conflitti in KB di clausole di Horn.

Implementazione Bottom-up

Si tratta di un estensione dell'algoritmo visto per la dimostrazione Bottom-Up delle clausole definite.

In questo caso però, l'insieme delle conseguenze C non è l'insieme degli atomi che seguono da KB, ma è un insieme di coppie $< a, A >$, dove a è un atomo e A è l'insieme degli assumibili che implicano a rispetto a KB.

Inizialmente, $C = \{< a, \{a\} > | a \text{ assumibile}\}$. Quindi si usano le clausole per derivare nuove conclusioni. Se esiste una clausola $h \leftarrow b_1 \wedge \cdots \wedge b_m$ tale che per ogni b_i esiste esiste un A_i tale che $< b_i, A_i > \in C$, allora si può aggiungere $< h, A_1 \cup \cdots \cup A_m >$ a C. Per i fatti $(m = 0)$ si aggiunge $< h, \{\} >$.

Implementazione Top-Down

L'implementazione top-down è simile alla versione per clausole definite, eccetto per il fatto che la query principale da provare sia $false$ e gli atomi assumibili incontrati nella dimostrazione non vanno dimostrati ma solo raccolti per essere assunti come veri.

4.6 Assunzione di Conoscenza Completa

Un database è spesso completo nel senso che tutto ciò che non implica è falso.

La logica delle clausole definite non permette la derivazione di una conclusione da una *mancanza di conoscenza* o di una mancata dimostrazione. Non assume che la conoscenza sia completa. In particolare, la negazione di un atomo non può mai essere una conseguenza logica di una base di conoscenza a clausola definita.

L'*assunzione di conoscenza completa* presuppone che, per ogni atomo, le clausole con l'atomo come testa coprano tutti i casi in cui l'atomo è vero. Sotto questa ipotesi, un agente può concludere che un atomo è falso se non può derivare che quell'atomo è vero. Questa è anche detta *assunzione di mondo chiuso*

(CWA)[7]. In contrasto con *l'assunzione di mondo aperto (OWA)*, secondo la quale l'agente non sa tutto e quindi non può trarre conclusioni da una mancanza di conoscenza.

Completare la base di conoscenza

Dato l'atomo a e tutte le clausole della KB che lo definiscono:

$a \leftarrow b_1.$
$\dots$
$a \leftarrow b_n.$

esse possono essere riassunte da un'unica proposizione (non clausola)

$a \leftarrow b_1 \vee \dots \vee b_n$

Inoltre, l'assunzione di conoscenza completa specifica che se a è vera in qualche interpretazione allora almeno uno dei b_i deve essere vero in quella interpretazione. In altre parole

$a \rightarrow b_1 \vee \dots \vee b_n.$

Pertanto possiamo ricavare l'equivalenza

$a \leftrightarrow b_1 \vee \dots \vee b_n$

detta *completamento di Clark* delle clausole per a. Il completamento di Clark per una base di conoscenza è il completamento di ogni suo atomo. Il completamento di Clark significa che se non ci sono regole per un atomo a, il completamento di questo atomo è $a \leftrightarrow false$, cioè a è falso.

Negation as Failure

Con il completamento, il sistema può derivare negazioni, quindi è utile estendere il linguaggio per permettere la presenza di negazioni nel corpo delle clausole. Per questo scopo possiamo estendere la definizione di clausola definita permettendo anche la presenza di letterali[8] nel corpo. Scriviamo la negazione di un atomo a sotto assunzione di conoscenza completa come $\sim a$ per distinguerla dalla negazione logica classica. Chiameremo questa negazione *negation as failure (NAF)*. Sotto NAF, il corpo g è una conseguenza di KB se $KB' \vDash g$, dove KB' è il completamento di Clark di KB. Una negazione $\sim a$ nel corpo di una clausola o di una query diventa $\neg a$ nel completamento.

[7] In Prolog, "Yes" significa che l'enunciato è dimostrabilmente vero. Di conseguenza "No" significa che l'enunciato è non dimostrabilmente vero. Questo significa che tale enunciato è falso solo se assumiamo che tutta la conoscenza rilevante sia presente nel rispettivo programma Prolog. Per la semantica dei programmi Prolog solitamente si effettua questa assunzione, ovvero che la conoscenza sia completa.

[8] Un *letterale* è o un atomo o la negazione di un atomo.

4.6.1 Ragionamento Non Monotono

Una logica è *monotona* se qualsiasi proposizione che può essere derivata da una base di conoscenza può anche essere derivata quando vengono aggiunte proposizioni extra alla base di conoscenza. Cioè, aggiungere conoscenza non riduce l'insieme delle proposizioni che possono essere derivate. La logica delle clausole definite è monotona.

Una logica è *non monotona* se alcune conclusioni possono essere invalidate aggiungendo più conoscenza. La logica delle clausole definite con negazione come fallimento è non monotona. Il ragionamento non monotono è utile per rappresentare casi predefiniti. Una regola predefinita (*default*) è una regola che resta valida fintantoché non si verifichi un'eccezione.

Per esempio, per dire che normalmente b è vera se c è vera, si può scrivere la regola

$$b \leftarrow c \wedge \sim ab\_a$$

con ab_a che indica l'anormalità rispetto a qualche aspetto di a. Dato c, si può derivare b, a meno che non venga asserito ab_a che inibisce la conclusione b.

4.6.2 Procedure di Dimostrazione per la NAF

Procedura Bottom-Up con NAF

La procedura di dimostrazione BU con NAF è una modifica della sua versione per clausole definite. La differenza è che può essere aggiunto un letterale nella forma $\sim p$ nell'insieme delle conseguenze C che sono state derivate. $\sim p$ viene aggiunto a C quanto si può determinare che p deve fallire.

Il fallimento può essere determinato ricorsivamente: p fallisce quando fallisce il corpo di ogni clausola che ha p in testa. Un corpo fallisce quando uno dei letterali del corpo fallisce. Un atomo b_i fallisce se può essere derivato $\sim b_i$. Una negazione $\sim b_i$, in un corpo, fallisce se b_i può essere derivato.

Procedura Top-Down con NAF

La proceduta TD con NAF è analoga a quella delle clausole definite, ma procede per negation-as-failure. Si tratta di una procedura non deterministica che può essere implementate tramite ricerca sulle scelte che hanno successo. Quando viene selezionato un atomo negato $\sim a$, viene avviata una dimostrazione per a. Se la dimostrazione per a fallisce, $\sim a$ ha successo. Se la dimostrazione per a ha successo, allora la dimostrazione per $\sim a$ fallisce e deve tentare altre scelte (se possibile).

Si noti che questa procedura è valida solo nel caso di *fallimento finito*. In caso di divergenza non si può trarre alcuna conclusione (ad esempio $p \leftarrow p$).

4.6.3 Abduzione

L'*abduzione*[9]è una forma di ragionamento in cui vengono fatte ipotesi (o assunzioni) per spiegare le osservazioni. Per esempio, se un agente dovesse osservare che una certa luce non funziona, ipotizza che cosa sta accadendo nel mondo per spiegare perché la luce non funziona.

Nell'abduzione, un agente ipotizza ciò che può essere vero in un caso osservato. Per formalizzare l'abduzione, usiamo il linguaggio delle clausole di Horn con gli assumibili. Al sistema sono dati

- KB, un insieme di clausole di Horn;
- A, un insieme di atomi, detti assumibili, per costruire ipotesi (anche detti abducibili).

Invece di aggiungere osservazioni alla base di conoscenza, le osservazioni devono essere spiegate.

Uno *scenario* di $< KB, A >$ è un sottoinsieme H di A tale che $KB \cup H$ sia soddisfacibile. $KB \cup H$ è soddisfacibile se esiste un modello in cui ogni elemento di KB ed ogni elemento di H è vero. Questo accade nessun sottoinsieme di H è un conflitto per KB.

Una *spiegazione* di una proposizione g da $< KB, A >$ è uno scenario $H \subseteq A$ che, assieme a KB, implichi g

Cioè,

$KB \cup H \vDash g$
$KB \cup H \nvDash false$

Diagnosi Abduttiva

Nella diagnosi abduttiva, l'agente ipotizza malfunzionamenti, così come ipotizza che alcune parti funzionino normalmente, per spiegare i sintomi osservati.

Questo differisce dalla diagnosi basata sulla coerenza (CBD) nei seguenti modi:

- Nel CBD, solo il comportamento normale deve essere rappresentato, e le ipotesi sono assunzioni di comportamento normale. Nella diagnosi di abduzione, devono essere rappresentati sia il comportamento difettoso che il comportamento normale.
- Nella diagnosi di abduzione, le osservazioni devono essere spiegate. Nel CBD, invece, si aggiungono alla base di conoscenza e si dimostra $false$.

Per l'abduzione possono essere utilizzate sia la procedura bottom-up che top-down. L'algoritmo BU calcola le spiegazioni minimali per ogni atomo. Con l'algoritmo TD si trovano spiegazioni di un g generando conflitti e dismostrando g (anziché $false$).

[9]Il termine abduzione è stato coniato da Peirce (1839-1914) per differenziare questo tipo di ragionamento dalla deduzione, che implica la determinazione di ciò che segue logicamente da un insieme di assiomi, e induzione, che implica l'inferenza di relazioni generali a partire da casi particolari (esempi).

Capitolo 5

Ragionamento e Rappresentazione Relazionale

5.1 Struttura Relazionale

Questo capitolo considera il ragionamento in termini di individui e relazioni:

Gli *individui* sono entità, cose e oggetti nel mondo.

Le *relazioni* specificano ciò che è vero su questi individui. Questo è inteso per essere il più generale possibile e include proprietà, che sono vere o false di singoli individui, proposizioni, che sono vere o false indipendentemente da qualsiasi individuo, così come relazioni tra più individui.

Modellare in termini di individui e relazioni ha diversi vantaggi rispetto all'utilizzo di features:

- Spesso è la rappresentazione naturale.
- Un agente potrebbe dover modellare un dominio senza sapere chi e quanti sono gli individui, quindi senza conoscere quali siano le features. Quando interagisce con l'ambiente, l'agente può costruire le features man mano che capisce quali sono gli individui di un particolare ambiente.
- Un agente potrebbe effettuare ragionamenti genericamente validi senza curarsi su chi siano gli specifici individui. Ad esempio, potrebbe derivare che una proprietà è valida per tutti gli individui senza sapere chi siano.

5.2 Simboli e Semantica

L'idea di base dietro l'uso della logica è che, quando i progettisti della base di conoscenza hanno un mondo particolare che vogliono caratterizzare, possono selezionare quel mondo come interpretazione intesa, selezionare significati per i simboli rispetto a tale interpretazione, e scrivere, come clausole, ciò che è vero in quel mondo. Quando un sistema calcola una conseguenza logica di una base

di conoscenza, un utente che conosce i significati dei simboli può interpretare questa risposta rispetto all'interpretazione intesa. Questo capitolo espande il linguaggio delle clausole proposizionali definite per consentire il ragionamento sugli individui e sulle relazioni. Le proposizioni atomiche ora hanno struttura interna in termini di relazioni e individui.

La mappatura tra i simboli nella mente dell'esperto e gli individui e le relazioni denotate da questi simboli è chiamata *concettualizzazione*[1].

La *correttezza* di una base di conoscenza è definita dalla *semantica*, non da un particolare algoritmo di ragionamento per dimostrare le query.

5.3 Datalog: Linguaggio Relazionale a Regole

Questa sezione espande la sintassi per il linguaggio delle clausole definite, con sintassi del calcolo dei predicati e convenzioni del Prolog.

Definiamo *parola* come la sequenza di lettere, numeri, underscore. La *sintassi* del Datalog è data dalle seguenti regole:

- Una *variabile* è una parola che inizia con maiuscola o underscore.
- Una *costante* è una parola che inizia con una minuscola o numero o stringa tra apici
- Un *predicato* è una parola che inizia con una minuscola. Le costanti e i predicati sono distinguibili in base al contesto.
- Un *termine* è una variabile o una costante.
- Un simbolo atomico (o *atomo*), è nella forma p o $p(t_, \ldots, t_n)$, dove p è un simbolo di predicato e ogni t_i è un termine. Ogni t_i è detto argomento del predicato.

Le nozioni di clausola definita, di regola, di query e di base di conoscenza sono le stesse di per le clausole definite proposizionali, ma con la definizione espansa di atomo.

Una *espressione* è un termine, o un atomo, o una clausola definita o una query. Una espressione è *ground* se non contiene variabili. Ad esempio, l'atomo insegna(nicola_fanizzi, 63507) è ground, invece insegna(Prof, Cod_Ins) non è ground.

5.3.1 Semantica del Datalog Ground

Il primo passo nel dare la semantica di Datalog è quello di dare la semantica per le espressioni ground (variabile-free).

Una *interpretazione* è una tripla $I =< D, \phi, \pi >$, dove

[1] In questo capitolo, assumiamo che la concettualizzazione sia nella testa dell'utente, o scritta informalmente, sotto forma di commenti. Rendere esplicite le concettualizzazioni è il ruolo di un'ontologia formale.

- D è un insieme non vuoto chiamato *dominio*. Gli elementi di D sono *individui*.
- ϕ è una funzione che assegna ad ogni costante c un elemento di D. Si dice che la costante c denota l'individuo $\phi(c)$. Quindi c è un simbolo, $\phi(c)$ può essere una qualsiasi cosa del mondo reale.
- π è una funzione che assegna ad ogni simbolo di predicato n-ario p una funzione che va da D^n in $\{true, false\}$. Quindi $\pi(p)$ specifica se una relazione denotata da un simbolo di predicato n-ario p è vero o falso per ogni n-upla di individui. Se il simbolo di predicato p non ha argomenti, allora $\pi(p)$ può essere o vero o falso e la semantica si riduce alla semantica per le clausole proposizionali definite.

Ogni termine ground denota un individuo in una interpretazione. Una costante c denota nell'interpretazione I l'individuo $\phi(c)$.

Un atomo ground può essere o vero o falso in una interpretazione. Un atomo $p(t_1, \ldots, t_n)$ è vero in I se $\pi(p)(< t'_1, \ldots, t'_n >) = true$, dove t'_i è l'individuo denotato dal termine t_i, ed è falso in I altrimenti.

Connettivi logici, modelli e conseguenza logica hanno la stessa semantica del calcolo proposizionale.

- Una clausola ground è falsa in una interpretazione se la testa è falsa e il corpo è vero (o vuoto), altrimenti, la clausola è vera nell'interpretazione
- Un *modello* di una base di conoscenza KB è una interpretazione in cui tutte le clausole della KB sono vere.
- Se una KB è una base di conoscenza e g è una proposizione, g è una conseguenza logica di KB, cioè $KB \vDash g$, se g è vera in ogni modello della KB.

5.3.2 Interpretare le Variabili

Quando una variabile appare in una clausola, la variabile è da intendersi universalmente quantificata nello *scope* della clausola. In altre parole, la clausola è vera in un'interpretazione solo se la clausola è vera per tutti i possibili valori di quella variabile.

Per definire formalmente la semantica delle variabili, un *assegnazione di variabile*, ρ, è una funzione che va dall'insieme delle variabili nel dominio D. Quindi assegna un elemento del dominio ad ogni variabile. Data una interpretazione $< D, \phi, \pi >$ e la funzione id assegnazione ρ, ogni termine denota un individuo nel dominio. Se il termine è una costante, l'individuo è dato da ϕ, se il termine è una variabile, l'individuo è dato da ρ. Pertanto, una clausola con variabili è vera in I se e solo se è vera $\forall\rho$.

La conseguenza logica è definita come visto per le clausole proposizionali definite: il corpo ground g (congiunzione di atomi) è conseguenza logica di KB, denotato con $KB \vDash g$, se e solo se g è vero in ogni modello di KB.

Il linguaggio delle clausole definite rende implicito il quantificatore universale ($\forall$). A volte è utile renderlo esplicito. La clausola $P(X) \leftarrow Q(X,Y)$ significa $\forall X \forall Y (P(X) \leftarrow Q(X,Y))$ che è equivalente a $\forall X (P(X) \leftarrow \exists Y Q(X,Y))$.

Quindi, le variabili libere che appaiono solo nel corpo sono quantificate esistenzialmente nello scope del corpo.

Semantica: Punto di Vista del Progettista

Estendendo al Datalog la metodologia per KB proposizionali:

1. Selezionare il dominio del task o mondo da rappresentare. In questo mondo, sia D l'insieme degli individui o delle cose alle quali ci si vuole riferire o sulle quali si vuole ragionare.

2. Associare le costanti del linguaggio a individui del mondo da nominare. Ad ogni elemento di D si assegna una costante per riferirvisi.

3. Per ogni relazione da rappresentare, si associa un simbolo di predicato. Ogni simbolo di predicato n-ario denota una funzione da D^n in $\{true, false\}$, la quale specifica il sottoinsieme di D^n per il quale la relazione è vera.

   Le relazioni possono avere qualunque *arietà*. L'associazione dei simboli con i rispettivi significati formano una interpretazione intesa.

4. Definire le clausole che sono vere nell'interpretazione intesa. Questo è spesso chiamato *assiomatizzazione del dominio*. Ad esempio, se la persona denotata da kim insegna il corso denotato da c322, la clausola teaches(kim,c322) sarà vera nell'interpretazione intesa.

5. Formulare le query riguardanti l'interpretazione intesa. Il sistema fornisce risposte da interpretare utilizzando la semantica attribuita ai simboli.

5.3.3 Query con Variabili

Le query sono utilizzate per chiedere se un enunciato sia una conseguenza logica di una KB. Con le query proposizionali, un utente può ricevere in riposta *si* o *no*. Le query con variabili per mettono all'utente di chiedere *quali individui* avverano la query.

L'*istanza* di una query è ottenuta sostituendo i termini alle variabili in essa presenti. Ogni occorrenza di una determinata variabile in una query deve essere sostituita con uno stesso termine. Data una query con variabili libere, una *risposta* è o una istanza della query che è conseguenza logica della KB, oppure "no", cioè non esiste un'istanza per la query che segua logicamente dalla KB (non significa che la query è falsa nell'interpretazione intesa!). Determinare quali istanze per una query seguano logicamente dalla KB prende il nome di *answer extraction*.

5.4 Sostituzioni e Dimostrazioni

Sia la procedura dimostrativa bottom-up che quella top-down possono essere estese al Datalog. Una procedura dimostrativa estesa alle variabili deve tener conto del fatto che una variabile libera in una clausola implica che tutte le istanze della clausola dovranno essere vere. Una dimostrazione potrebbe basarsi du istanze diverse di una stessa clausola.

5.4.1 Istanze e Sostituzioni

Un'*istanza* di una clausola è ottenuta sostituendo uniformemente i termini alle variabili nella clausola. Tutte le occorrenze di una variabile sono sostituite dallo stesso termine.

Una *sostituzione* specifica per ciascuna variabile il termine da sostituire:

$$\{V_1/t_1, \ldots, V_n/t_n\}$$

dove ogni V_i è una diversa variabile ed ogni t_i è un termine. L'elemento V_i/t_i è detto *binding* per V_i. Una sostituzione è in *forma normale* se nessun V_i appare in alcun t_j, con $i \neq j$.

Esempio 7. *Per esempio,* $\{X/Y, Z/a\}$ *è una sostituzione in forma normale. La sostituzione* $\{X/Y, Z/X\}$ *non è in forma normale in quanto* X *appare sia a sinistra che a destra di un binding.*

L'*applicazione* di una sostituzione $\sigma = \{\{V_1/t_1, \ldots, V_n/t_n\}\}$ all'espressione e, scritto come $e\sigma$, è un espressione identica a e, ad eccezione del fatto che ogni occorrenza di V_i in e è sostituita con il corrispondente termine t_i. L'espressione $e\sigma$ è chiamata *istanza* di e. Se $e\sigma$ non contiene alcuna variabile, è chiamata *istanza ground* di e.

Esempio 8. *Esempi di applicazione di sostituzioni:*

$p(a, X)\{X/c\} = p(a, c)$.
$p(X, X, Y, Y, Z)\{X/Z, Y/t\} = p(Z, Z, t, t, Z)$.

Per le clausole, data la sostituzione $\{X/Y, Z/a\}$ *si passa da*

$p(X, Y) \leftarrow q(a, Z, X, Y, Z)$

a

$p(Y, Y) \leftarrow q(a, a, Y, Y, a)$.

Una sostituzione σ è un *unificatore* delle espressioni e_1 ed e_2 se $e_1\sigma$ è identico a $e_2\sigma$. In altre parole, un unificatore di due espressione è una sostituzione che quando applicata ad entrambe le espressione il risultato è un'espressione identica.

Esempio 9. $\{X/a, Y/b\}$ *è un unificatore di* $t(a, Y, c)$ *e* $t(X, b, c)$ *perché* $t(a, Y, c)\{X/a, Y/b\} = t(X, b, c)\{X/a, Y/b\} = t(a, b, c)$.

Una sostituzione σ è detta *unificatore più generale* (MGU) per le espressioni e_1 ed e_2 se

- σ è un unificatore di e_1 ed e_2
- per qualunque altro unificatore di e_1 ed e_2 σ' si ha che $e\sigma'$ è una istanza di $e\sigma$

L'espressione e_1 è una ridenominazione di e_2 se differiscono solo per il nome delle variabili. In questo caso, l'una è istanza dell'altra. Due espressioni unificabili avranno almeno un MGU.

5.4.2 Procedura Bottom-up con Variabili

La procedura di dimostrazione bottom-up può essere estesa al Datalog utilizzando le istanze ground delle clausole.Una istanza ground di una clausola è ottneuta sostituendo uniformemente le costanti alle variabili nella clausola. Le costanti richieste sono quelle presenti nella base di conoscenza o nella query. Qualora non ve ne fossero, vanno inventate.

Esempio 10. *Data la base di consocenza*

$q(a).$
$q(b).$
$r(a).$
$s(W) \leftarrow r(W).$
$p(X, Y) \leftarrow q(X) \wedge s(Y).$

L'insieme di tutte le istanze gound è

$q(a).$
$q(b).$
$r(a).$
$s(a) \leftarrow r(a).$
$s(b) \leftarrow r(b).$
$p(a, a) \leftarrow q(a) \wedge s(a).$
$p(a, b) \leftarrow q(a) \wedge s(b).$
$p(b, a) \leftarrow q(b) \wedge s(a).$
$p(b, b) \leftarrow q(b) \wedge s(b).$

Applicando la procedura proposizionale, dalle istanze ground si derivano le seguenti conseguenze logiche della KB: $q(a), q(b), r(a), s(a), p(a, a)$ *e* $p(b, a)$.

Esempio 11. *Sia data la base di conoscenza*

$p(X, Y).$
$g \leftarrow p(W, W).$

*La procedura bottom-up per la query "*ask g*" deve inventare un nuovo simbolo di costante, diciamo c. L'insieme di tutte le istanze ground è quindi*

$p(c, c)$.
$g \leftarrow p(c, c)$.

La procedura bottom-up deriva $\{p(c, c), g\}$*, rispondendo "yes".*

La procedura BU applicata al grounding della KB è *corretta* (sound), perché ogni istanza di ogni regola è vera in ogni modello. La procedura è la stessa del caso delle variabili libere, ma utilizza l'insieme delle istanze ground delle clausole, ognuna delle quali è vera perchè le variabili in una clausola sono quantificate universalmente.

La procedura è anche *completa* per atomi ground. Cioè se un atomo ground è una conseguenza della KB, allora si può derivare. Per dimostrarlo si costruisce un particolare modello generico, chiamato *interpretazione di Herbrand* dove il dominio è simbolico e consiste di tutte le constanti del linguaggio. Poiché ogni costante denota se stessa, ϕ è fissata e tutto ciò che resta da fare è definire π per i predicati. Questo modello per una data KB è *minimale* in quanto contiene il minor numero di atomi veri rispetto ad ogni altro modello.

5.4.3 Risoluzione Definita con Variabili

La procedura di prova top-down può essere estesa per gestire le variabili consentendo l'uso di istanze di regole nella derivazione.

Una *clausola di risposta generalizzata* è della forma

$$yes(t_1, \ldots, t_k) \leftarrow a_1 \wedge a_2 \wedge \cdots \wedge a_m$$

dove $t_1, \ldots, t_k$ sono termini e $a_1, \ldots, a_m$ sono atomi. L'uso di *yes* consente l'estrazione della risposta: determinare quali istanze della query (con variabili) sono conseguenze logiche della base di conoscenza.

Inizialmente, la clausola di risposta generalizzata per la query q è

$$yes(V_1, \ldots, V_k) \leftarrow q$$

dove $V_1, \ldots, V_k$ sono le variabili che compaiono in q. Ciò significa che una istanza di $yes(V_1, \ldots, V_k)$ è vera se la corrispondente istanza di q è vera.

In ogni fase, l'algoritmo seleziona un atomo nel corpo della clausola di risposta corrente e sceglie una clausola di KB la cui testa si unifichi con tale atomo.

La *risoluzione SLD* della clausola di risposta generalizzata

$$yes(t_1, \ldots, t_k) \leftarrow a_1 \wedge a_2 \wedge \cdots \wedge a_m$$

avendo selezionato l'atomo a_1, con la clausola scelta nella KB

$$a \leftarrow b_1 \wedge \cdots \wedge b_p$$

dove a_1 e a hanno l'MGU σ, è la clausola di risposta

$$(yes(t_1, \ldots, t_k) \leftarrow b_1 \wedge \cdots \wedge b_p \wedge a_2 \wedge \cdots \wedge a_m)\sigma$$

dove il corpo della clausola scelta ha sostituito a_1 nella clausola di risposta e l'MGU σ è applicato all'intera clausola di risposta.

una *derivazione SLD* è una sequenza di clausole di risposta $\gamma_0, \gamma_1, \ldots, \gamma_n$ tale che

- γ_0 è la clausola di risposta corrispondente alla query originaria q.
- γ_i è ottenuto per risoluzione SLD, selezionando un atomo a_1 nel corpo di γ_{i-1}, scegliendo una clausola $a \leftarrow b_1 \wedge \cdots \wedge b_p$ dalla base di conoscenza, la cui testa a si unifichi con a_i, sostituendo a_1 con il corpo $b_1 \wedge \cdots \wedge b_p$, e applicando l'unificatore all'intera clausola di risposta risultante.
- γ_n è una risposta (se la dimostrazione è andata a buon fine). Ovvero, è nella forma

  $yes(t_1, \ldots, t_k) \leftarrow .$

  Quando succede, l'algoritmo restituisce la risposta alla query

  $V_1 = t_1, \ldots, V_k = t_k.$

Valgono le stesse considerazioni sul non determinismo effettuate per la procedura TD per le clausole definite.

5.5 Simboli di Funzione

Datalog richiede un nome (simbolo di costante) per ogni individuo sul quale si deve ragionare. Spesso, però, è più semplice identificare un individuo in termini di altri individui (o di componenti), piuttosto che utilizzare costanti separate per ogni individuo.

Usare una costante per ogni individuo significa che la KB può rappresentare solo un numero finito di individui, fissato al momento della costruzione della base. Tuttavia, si potrebbe voler ragionare su un insieme di individui potenzialmente infiniti.

Un simbolo di funzione ci permette di descrivere un individuo in termini di altri individui. Sintatticamente, un *simbolo di funzione* è una parola che inizia con una lettera minuscola. Pertanto, estendiamo la definizione di termine, affinché un termine possa essere una variabile, una costante o della forma $f(t_1, \ldots, t_n)$ con f simbolo di funzione (n-aria) ed ogni t_i è un termine.

A questo punto la semantica del Datalog può essere estesa per includere i simboli di funzione. In ogni interpretazione $< D, \phi, \pi >$, ϕ assegna, ad oni simbolo di funzione n-aria, una funzione $D^n \rightarrow D$. Una costante può essere vista come un simbolo di funzione con arietà zero.

Esempio 12. *Supponiamo si vogliano definire le date. Allora si devono avere a disposizione:*

- *Costanti: date dall'unione dell'insieme dei numeri interi e l'insieme*

  {jan, feb, mar, apr, may, jun, jul aug, sep, oct, nov, dec}

- *Funzioni: la funzione* ce *(common era) definita come*

  $ce(T, M, D)$,

  con anno Y*, mese* M *e giorno* D

- *Predicati: ovvero clausole quantificate sugli argomenti delle funzioni. Ad esempio si possono definire le relazioni before e month.*
  - $before(D_1, D_2)$ *è vera se la data* D_1 *è precedente alla data* D_2*.*
  - $month(M, N)$ *è vera se il mese* M *è l'*N*-esimo mese dell'anno, con* N *numero intero.*

Quindi assiomatizziamo la relazione "before" per le date della common era:

$before(ce(Y_1, M_1, D_1), ce(Y_2, M_2, D_2)) \leftarrow Y_1 < Y_2.$
$before(ce(Y, M_1, D_1), ce(Y, M_2, D_2)) \leftarrow$
$month(M_1, N_1) \wedge month(M_2, N_2) \wedge N_1 < N_2.$
$before(ce(Y, M, D_1), ce(Y, M, D_2)) \leftarrow D_1 < D_2.$
$month(jan, 1).$
$month(feb, 2).$
$month(mar, 3).$
$month(apr, 4).$
$\ldots$

Programmazione Logica Qualunque funzione computabile può essere calcolata usando una KB di clausole con simboli di funzione interpretabile come un programma logico. Il linguaggio dei programmi logici è Turing completo. I simboli di funzione possono definire *strutture dati* come alberi e liste.

Esempio 13. *Un albero è una struttura di dati molto utile. La si può costruire con due funzioni e alcune relazioni.*

- *Funzioni:*
  - $node(N, LT, RT)$ *denota un nodo interno di un albero con il nome* N *e due sotto-alberi sinistro* LT *e destro* RT
  - $leaf(L)$ *indica un nodo foglia con etichetta* L*.*
- *Relazioni:*
  - $at\_leaf(L, T)$ *vera se* L *è l'etichetta di una foglia dell'albero* T*:*
    - $at\_leaf(L, leaf(L))$.
    - $at\_leaf(L, node(N, LT, RT)) \leftarrow at\_leaf(L, LT)$

    - $at\_leaf(L, node(N, LT, RT)) \leftarrow at\_leaf(L, RT)$

- $in\_tree(L, T)$ *vera se* L *è l'etichetta di un nodo interno dell'albero* T*:*
    - $in\_tree(L, node(L, RT, LT))$.
    - $in\_tree(L, node(N, LT, RT)) \leftarrow in_t ree(L, LT)$
    - $in\_tree(L, node(N, LT, RT)) \leftarrow in\_tree(L, RT)$

Esempio 14. *Una lista è una sequenza ordinata di elementi. Una lista è o vuota, o un elemento seguito da una lista. Stabiliamo che la costante nil denoti la lista vuota. Si può scegliere un simbolo di funzione, diciamo* $cons(Hd, Tl)$*, con l'interpretazione intesa per la quale esso denota una lista che ha per primo elemento* Hd *ed il resto della lista* Tl

Ad esempio, la lista contente a, b, c può essere rappresentata come $cons(a, cons(b, cons(c, nil)))$.

Questa struttura può essere utilizzata attraverso predicati come $append(X, Y, Z)$*, vero quando* X, Y *e* Z *sono liste, tali che* Z *è composta dagli elementi di* X *seguiti da quelli di* Y*. Predicato assiomatizzato come segue:*

- $append(nil, L, L)$.
- $append(cons(Hd, X), Y, cons(Hd, Z)) \leftarrow (X, Y, Z)$

5.5.1 Procedure di Dimostrazione con Simboli di Funzione

L'uso dei simboli di funzione coinvolge infiniti termini. Ciò significa che, nel forward chaining delle clausole, dobbiamo garantire che il criterio di selezione delle clausole sia equo *fairness*. Il problema di ignorare alcune clausole per sempre è conosciuto come *starvation*. Un criterio di selezione è equo se garantisce che qualsiasi clausola adatta ad essere selezionata venga prima o poi selezionata.

La procedura di dimostrazione bottom-up può generare una sequenza infinita di conseguenze e se la selezione è *fair*, ogni conseguenza potrà essere generata e quindi la procedura di dimostrazione è completa.

La procedura di dimostrazione top-down è la stessa utilizzata per Datalog. L'unica modifica nell'algoritmo è l'aggiunta dell'*occurs-check*, o controllo di occorrenza. Questa modifica consiste nel non unificare una variabile X con i termini t in cui già occorre (se $t \neq X$). Altrimenti la procedura perderebbe la soundness (non sarebbe corretta).

Zucchero Sintattico Per il resto del capitolo applicheremo lo "zucchero sintattico" della notazione in Prolog. La lista vuota sarà indicata con []. $[E|R]$ indica la lista primo elemento E ed il resto della lista R. $[X|[Y]]$ si può scrivere come $[X, Y]$, con Y sequenza di valori. Ad esempio, $[a|[]] \rightarrow [a]$, $[b|[a|[]]] \rightarrow [b, a]$ e $[a|[b|C]] \rightarrow [a, b|C]$.

5.6 Uguaglianza

A volte è utile utilizzare più di un termine per riferirsi ad uno *stesso* individuo. Altre volte serve che ogni nome si riferisca ad un individuo diverso. Spesso non si sa se due nomi denotino lo stesso individuo.

La relazione di uguaglianza ci permette di rappresentare se due termini denotano o meno lo stesso individuo nel mondo[2].

L'uguaglianza è un simbolo predicativo speciale con un'interpretazione intesa standard indipendente dal dominio.

I termini t_1 e t_2 sono *uguali* in una interpretazione I se t_1 e t_2 denotano lo stesso individuo in I[3].

5.6.1 Permettere Asserzioni di Uguaglianza

Se non si consente l'uguaglianza per la testa delle clausole, l'unica cosa che è uguale a un termine in tutte le interpretazioni è se stesso.

Per affermare o dedurre che due termini denotano lo stesso individuo, il sistema di rappresentazione e di ragionamento deve essere in grado di derivare ciò che segue da una base di conoscenza che include clausole con uguaglianza nella testa delle clausole. Ci sono due modi per farlo. Il primo è assiomatizzare l'uguaglianza come qualsiasi altro predicato. L'altro è quello di definire procedure speciali d'inferenza per l'uguaglianza.

Assiomatizzare l'uguaglianza

L'uguaglianza può essere assiomatizzata come segue. I primi tre assiomi affermano che l'uguaglianza è riflessiva, simmetrica e transitiva:

$$X = X$$
$$X = Y \leftarrow Y = X$$
$$X = Z \leftarrow X = Y \wedge Y = Z$$

Gli altri assiomi dipendono dall'insieme di simboli di funzione e di relazione nel linguaggio; per i quali si formano *schemi di assiomi* da istanziare.

Gli schemi di assiomi specificano, sia per funzioni n-arie che per predicati n-ari, che sostituire un termine con un altro termine uguale non cambia il valore della funzione o del predicato.

Nello schema di assiomi c'è una regola della forma

$$f(X_1, \ldots, X_n) = f(Y_1, \ldots, Y_n) \leftarrow X_1 = Y_1 \wedge \cdots \wedge X_n = Y_n$$

per ogni funzione n-aria f, ed una regola della forma

[2] Si noti che, nel linguaggio delle clausole definite tutte le risposte erano valide indipendentemente dal fatto che i termini denotassero o meno gli stessi individui.

[3] Attenzione, il predicato di uguaglianza non denoto similarità tra due oggetti, ma identicità. Indica che ci sono due nomi per lo stesso individuo, non due individui del dominio con caratteristiche simili.

$$p(X_1, \ldots, X_n) \leftarrow p(Y_1, \ldots, Y_n) \wedge X_1 = Y_1 \wedge \cdots \wedge X_n = Y_n$$

per ogni predicato n-ario p.

Avere questi assiomi espliciti come parte della base di conoscenza risulta essere molto inefficiente. Inoltre, non è garantito che l'applicazione di queste regole si fermi se viene utilizzato un interprete top-down depth-first. Per esempio, l'assioma simmetrico potrebbe causare un ciclo infinito a meno di controlli sulla ripetizione di sotto-goal.

Procedure Speciali di Ragionamento con l'Uguaglianza

La *paramodulazione* è un modo per implementare l'uguaglianza estendendo la procedura di dimostrazione.

L'idea generale è che, se $t_1 = t_2$, qualsiasi occorrenza di t_1 può essere sostituita da t_2. L'uguaglianza può quindi essere trattata come una *regola di riscrittura*, sostituendo eguali con eguali. Questo approccio funziona meglio se si seleziona una rappresentazione canonica per ogni individuo, ovvero un termine convenzionale sul quale possono essere mappate altre rappresentazioni per quell'individuo.

5.6.2 Unique Names Assumption

L'*assunzione di nome unico (UNA)* costituisce un'alternativa all'assiomatizzazione dell'uguaglianza.

L'idea di base è che invece di aspettarsi che l'utente assiomatizzi quali nomi denotano lo stesso individuo e quali denotano individui diversi, è spesso più facile avere la convenzione che termini di ground diversi denotano individui diversi.

Sotto l'assunzione di nome unico, termini ground distinti denotano individui differenti. Tale assunzione non deriva dalla semantica del linguaggio delle clausole definite.

L'assunzione di nome unico può essere assiomatizzata con il seguente schema di assiomi per la disuguaglianza, da aggiungere a quello per l'uguaglianza visto precedentemente:

$c \neq c'$ per ogni costante distinta c e c'
$f(X_1, \ldots, X_n) \neq g(Y_1, \ldots, Y_m)$ per ogni diverso simbolo di funzione f e g
$f(X_1, \ldots, X_n) \neq f(Y_1, \ldots, Y_n) \leftarrow X_i \neq Y_i$, per ogni simbolo di funzione f. Di questo schema ci sono n istanze per ogni simbolo di funzione n-aria f (uno per ogni i tale che $1 \leq i \leq n$)
$f(X_1, \ldots, X_n) \neq c$ per ogni simbolo di funzione f e costante c

Con questa assiomatizzazione, i termini ground sono identici se e solo se si unificano. Questo non vale per i termini non ground. Ad esempio $a \neq X$ ha istanze vere nelle quali X ha un certo valore b e un'istanza falsa in cui X ha il valore a.

L'assunzione di unicità dei nomi è utile per non dover specificare distinzioni come $kim \neq sam$, $kim \neq chris$, $chris \neq sam$, ..., però può essere inopportuna in alcuni casi, ad esempio per $2 + 2 \neq 4$

Procedura Top-Down con UNA

La procedura di dimostrazione top-down che incorpora l'assunzione di unicità dei nomi non dovrebbe trattare la disuguaglianza come uno degli altri predicati, principalmente perché esistono troppi individui diversi di un dato individuo.

Se c'è un sub-goal $t_1 \neq t_2$, per i termini t_1 e t_2 ci sono tre possibili casi:

1. t_1 e t_2 non si unificano, quindi in questo caso la dimostrazione di $t_1 \neq t_2$ ha successo.
2. t_1 e t_2 sono identici, incluso l'avere le stesse variabili nelle stesse posizioni, in questo caso la dimostrazione di $t_1 \neq t_2$ fallisce.
3. Altrimenti ci sono alcune istanze di $t_1 \neq t_2$ la cui dimostrazione ha successo (istanze ground non compatibili con l'MGU) ed altre che falliscono (istanze compatibili con l'MGU).

   Diversamente dagli altri sub-goal, in questo caso bisogna evitare di enumerare tutte le istanze di successo in quanto sarebbero troppe.

La procedura di dimostrazione top-down può essere estesa per incorporare l'UNA. Le disuguaglianze del primo tipo avranno successo e quelle del secondo tipo falliranno. Le disuguaglianze del terzo tipo possono essere *posticipati*, in attesa che i goal successivi unifichino le variabili in modo che si verifichi uno dei primi due casi.

Per ritardare un obiettivo, durante la dimostrazione, quando si seleziona un atomo nel corpo, l'algoritmo dovrebbe selezionare prima uno degli atomi che non è stato posticipato. Se non ci sono altri atomi da selezionare, e nessuno dei primi due casi è applicabile, la query dovrebbe avere successo.

Un'istanza in cui la disuguaglianza ha successo c'è sempre. Vale a dire, l'istanza che assegna ad ogni variabile una costante diversa non già usata. Quando questo accade, l'utente deve fare attenzione quando interpreta le variabili libere nella risposta. La risposta non è vera per ogni istanza delle variabili libere, ma è vera solo per qualche esempio.

5.7 Assunzione di Conoscenza Completa

L'assunzione di conoscenza completa (o di mondo chiuso), vista in precedenza, è l'assunzione che qualsiasi affermazione che non deriva da una base di conoscenza è falsa. Permette anche dimostrazioni per negation as failure.

Per estendere l'assunzione della conoscenza completa a programmi logici con variabili e simboli di funzioni, abbiamo bisogno di assiomi per l'uguaglianza, la chiusura del dominio e una nozione più sofisticata del completamento. Ancora una volta, questo definisce una forma di *negation as failure*.

Esempio 15. *Supponiamo che la relazione* student *sia definita da*

```
student(mary).
student(john).
student(ying).
```

Per l'assunzione di conoscenza completa sarebbero gli unici studenti:

$\mathsf{studenti(X)} \leftrightarrow X = \mathsf{mary} \vee X = \mathsf{john} \vee X = \mathsf{ying}.$

In altre parole, se X è mary, john o ying, allora è uno studente, e se X è uno studente allora deve essere uno di questi tre. In particolare kim *non è uno studente.*

Per concludere che $\neg\mathsf{student(kim)}$ *è necessario dimostrare che* $\mathsf{kim} \neq \mathsf{mary} \wedge \mathsf{kim} \neq \mathsf{john} \wedge \mathsf{kim} \neq \mathsf{ying}$. *Per derivare la disuguaglianza, è richiesta l'assunzione di unicità del nome.*

L'assunzione di conoscenza completa include l'assunzione di unicità del nome, pertanto si devono includere gli schemi di assiomi per l'uguaglianza e per la disuguaglianza.

La forma normale di Clark della clausola

$p(t_1, \ldots, t_k) \leftarrow B.$

è la clausola

$p(V_1, \ldots, V_k) \leftarrow \exists W_1 \ldots \exists W_m V_1 = t_1 \wedge \cdots \wedge V_k = t_k \wedge B.$

dove $V_1, \ldots, V_k$ sono k variabili che non appaiono nella clausola originale e $W_1 \ldots W_m$ sono le variabili originarie della clausola.

Supponiamo che tutte le clausole per p siano in forma normale di Clark con lo stesso insieme di nuove variabili, ottenendo

$p(V_1, \ldots, V_k) \leftarrow B_1.$
$\ldots p(V_1, \ldots, V_k) \leftarrow B_n.$

esse sono equivalenti a

$p(V_1, \ldots, V_k) \leftarrow B_1 \vee \cdots \vee B_n.$

Il completamento di Clark del predicato p è l'equivalenza

$\forall V_1 \ldots \forall V_k p(V_1 \ldots V_k) \leftrightarrow B_1 \wedge \cdots \wedge B_n$

dove nei corpi la negazione per fallimento $\sim$ è sostituita dalla negazione logica standard $\neg$. Il completamento significa che $p(V_1, \ldots, V_k)$ è vero se e solo se almeno un corpo B_i è vero.

Il completamento di Clark di una KB comprende i complementi di ogni predicato e degli assiomi di uguaglianza e disuguaglianza.

Esempio 16. *Per le clausole*

student(mary).
student(john).
student(ying).

la forma normale di Clark è

student(V)$\leftarrow$*V=mary.*
student(V)$\leftarrow$*V=john.*
student(V)$\leftarrow$*V=ying.*

che equivale a

student$(V) \leftarrow V = mary \vee V = john \vee V = ying.$

Il completamento del predicato *student* *è*

$\forall V$ *student(V)* $\leftrightarrow V = mary \vee V = john \vee V = ying.$

Capitolo 6

Sistemi Basati su Conoscenza e Ontologie

6.1 Implementare Sistemi Basati su Conoscenza

Spesso è utile per un agente essere in grado di rappresentare e ragionare sul proprio ragionamento, ovvero essere in grado di *riflettere.* Il ragionamento esplicito sulla propria rappresentazione e sul proprio ragionamento permette ad un sistema basato su conoscenza (o agente) di adattarsi a particolari esigenze.

Dal punto di vista pratico, implementare la riflessione significa costruire nuovi linguaggi con caratteristiche richieste da particolari applicazioni.

Un *meta-interprete* per un linguaggio è un interprete scritto nello stesso linguaggio interpretato. Un tale interprete è utile nella prototipazione di nuovi linguaggi con nuove caratteristiche utili.

Quando si implementa un linguaggio in un altro, il linguaggio che viene implementato è chiamato *linguaggio base* e il linguaggio in cui si implementa è chiamato *metalinguaggio*. Le espressioni nel linguaggio base sono dette di *livello base*, le espressioni nel meta-linguaggio sono dette di *meta-livello*.

6.1.1 Linguaggi di Base e Meta-Linguaggi

Quando si scrive un meta-interprete di programmazione logica, c'è una scelta da effettuare su come rappresentare le variabili. Nella *rappresentazione non ground*, un termine di livello base è rappresentato come lo stesso termine nel meta-linguaggio, quindi in particolare, le variabili di livello base sono rappresentate come variabili di meta-livello. Questo significa che l'unificazione del meta-livello può essere utilizzata per unificare i termini del livello base. Al contrario, nella *rappresentazione ground* le variabili del linguaggio di base sono rappresentate come costanti nel metalinguaggio.

Un meta-linguaggio deve essere capace di rappresentare tutti i costrutti del linguaggio base.

- Le variabili, costanti e simboli di funzione del livello-base sono rappresentate come la rispettive variabili, costanti e simboli di funzione del meta-livello.
- Un simbolo di predicato p a livello base viene rappresentato come simbolo di funzione p nel meta-livello.
- Un atomo di livello base $p(t_1, \ldots, t_k)$ viene rappresentato come un termine $p(t_1, \ldots, t_k)$ nel meta-livello[1].
- I corpi (congiunzioni di atomi) a livello base vengono rappresentati come termini nel meta-livello.

  Se e_1 ed e_2 sono termini di meta-livello che denotato atomi o corpi di livello base, allora il termine di meta-livello $oand(e_1, e_2)$ denota la loro congiunzione.
- Le clausole definite di livello base sono rappresentate come atomi (o fatti) nel meta-livello. La regola di livello base $h \leftarrow b$ diventa un atomo di meta-livello $clause(h, b')$, dove b' è la rappresentazione del corpo di b. Un fatto a di livello base è rappresentato come un atomo di meta-livello $cluse(a, true)$, dove $true$ è una costante di meta-livello che denota il corpo vuoto a livello-base.

Per migliorare la leggibilità del livello base, possiamo usare una notazione infissa per la quale

- $oand(a_1, e_2)$ si indica con $e_1 \& e_2$, il simbolo di funzione & nel meta-linguaggio denota l'operatore infisso $\wedge$, tra atomi nel linguaggio-base.
- $clause(h, b)$ si indica con $h \Leftarrow b$, l'operatore $\leftarrow$ viene rappresentato a meta-livello con il simbolo di predicato $\Leftarrow$

Quindi, una clausola di livello base del tipo $h \leftarrow a_1 \wedge \cdots \wedge a_n$ è rappresentata come un atomo a meta-livello del tipo

$h \Leftarrow a_1 \& \ldots \& a_n$.

6.1.2 Meta-Interprete Vanilla

Un meta-interprete basilare (vanilla) per il linguaggio delle clausole definite (scritto nel medesimo linguaggio) consiste nell'assiomatizzazione della relazione prove. Questo meta-interprete copre ogni caso possibile nel corpo di una clausola

[1] Ricordiamo che un atomo è una proposizione elementare, ovvero un'affermazione che non può essere ulteriormente scomposta in proposizioni più semplici. Ad esempio, "la palla è rotonda" è un atomo.

Un termine è un'entità che può essere utilizzata per descrivere un oggetto o un concetto all'interno di un sistema logico. I termini possono essere combinati per formare proposizioni più complesse. Ad esempio, "la palla" e "è rotonda" sono entrambi termini che possono essere combinati per formare l'atomo "la palla è rotonda".

Syntactic construct		Meta-level representation of the syntactic construct	
variable	X	variable	X
constant	c	constant	c
function symbol	f	function symbol	f
predicate symbol	p	function symbol	p
"and" operator	$\wedge$	function symbol	$\&$
"if" operator	$\leftarrow$	predicate symbol	$\Leftarrow$
clause	$h \leftarrow a_1 \wedge \cdots \wedge a_n.$	atom	$h \Leftarrow a_1 \,\&\, \cdots \,\&\, a_n$
clause	$h.$	atom	$h \Leftarrow true$

Figura 6.1: Rappresentazione non ground per il linguaggio di base.

o in una query, e specifica come risolvere ogni caso. Ogni corpo, dato come argomento di prove, può essere o vuoto, o una congiunzione o un atomo. Se il corpo di livello base è vuoto, la dimostrazione è immediata. Per dimostrare una cosngiunzione di livello base $A\&b$, sia dimostra sia A che B. Per dimostrare un atomo H, di deve trova una clausola di livello base con testa H e dimostrarne il corpo.

Esempio 17. *Meta-Interprete Vanilla per le clausole definite.*

- $prove(true)$.
- $prove((A\&B)) \leftarrow prove(A) \wedge prove(B)$.
- $prove(H) \leftarrow (H \Leftarrow B) \wedge prove(B)$.

6.1.3 Estensioni del Linguaggio-Base

Il linguaggio-base può essere modificato modificandone il meta-interprete. L'insieme delle conseguenze dimostrabili può essere ampliato aggiungendo clausole al meta-interprete o ridotto aggiungendo condizioni alle clausole del meta-interprete.

Un esempio di estensione del linguaggio base consiste nel permettere la disgiunzione ($\vee$) nel corpo di una clausola. Questo non richiede che sia definita la disgiunzione anche nel meta-linguaggio.

Nella pratica, però, on tutti i predicati devono essere assiomatizzati (definiti da clausole). Ad esempio, non ha senso senso assiomatizzare l'aritmetica in un calcolatore. In effetti, invece di assiomatizzare questi predicati, è meglio utilizzare chiamate dirette al sistema sottostante. Assumiamo che $call(g)$ valuti G direttamente.

I predicati predefiniti (o built-in) per procedure di livello-base possono essere valutati attraverso relazioni di meta-livello del tipo $built_in(X)$ che è $true$ se tutte le istanze di X si possono valutare direttamente.

Esempio 18. *Esempio di un Meta-Interprete che usa chiamate built-in e disgiunzione.*

- $prove(true)$.
- $prove((A \& B)) \leftarrow prove(A) \wedge prove(B)$.
- $prove((A \vee B)) \leftarrow prove(A)$.
- $prove((A \vee B)) \leftarrow prove(B)$.
- $prove(H) \leftarrow built\_in(H) \wedge call(H)$.
- $prove(H) \leftarrow (H \Leftarrow B) \wedge prove(B)$.

6.1.4 Ragionamento a Profondità Limitata

Per specializzare il ragionamento si possono aggiungere condizioni a clausole di meta-livello che limitino quello che può essere dimostrato.

Un meta-interprete utile implementa depth-bounded search. Può essere usato per cercare dimostrazioni brevi o come parte di un iterative-deepening searcher che effettua ripetute ricerche DFS, aumentando il limite di profondità in ogni fase.

Questo può essere fatto assiomatizzando la relazione $bprove(G, D)$, che è verificata se G può essere dimostrato con un albero profondo al più D, con $D \in \mathbb{N}$.

Esempio 19. *Esempio di Meta-Interprete per depth-bounded search.*

- $bprove(true, D)$.
- $bprove((A \& B), D) \leftarrow bprove(A, D) \wedge bprove(B, D)$.
- $bprove(H, D) \leftarrow D \geq 0 \wedge D_1 \text{ is } D - 1 \wedge (H \Leftarrow B) \wedge bprove(B, D_1)$.

Osserviamo che:

- *Viene utilizzato il predicato infisso* is *che è vero se il valore numerico di V è E. In E il simbolo "$-$" indica la sottrazione come funzione infissa*
- *Se D è limitato da un numero nella query non tratterà dimostrazioni di lunghezza maggiore, pertanto l'interprete sarà incompleto.*
- *Ogni dimostrazione trovata dal meta-interprete* prove *può essere trovata da* bprove *se il valore di D è sufficientemente grande.*
- *brpove potrebbe trovare dimostrazioni che* prove *non riesce a trovare, in effetti* prove *può andare in loop prima di aver esplorato tutte le dimostrazioni.*

Un altro meta-interprete utile è quello che consente di implementare le domande *how* costruendo esplicitamente l'albero di dimostrazione per una risposta.

Un'abilità utile nei meta-interpreti è la capacità di *differire* (o posticipare) i goal[2]. Alcuni goal, invece di essere dimostrati, possono essere inseriti in una lista. Al termine della dimostrazione, il sistema deriva l'implicazione che, se i goal differiti sono tutti veri allora la risposta calcolata è giusta.

[2]Ricordiamo la procedura di dimostrazione top-down con assunzione di unicità dei nomi.

6.2 Condivisione della Conoscenza

Ricordiamo che un'*ontologia* è una specifica formale del significato dei simboli in un sistema basato su conoscenza.

Il significato dei simboli può essere specificato informalmente, ad esempio può risiedere nella mente del progettista, in un manuale utente, o nei commenti aggiunti alla base di conoscenza. Specificare formalmente il significato dei simboli, però, è importante per l'*interoperabilità semantica*, ovvero, la capacità delle diverse basi di conoscenza di lavorare insieme a livello semantico in modo che i significati dei simboli siano rispettati.

La specifica non ha bisogno di definire i dettagli, ha solo bisogno di definire i termini abbastanza bene da garantirne un uso coerente[3].

Data una specifica del significato dei simboli, un agente può usare quel significato per l'acquisizione di ulteriore conoscenza, per la spiegazione e il debug a livello di conoscenza.

6.3 Rappresentazioni Flessibili

La logica (con variabili, termini e relazioni) può essere usata per costruire rappresentazioni flessibili che consentono l'aggiunta modulare della conoscenza, inclusa l'aggiunta di argomenti alle relazioni.

6.3.1 Scegliere Individui e Relazioni

Dato un linguaggio di rappresentazione logico e un mondo sul quale ragionare, i progettisti devono scegliere individui e relazioni da rappresentare. Decidere come decomporre il mondo d'interesse sta a chi lo modella, ma si sono delle linee guida utili per scegliere le relazioni e gli individui da rappresentare.

Discutiamone con un esempio molto esplicativo.

Esempio 20. *Supponiamo di decidere che "red" sia una categoria appropriata per classificare oggetti.*

Inizialmente si potrebbe trattare il nome red *come una relazione unaria e scrivere che l'oggetto* a *è rosso cone*

$red(a)$.

Se si modella il colore rosso in questo modo, si può facilmente chiedere tutto ciò che è rosso con la query

ask $red(X)$.

La X restituita sono gli oggetti rossi.

Ciò che non possiamo chiedere è "Di che colore è a*?". Infatti, nella sintassi delle clausole definite, non si può chiedere*

[3] Specificare che un termometro misura la temperatura in Celsius sarebbe sufficiente per molte applicazioni, senza dover definire quale sia la temperatura o la precisione del termometro.

ask $X(a)$.

in quanto, nella logica per primo ordine, i nomi di predicati non possono essere variabili. E comunque, in logiche di ordine superiore questa query restituirebbe tutte le proprietà di **a** *e non solo il suo colore.*

A questo punto si potrebbe pensare anche ai colori come individui ed usare la costante **red** *per denotare il colore rosso. Quindi si potrebbe usare il predicato* **color(Ind,Val)** *per indicare che l'individuo fisico* **Ind** *è di colore* **Val**. *"L'oggetto è rosso" si può ora scrivere come*

$color(a, red)$.

Con questa rappresentazione si può chiedere "Cosa è di colore rosso?" con la query

ask $color(X, red)$.

e "Di che colore è l'oggetto a"? con la query

ask $color(a, C)$

Trasformare un concetto astratto in un oggetto si chiama *reificazione*. Nell'esempio abbiamo reificato il colore *red*.

Esempio 21. *La rappresentazione di* **color** *come predicato non ci permette di chiedere "Quale proprietà di questo oggetto ha valore* **red***?" dove la risposta corretta sarebbe* **color**. *Per risolvere, anche le proprietà come individui ed inventare la relazione* **prop** *indicando che "l'oggetto* **a** *ha per proprietà* **color** *il valore* **red***" scrivendo*

$prop(a, color, red)$.

Questa rappresentazione permette di rispondere a tutte le query viste.

La rappresentazione *individuo-proprietà-valore* è definita in termini di una singola relazione *prop* dove

prop(Ind,Prop,Val)

significa che l'individuo *Ind* ha il valore *Val* per la proprietà *Prop*. Questa è anche chiamata rappresentazione in triple poiché tutte le relazioni sono rappresentate come triple. Il primo elemento della tripla è detto *soggetto*, il secondo è il *predicato* ed il terzo è l'*oggetto*.

In una tripla, il verbo è una proprietà p con dominio l'insieme di individui che possono essere soggetti in triple con verbo p e codominio l'insieme dei valori che possono essere oggetti in triple con verbo p.

Un *attributo* è una coppia *proprietà-valore*. Ad esempio, per un oggetto un attributo potrebbe essere colore-rosso.

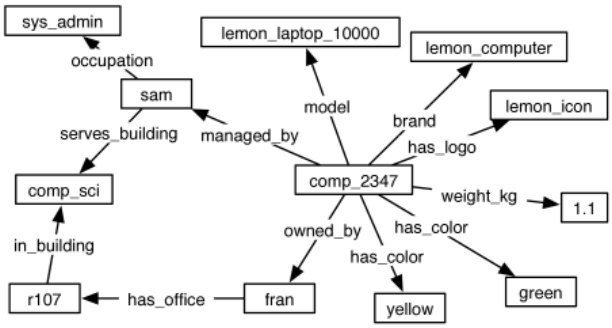


Figura 6.2: Un esempio di knowledge graph.

6.3.2 Rappresentazioni Grafiche

La relazione *prop* può essere interpretata in termini di grafo orientato dove la relazione

prop(Ind,Prop,Val)

è rappresentata con *Ind* e *Val* come nodi collegati da un arco etichettato con *Prop*. Un tale grafo è detto *semantic network* o *knowledge graph* (esempio in Figura 6.2).

Il vantaggio principale di questa notazione grafica è facilitare la visualizzazione delle relazioni agli umani.

6.3.3 Classi

In genere, si conosce di più su un dominio che di un database di fatti. Infatti si conoscono le regole generali da cui altri fatti possono essere derivati. Quali fatti sono esplicitamente dati e quali sono derivati è una scelta da fare quando si progetta una base di conoscenza.

La *conoscenza primitiva* è la conoscenza che viene specificata esplicitamente in termini di fatti. La *conoscenza derivata* è la conoscenza che può essere inferita da altra conoscenza. La conoscenza derivata è tipicamente specificata usando regole.

L'uso di regole consente una rappresentazione più compatta della conoscenza. Le relazioni derivate consentono di trarre conclusioni dalle osservazioni sul dominio. Questo è importante perché non si può osservare tutto riguardo un dominio. Gran parte della conoscenza circa un dominio è dedotta dalle osservazioni e dalla conoscenza più generale disponibile.

Un modo standard per usare la conoscenza derivata è quello di raggruppare gli individui in classi e associare proprietà generali a tali classi in modo che i loro individui le ereditino.

Una *classe* è l'insieme di quegli individui effettivi e potenziali (che potrebbero essere membri della classe). Questo è tipicamente un *insieme intensionale*,

definito da una *funzione caratteristica* che è vera per i membri dell'insieme e falsa di altri individui. L'alternativa a un insieme intensionale è un *insieme estensionale* definito elencando i suoi elementi.

La definizione di classe permette a qualsiasi insieme che può essere descritto di essere una classe. Un *tipo naturale* è una classe che rende più concisa la descrizione degli individui, rispetto alla descrizione che avrebbero se non appartenessero a quella classe. Ad esempio, "mammifero" è un tipo naturale, perché descrivere gli attributi comuni dei mammiferi rende una base di conoscenza che utilizza "mammifero" più concisa di una che non usa "mammifero" e invece ripete gli attributi per ogni individuo.

La classe S è una sottoclasse di C se $S \subseteq C$. Cioè, tutti gli individui di tipo S sono di tipo C.

La relazione tra tipi e sottoclassi è definibile come clausola

$prop(X, type, C) \leftarrow prop(S, subClassOf, C) prop(X, type, S).$

Possiamo considerare *type* e *subClassOf* come proprietà speciali che permettono l'ereditarietà di proprietà. L'ereditarietà di proprietà avviene quando un valore per una proprietà è specificato a livello di classe ed ereditato dai membri della sottoclasse. Possiamo scrivere

$prop(Ind, p, v) \leftarrow prop(Ind, type, c).$

6.4 Ontologie e Condivisione della Conoscenza

Costruire grandi sistemi basati su conoscenza è complesso:

- La conoscenza spesso deriva da molteplici sorgenti e va integrata.
- I sistemi si evolvono nl tempo ed è difficile anticipare le distinzioni che saranno utili in futuro.
- Il mondo da rappresentare è spesso indistinto, spetta ai progettisti scegliere individui e relazioni da rappresentare.
- È difficile memorizzare notazione e significato sia propria che altrui.

  Per condividere e comunicare la conoscenza, è importante essere in grado di sviluppare un vocabolario con un significato comunemente accettato.

  Una *concettualizzazione* è una mappatura tra i simboli utilizzati nel computer, il vocabolario, e gli individui e le relazioni nel mondo. Fornisce una particolare astrazione del mondo con la relativa notazione. Una concettualizzazione per piccole basi di conoscenza può essere nella testa del progettista o specificato in linguaggio naturale nella documentazione. Questa specificazione informale di una concettualizzazione non scala a sistemi più grandi dove la concettualizzazione deve essere condivisa.

  Nell'IA, un'ontologia è una specifica formale della semantica dei simboli utilizzati da un KBS. Ovvero, è una concettualizzazione di individui e relazioni assunti come esistenti e la terminologia relativa.

Le ontologie sono solitamente scritte indipendentemente da una particolare applicazione.

Un'ontologia consiste di

- un vocabolario delle categorie di cose da rappresentare (sia classi che proprietà)
- un'organizzazione delle categorie, ad esempio in una gerarchia di ereditarietà utilizzando proprietà speciali come *subClassOf* o *subPropertyOf*
- un insieme di assiomi che vincolano la definizione di alcuni simboli per riflettere meglio il significato inteso. Ad esempio la transitività di una proprietà, o il numero di valori che la proprietà può assumere per individuo.

In sintesi, lo scopo primario di una ontologia è documentare il significato dei simboli (nella macchina) associandoli a concetti (nella mente del progettista). Lo scopo secondario è consentire l'inferenza o individuare eventuali contraddizioni.

6.4.1 Uniform Resource Identifier

Un *Uniform Resource Identifier (URI)* è un identificatore univoco per una risorsa, dove per risorsa si intende qualsiasi cosa possa essere identificata, compresi individui, classi e proprietà.

La sintassi si basa su quella degli URL e può essere abbreviata localmente con *abbre:name*.

6.4.2 Logiche Descrittive

I moderni linguaggi di ontologie come OWL[4]si basano su *logiche descrittive*.

L'idea principale dietro una descrizione logica è di separare

- una base di conoscenza terminologica (TBox) che descriva la semantica dei simboli
- una base di conoscenza asserzionale (ABox) che specifichi verità fattuali

Solitamente si utilizzano le triple per definire la parte asserzionale e un linguaggio come OWL per definire la parte terminologica.

Con OWL è possibile descrivere domini in termini di

- individui: entità del mondo che si descrive
- Classi: insiemi di individui

[4]OWL (Web ontology Language) è un linguaggio ontologico per il World Wide Web. Descrive un mondo in termini di classi, individui e proprietà. Ha un meccanismo built-in di uguaglianza tra individui, classi e proprietà, in aggiunta ad alcune restrizioni, ad esempio transitività e cardinalità.

- Proprietà: relazioni che descrivono individui associandoli o a valori predefiniti, come interi o stringhe (es. *streetName*), o ad altri individui (es. *nextTo*).

OWL non fa assunzione di unicità dei nomi; due nomi non indicano necessariamente individui diversi o classi diverse. Inoltre non fa l'assunzione di conoscenza completa; non presuppone che tutti i fatti rilevanti siano stati dichiarati.

OWL non è esprimibile attraverso clausole definite. Per dire che tutti gli elementi S hanno il valore v per un dato predicato p, si dice che S è sottoinsieme di quello di tutte le cose vere con valore v per p (assiome di inclusione $\sqsubseteq$).

Una *ontologia di dominio* riguarda un particolare dominio di interesse. Ci sono alcune linee guida per scrivere ontologie di dominio che permettano knowledge sharing:

- Se possibile, usare ontologie esistenti.
- Se esiste un'ontologia che non corrisponda esattamente ai requisiti, la si può importare facendovi delle aggiunte.
- Assicurarsi che l'ontologia si integri con ontologie affini. Utilizzare la stessa terminologia per gli stessi individui è fondamentale in questa direzione.
- Tentare di conformarsi a ontologie di livello superiore (o Top-Level Ontologies)[5].
- Nel progettare una nuova ontologia, consultarsi con altri utenti potenziali.
- Seguire le convenzioni di denominazione, ad esempio chiamando una classe con il nome singolare dei suoi membri: *Resort* e non *Resorts* o peggio ancora *ResortConcept*.
- Specificare la corrispondenza tra ontologie. A volte serve un allineamento tramite matching di ontologie sviluppate in modo indipendente. Da evitare se rende la conoscenza più complicata o ridondante.

Un editor che permette di editare rappresentazioni OWL è Protégé.

[5] Un'ontologia di alto livello, o top-level ontology, fornisce una definizione di tutto a un livello molto astratto. L'obiettivo di un'ontologia di alto livello è quello di fornire una categorizzazione utile su cui basare altre ontologie.

Capitolo 7

Apprendimento Supervisionato

L'*apprendimento* è la capacità di un agente di migliorare il suo comportamento basandosi sull'esperienza. Questo significa

- Estensione delle abilità; l'agente può fare più cose.
- Miglioramento dell'accuratezza nei compiti svolti; l'agente può fare le cose meglio
- Miglioramento dell'efficienza; l'agente può fare le cose più velocemente.

Questo capitolo tratta l'apprendimento supervisionato per effettuare previsioni: dato un insieme di esempi di training costituiti da coppie input-output, prevedere l'output di un nuovo esempio in cui vengono forniti solo gli input.

Esploriamo quattro approcci all'apprendimento:

1. Scegliere una singola ipotesi di modello che si adatti bene agli esempi di training,
2. Prevedere direttamente dagli esempi di addestramento,
3. Selezionare il sottoinsieme di uno spazio di ipotesi coerente con gli esempi di addestramento,
4. Predizioni basate su distribuzioni di probabilità delle ipotesi condizionate dagli esempi di training.

7.1 Problematiche di Apprendimento Automatico

Il problema dell'apprendimento è quello di prendere conoscenza e dati precedenti (ad esempio, sulle esperienze dell'agente) e di creare una rappresentazione interna (la base di conoscenza) che viene utilizzata dall'agente mentre agisce.

Le tecniche di apprendimento affrontano i seguenti problemi:

Task Praticamente qualsiasi attività per la quale un agente può ottenere dati o esperienze può essere appresa. Il compito di apprendimento più comunemente studiato è l'*apprendimento supervisionato*: date alcune features di input, alcune features target e una serie di esempi di training in cui sono specificate sia le features di input e le features target, prevedere il valore delle features target per i nuovi esempi dati i loro valori sulle features di input. Questa si chiama *classificazione* quando le features target sono discrete e *regressione* quando le features target sono continue.

Altri compiti di apprendimento includono l'apprendimento di classificazioni quando gli esempi non hanno target definiti, è il caso dell'*apprendimento non supervisionato*, imparare cosa fare sulla base di ricompense e punizioni (*reinforcement learning*), l'apprendimento di rappresentazioni più espressive, creazione di ranking.

Feedback I compiti di apprendimento possono essere caratterizzati da un feedback dipende dal task di apprendimento.

Nell'apprendimento supervisionato, ciò che deve essere appreso è specificato per ogni esempio di training fornito dall'esperto, quindi l'agente riceve un feedback immediato.

L'apprendimento non supervisionato si verifica quando il learner deve scoprire categorie e pattern nei dati. Allora il feedback, come nell'apprendimento per rinforzo, viene espresso in termini di ricompense e punizioni. Ciò porta al problema dell'*assegnazione del credito*, cioè determinare quali azioni sono responsabili dei premi o delle punizioni.

Rappresentazione Affinché un agente possa utilizzare le proprie esperienze, le esperienze devono influenzare la sua rappresentazione interna. Questa rappresentazione è tipicamente una rappresentazione compatta che generalizza i dati.

Il problema di inferire una rappresentazione interna basata su esempi è chiamato *induzione* in contrasto con la *deduzione*, che deriva conseguenze di una base di conoscenza, e *abduzione*, cioè ipotizzare ciò che può essere vero su un caso particolare.

Ci sono due principi guida (in disaccordo tra loro) nella scelta di una rappresentazione:

- Più ricca è la rappresentazione, più è utile per la successiva risoluzione dei problemi. Affinché un agente impari un modo per risolvere un compito, la rappresentazione deve essere abbastanza ricca da esprimere un modo per risolvere il compito.
- Più ricca è la rappresentazione, più è difficile da imparare.

Le rappresentazioni richieste per l'intelligenza sono un compromesso tra molti *desiderata*. La capacità di imparare la rappresentazione è uno di loro, ma non è l'unico.

Gran parte dell'apprendimento automatico è studiato nel contesto di particolari rappresentazioni (ad esempio, alberi decisionali, reti neurali o basi di casi).

Online e offline Nell'*apprendimento offline*, tutti gli esempi di training sono disponibili per l'agente prima di agire. Nell'*apprendimento online*, gli esempi di formazione arrivano mentre l'agente agisce. Un agente che apprende online ha una rappresentazione iniziale. Quando si osservano nuovi esempi, l'agente deve aggiornare tale rappresentazione. Tipicamente non si possono osservare tutti i potenziali esempi, l'*apprendimento attivo* è una forma di apprendimento online in cui l'agente agisce per acquisire esempi utili da cui imparare.

Misurare il successo L'apprendimento è definito in termini di miglioramento delle prestazioni sulla base di una certa metrica. La metrica di solito non misura quanto bene l'agente agisce sui dati di training, ma quanto bene l'agente agisce per i nuovi dati.

Un modo standard per valutare una procedura di apprendimento è quello di dividere gli esempi in *esempi di training* (o training set) ed *esempi di test* (o test set). Una rappresentazione viene costruita utilizzando gli esempi di training e l'accuratezza predittiva viene misurata sugli esempi di test. Per valutare correttamente il metodo, i casi di test non devono essere noti in fase di training. Naturalmente, l'utilizzo di un test set è solo un'approssimazione di ciò che si vuole, saranno i dati reali a fornire una performance effettiva.

Bias La tendenza a preferire un'ipotesi rispetto ad un'altra è chiamata *bias*. Per ottenere un qualsiasi processo induttivo atto a fare previsioni su dati mai visti prima, per un agente è necessario avere un *bias*. La scelta del bias è un problema empirico da risolvere attraverso la pratica.

Ricerca Data una rappresentazione e un bias, il problema dell'apprendimento può essere ridotto a quello della ricerca. L'apprendimento è una ricerca attraverso lo spazio delle possibili rappresentazioni (o spazio delle ipotesi), cercando di trovare la rappresentazione che meglio si adatta ai dati dato il bias. Sfortunatamente, gli spazi di ricerca sono in genere proibitivi per la ricerca sistematica. Quasi tutte le tecniche di ricerca utilizzate nell'apprendimento automatico possono essere viste come forme di ricerca locale su uno spazio di rappresentazioni. La definizione dell'algoritmo di apprendimento diventa quindi quella di definire lo spazio di ricerca, la funzione di valutazione e il metodo di ricerca.

Rumore Nella maggior parte delle situazioni del mondo reale, i dati non sono perfetti. Ci può essere *rumore* per vari motivi:

- dall'aver osservato features inutili alla classificazione
- dati mancanti dovuti alla mancata osservazione features utili,
- valori errati di alcun features.

Interpolazione ed estrapolazione L'*interpolazione* comporta effettuare predizioni per i casi compresi tra i dati osservati. L'*estrapolazione* comporta fare una predizione per i casi che vanno oltre gli esempi visti.

L'estrapolazione è solitamente molto meno accurata dell'interpolazione.

7.2 Apprendimento Supervisionato

Come già detto, un obbiettivo di apprendimento automatico è l'apprendimento supervisionato, caratterizzato da un insieme di esempi ed un insieme di features, queste ultime suddivise in features di input e feature target. Una *features* è una funzione che va dall'insieme degli esempi ad un valore. Dato l'esempio e e la feature F, $F(e)$ è il valore della feature F per l'esempio e. Il dominio di una feature è l'insieme dei valori che può restituire.

In un task di apprendimento supervisionato sono dati:

- un insieme di features di input, $X_1, \ldots, X_n$
- un insieme di features target, $Y_1, \ldots, Y_k$
- un insieme di esempi di training, dove i valori per le features di input e le features target sono date per ogni esempio
- un insieme di esempi di test, dove sono dati solo i valori per le features di input.

L'obiettivo è predire i valori per le features target per gli esempi di test e gli esempi non ancora visti.

7.2.1 Valutare le Predizioni

La *stima puntuale* per una feature target Y su un esempio e è la predizione del valore $Y(e)$ e la indichiamo con $\widehat{Y}(e)$. L'*errore* misura quanto $\widehat{Y}(e)$ sia vicino a $Y(e)$.

In caso di regressione sia $\widehat{Y}(e)$ che $Y(e)$ sono numeri reali confrontabili aritmeticamente. In caso di classificazione, Y è una funzione discreta e sono possibili diverse alternative:

- Quando il dominio di Y è *binario*, un valore può essere associato a 0 e l'altro ad 1 (o $-1, 1$, o zero, non-zero). Il valore predetto potrebbe essere un numero qualsiasi oppure essere "normalizzato" a 0 o 1. I valori effettivi e quelli predetti possono essere comparati numericamente.

- Per una feature *cardinale* i valori sono mappati su numeri reali. È una mappatura appropriata quando i valori nel dominio di Y sono totalmente ordinati e la loro differenza è significativa. In questo caso, il valore predetto e quello effettivo possono essere confrontati su questa scala.

  Spesso, mappare i valori sui numeri reali non è appropriato anche se i valori sono totalmente ordinati. In questo caso la feature è detta *ordinale*. Ad esempio, dato il dominio $\{small, medium, large\}$, la predizione $\widehat{Y}(e) \in \{small, large\}$ è molto diversa da $\widehat{Y}(e) = medium$.

- Per una feature Y totalmente ordinata, dato un suo valore v, si può definire una feature booleana $Y \leq v$, che funga da *cut*, la quale ha valore 1 quando $Y \leq v$ e 0 altrimenti. Combinare più feature di cut permette di avere features con dominio ad intervalli.

- Quando Y è discreta con dominio $\{v_1, \ldots, v_k\}$, con $k > 2$, si possono avere predizioni separate per ciascun v_i. Questo può essere effettuato utilizzando *variabili-indicatore* binarie Y_i associate ad ogni v_i dove $Y_i(e) = 1$ se $Y(e) = v_i$, e $Y_i(e) = 0$ altrimenti. Per ogni esempio e, esattamente una tra $Y_1(e), \ldots, Y_k(e)$ sarà uguale a 1, e le altre saranno tutte uguali a 0.

Esempio 22. *Imparare le preferenze degli utenti sulla lunghezza delle vacanze, supponendo una durata compresa tra 1 e 6 giorni.*

- *Una rappresentazione consiste nell'avere una variabile Y a valori reali nel numero dei giorni*

Esempio	Y
e_1	*1*
e_2	*6*
e_3	*6*
e_4	*2*
e_5	*1*

- *Un'altra rappresentazione è in termini di variabili indicatore, $Y_1, \ldots, Y_6$, dove la Y_i rappresenta la proposizione che alla persona piacerebbe una vacanza di i giorni*

Esempio	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6
e_1	*1*	*0*	*0*	*0*	*0*	*0*
e_2	*0*	*0*	*0*	*0*	*0*	*1*
e_3	*0*	*0*	*0*	*0*	*0*	*1*
e_4	*0*	*1*	*0*	*0*	*0*	*0*
e_5	*1*	*0*	*0*	*0*	*0*	*0*

- *Un terza rappresentazione consiste nell'avere una feature cut binaria $Y \leq i$ per i vari valori di i*

Esempio	$Y \leq 1$	$Y \leq 2$	$Y \leq 3$	$Y \leq 4$	$Y \leq 5$
e_1	*1*	*1*	*1*	*1*	*1*
e_2	*0*	*0*	*0*	*0*	*0*
e_3	*0*	*0*	*0*	*0*	*0*
e_4	*0*	*1*	*1*	*1*	*1*
e_5	*1*	*1*	*1*	*1*	*1*

Nelle seguenti misure dell'errore, Es è l'insieme degli errori e T è l'insieme delle features target. Per la feature target $Y \in T$ e l'esempio $e \in Es$, il valore effettivo è $Y(e)$ e quello predetto è $\widehat{Y}(e)$.

- L'*errore 0/1* su Es è la somma del numero di predizioni errate

$$\sum_{e \in Es} \sum_{Y \in T} Y(e) \neq \widehat{Y}(e)$$

  dove $Y(e) \neq \widehat{Y}(e)$ vale 0 quando è falso, 1 quando è vero.

- L'*errore assoluto* su Es è la somma delle differenze in valore assoluto tra il valore effettivo e quello predetto

$$\sum_{e \in Es} \sum_{Y \in T} |Y(e) - \widehat{Y}(e)|$$

  Questo è sempre un valore non negativo, vale zero quando tutte le predizioni sono corette.

- L'*errore somma dei quadrati* su Es è

$$\sum_{e \in Es} \sum_{Y \in T} (Y(e) - \widehat{Y}(e))^2$$

  Questa misura enfatizza gli errori proporzionalmente alla loro grandezza.

- L'*errore nel caso pessimo* su Es è la differenza massima in valore assoluto tra valori effettivi e stime

$$\max_{e \in Es} \max_{Y \in T} |Y(e) - \widehat{Y}(e)|$$

  In questo caso la valutazione si basa su quanto si può sbagliare il learner.

Gli errori appena visti spesso sono definiti in termini di *norme* delle differenze (vettoriali) tra predizioni ed errori effettivi. L'errore 0/1 è detto errore L_0, non esattamente una norma. L'errore assoluto è detto errore L_1 e corrisponde alla norma uno, l'errore somma dei quadrati è detto errore L_2 e corrisponde alla norma due, infine l'errore nel caso pessimo è detto errore L_∞ e corrisponde alla norma infinito.

Un *outlier* è un esempio che non segue il pattern degli altri esempi. La differenza tra le rette che minimizzano le varie misure di errore è resa più evidente proprio da come trattano gli esempi outlier.

Per il caso speciale in cui il dominio di Y è $\{0, 1\}$, e la predizione è nel range $[0, 1]$, si possono utilizzare le seguenti misure per predizioni binarie:

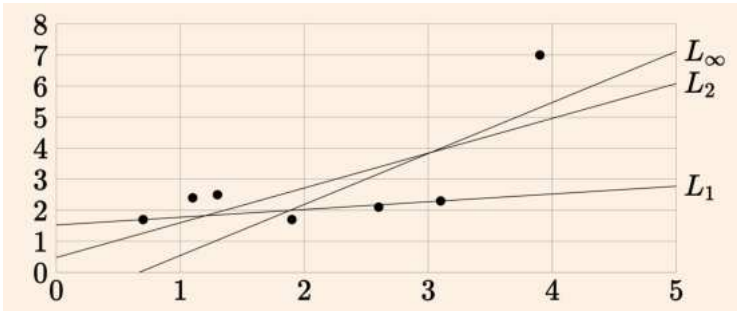


Figura 7.1: Regressione lineare per un semplice esempio di predizione. I punti neri sono gli esempi di training. L_1 è la predizione che minimizza l'errore assoluto degli esempi di training. L_2 è la predizione che minimizza la somma dei quadrati degli errori negli esempi di training. L_∞ è la predizione che minimizza il caso peggiore negli esempi di training.

- La *verosimiglianza dei dati* (o likelihood) interpreta $\widehat{Y}(e)$ come probabilità, con predizioni indipendenti sugli esempi

$$L = \prod_{e \in Es} \prod_{Y \in T} \widehat{Y}(e)^{Y(e)} (1 - \widehat{Y}(e))^{(1-Y(e))}$$

  L'esponente può essere 1 o 0, quindi questa produttoria utilizza $\widehat{Y}(e)$ quando $Y(e) = 1$ e $(1 - \widehat{Y}(e))$ quando $Y(e) = 0$.

- La *log-likelihood* è il logaritmo della likelihood

$$LL = \sum_{e \in Es} \sum_{Y \in T} (Y(e) \log \widehat{Y}(e) + (1 - Y(e)) \log(1 - \widehat{Y}(e)))$$

  Queste due metriche vanno massimizzate per ottenere predizioni più accurate. Per rendere queste misure delle grandezze da minimizzare, si utilizza la *log-loss*[1], definita come il negativo del log-likelihood diviso il numero degli esempi $(-LL/|Es|)$.

7.2.2 Tipo di Errore

Non tutti gli errori sono uguali, le conseguenze di alcuni tipi di errori possono essere più gravi delle conseguenze di altri tipi. Ad esempio, in medicina non diagnosticare una malattia ad un paziente, che quindi non si curerà, è più grave rispetto a predire una malattia che non ha, consigliando quindi ulteriori accertamenti.

Consideriamo il caso semplice in cui il dominio della feature target è booleano e le predizioni sono booleane. Un modo per valutare una predizione è di

[1] La misura log-loss è strettamente correlata alla nozione di *entropia* della teoria dell'informazione. Per entropia si intende il contenuto informativo in una codifica binaria.

	actual positive (ap)	actual negative (an)
predict positive (pp)	true positive (tp)	false positive (fp)
predict negative (pn)	false negative (fn)	true negative (tn)

Figura 7.2: Matrice di confusione per features booleane.

considerare i quattro casi possibili tra valore effettivo e valore stimato, riportati in Figura 7.2.

Un *errore falso positivo* è una predizione positiva che è sbagliata. Un *errore falso negativo* è una predizione negativa che che è sbagliata.

Per un dato *predittore* (o modello predittivo) binario, e per un dato insieme di esempi siano tp il numero di veri positivi, fp il numero di falsi positivi, tn il numero di veri negativi e fn il numero di falsi negativi. Spesso si utilizzano le seguenti metriche:

- La *precision* è $\frac{tp}{tp+fp}$, la proporzione di predizioni positive che sono veri positive.
- La *recall* (o true-positive rate) è $\frac{tp}{tp+fn}$, la proporzione di veri positivi che sono stati predetti come positivi.
- Il *false-positive rate* è $\frac{fp}{fp+tn}$, la proporzione di veri negativi che sono stati predetti come positivi.

Un agente dovrebbe massimizzare precision e recall e minimizzare il false-positive rate.

Per confrontare dei modelli predittivi per un dato insieme di esempi si può considerare uno *spazio ROC* (receiver operating characteristic), ovvero un grafico che confronta false-positive rate e true-positive rate (o recall). In questo spazio ogni predittore diventa un punto.

Altrimenti, si può considerare uno *spazio precision-recall*, cioè un grafico che confronta precion e recall.

7.3 Modelli Base per l'Apprendimento Supervisionato

La maggior parte dei metodi di apprendimento supervisionato prende le feature di input, le feature target e i dati di training e restituisce una rappresentazione compatta di una funzione che può essere utilizzata per future predizioni su esempi mai visti. Un'alternativa a questo è il ragionamento basato sui casi, che utilizza direttamente gli esempi piuttosto che costruire un modello (CBR e k-NN).

Gli algoritmi di apprendimento differiscono tra loro in base alla rappresentazione delle funzioni.

7.3.1 Alberi di Decisione

Un *albero di decisione* è una delle più semplici tecniche utili per l'apprendimento della classificazione supervisionata.

Un albero di decisione o un albero di classificazione è un albero in cui

- ogni nodo interno (non foglia) è etichettato con una condizione, una funzione booleana sui valori delle feature degli esempi
- ogni nodo interno ha due figli, uno etichettato con $true$ e l'altro con $false$
- ogni foglia dell'albero è etichettata con una stima puntuale sulla classe.

Per classificare un esempio, ogni condizione riscontrata nell'albero viene valutata e viene seguito l'arco corrispondente al risultato. Quando si raggiunge una foglia, viene restituita la classe corrispondente. Un albero di decisione corrisponde a una struttura annidata if-then-else in un linguaggio di programmazione.

Per utilizzare gli alberi decisionali ci sono una serie di domande che sorgono:

- Dati alcuni esempi di training, quale albero di decisione dovrebbe essere generato? Poiché un albero di decisione può rappresentare qualsiasi funzione con features di input (X_i) discrete, il bias necessario è insito nel criterio di preferenza tra i diversi alberi. Una proposta è quella di preferire l'albero più piccolo (purché coerente con i dati), che potrebbe essere l'albero con la minore profondità o l'albero con il minor numero di nodi. Quali alberi decisionali siano i migliori predittori per dati non ancora visti è una domanda empirica.
- Come dovrebbe fare un agente per costruire un albero di decisione? Un modo è quello di cercare nello spazio degli alberi decisionali il più piccolo albero di decisione che si adatta ai dati. Sfortunatamente, lo spazio degli alberi decisionali è enorme. Una soluzione pratica è quella di effettuare una ricerca greedy sullo spazio degli alberi di decisione, con l'obiettivo di minimizzare l'errore.

Ricerca di un Buon Albero di Decisione

Un algoritmo ricorsivo per costruire un albero di decisione prende in input un insieme di features di input, un insieme di target feature ed un insieme di esempi di training. Se le feature sono booleane possono essere utilizzate direttamente come condizioni ai nodi.

Un albero di decisione prende un esempio e restituisce una previsione per quell'esempio che può essere:

- o una predizione che ignora le feature di input (una foglia)
- o una predizione nella forma "se $c(e)$ allora t_1 altrimenti t_0, dove t_1 e t_0 sono alberi di decisione e t_1 viene usato quando la condizione c è vera per l'esempio e, e t_0 viene usato quando $c(e)$ è falsa.

Il modello prima testa se il criterio di stop è soddisfatto. Se sì, restituisce una stima puntuale o un valore su una distribuzione di probabilità per Y (Y è una target feature).

Se il criterio di stop non è soddisfatto, il modello usa la condizione c per dividere gli esempi in due partizioni, quelli che soddisfano $c(e)$ e quelli che soddisfano $\neg c(e)$.

Come si può capire, ci sono alcuni dettagli da specificare.

- Il modello ha bisogno di un *criterio di stop* per determinare quando le condizioni sono esaurite o tutti gli esempi appartengono alla stessa classe. Alcuni metodi:
  - Numero minimo di esempi dei nodi-figli, non partizionare ulteriormente se uno dei figli avrebbe un numero di esempi inferiore rispetto ad un certo treshold.
  - Minimo numero di esempi ad un nodo, non partizionare ulteriormente se nel nodo c'è un numero di esempi inferiore rispetto ad un certo treshold.
  - Profondità massima, non partizionare ulteriormente se la profondità dell'albero ha raggiunto un limite prestabilito.
- Va stabilito l'*output* restituito dalle foglie. Una stima puntuale che non dipenda da tutte le features di input, ma solo dalle condizioni nel percorso (cioè le features incontrate nel percorso verso la foglia). Tipicamente si tratta della classificazione più probabile: media, mediana o distribuzione di probabilità sulle classi.
- Va stabilito quale condizione scegliere per creare le partizioni. L'obiettivo è scegliere una condizione che porti all'albero più piccolo. Il modo standard per farlo è applicare una scelta greedy; ad ogni passo, il modello sceglie una qualsiasi condizione che porta alla migliore classificazione.

Abbiamo considerato il caso di features di input booleane, utilizzate direttamente come condizioni di partizionamento. Le feature di input non booleane possono essere gestite in due modi:

- Espandere l'algoritmo per consentire un partizionamento a più vie (multiway split). Così facendo ci sarebbe un figlio per ogni valore nel dominio della variabile. Ciò significa che la rappresentazione dell'albero decisionale diventa più complicata della semplice forma if-then-else usata per le feature binarie.

  Ci sono due problemi principali con questo approccio. Il primo è cosa fare con i valori di una caratteristica per cui non ci sono esempi di training. Il secondo è che per la maggior parte delle euristiche di partizionamento miopi è meglio dividere su una variabile con un dominio più grande perché produce più figli e quindi può adattarsi meglio ai dati. Tuttavia, suddividere una funzione con un dominio più piccolo mantiene la rappresentazione più compatta.

- Suddividere il dominio di una feature di input in due sottoinsiemi disgiunti (come con le feature ordinali o le variabili indicatore).

  Se il dominio della variabile di input X è totalmente ordinato, come condizione si può usare un cut del dominio. Cioè, i figli potrebbero corrispondere a $X \leq v$ e $X > v$ per un certo valore v.

  Se il dominio di X è discreto ma senza ordine naturale, si può effettuare un partizionamento arbitrario.

Un problema importante nell'apprendimento automatico è l'*overfitting* (sovradattamento) ai dati. Esso si verifica quando l'algoritmo cerca di adattarsi a criteri che valgono per i dati di training ma non in generale (esempi di test, o dati futuri).

Ci sono due modi principali per superare il problema del sovradattamento negli alberi decisionali:

- Limitare il partizionamento applicandolo solo quando è utile, ad esempio solo quando l'errore sul training-set di si riduce di più di una certa soglia.
- Consentire il partizionamento senza restrizioni e poi potare l'albero nei punti in cui effettua distinzioni inutili.

7.3.2 Regressione e Classificazione Lineari

Le funzioni lineari forniscono una base per molti algoritmi di apprendimento. Affrontiamo prima la regressione - il problema di stimare una funzione reale da esempi di training - successivamente consideriamo il caso discreto di classificazione.

La *regressione lineare* è il problema di adattare una funzione lineare a una serie di esempi di training, in cui le feature di input e target sono numeriche.

Supponiamo che le feature di input, $X_1, \ldots, X_n$, siano tutte numeriche, e ci sia una singola feature target Y. Una *funzione lineare* delle feature di input è una funzione della forma

$$\widehat{Y}^{\overline{w}}(e) = w_0 + w_1 * X_1(e) + \cdots + w_n * X_n(e) = \sum_{i=0}^{n} w_i * X_i(e)$$

dove $\overline{w} =< w_0, w_1, \ldots, w_n >$ è una tupla di pesi. Per evitare che w_0 sia un caso speciale, inventiamo una feature nuova, X_0, il cui valore è sempre 1.

Supponiamo che Es sia un insieme di esempi. L'*errore quadratico* sugli esempi di Es per il target Y è

$$error(Es, \overline{w}) = \sum_{e \in Es} (Y(e) - \widehat{Y}^{\overline{w}}(e))^2 = \sum_{e \in Es} \left(Y(e) - \sum_{i=0}^{n} w_i * X_i(e) \right)^2 \quad (7.1)$$

In questo caso lineare, il vettore dei pesi $\overline{w}$ è calcolabile analiticamente. In generale si utilizza un approccio iterativo.

Il *gradient descent* è metodo iterativo per la ricerca di un minimo locale di una funzione. Per minimizzare *error*, il gradient descent parte con un insieme di pesi (causali) per $\overline{w}$ e, ad ogni passo, decremento ogni w_i in proporzione alla sua derivata parziale

$$w_i \leftarrow w_i - \eta \cdot \frac{\partial}{\partial w_i} error(Es, \overline{w})$$

dove η è detto *learning rate* (gadient descent step size) e le derivate misurano l'influenza sull'errore di piccole variazioni dei pesi.

L'errore quadratico per una funzione lineare è convesso e ha un minimo locale univoco, che coincide il minimo globale. Poiché la discesa del gradiente con dimensioni del passo abbastanza piccole convergerà a un minimo locale, allora convergerà al minimo globale.

Consideriamo di minimizzare l'errore quadratico. La derivata parziale dell'errore nell'Equazione 7.1 rispetto ai pesi w_i è

$$\frac{\partial}{\partial w_i} error(Es, \overline{w}) = \frac{\partial}{\partial w_i} \sum_{e \in Es} \left(Y(e) - \sum_{i=0}^{n} w_i * X_i(e) \right)^2 = \sum_{e \in Es} -2 \cdot \delta(e) \cdot X_i(e) \tag{7.2}$$

con $\delta(e) = Y(e) - \widehat{Y}^{\overline{w}}(e)$.

Nella discesa del gradiente incrementale (*incremental gradient descent*), l'algoritmo aggiornerà i pesi dopo aver analizzato tutti gli esempi o dopo ogni esempio $e \in Es$:

$$w_i \leftarrow w_i + \eta \cdot \delta(e) \cdot X_i(e)$$

con costante moltiplicativa 2 assorbita da η.

Come criterio di terminazione dell'algoritmo (iterativo) si può utilizzare un numero massimo di iterate, una certa tolleranza nell'approssimazione dell'errore, oppure cambiamenti nei pesi inferiori ad una certa soglia.

Un'alternativa al gradient descent incrementale è il gradietn descent stocastico, un metodo in cui gli esempi vengono scelti casualmente caratterizzato da passi incrementali meno costosi, una convergenza più veloce, ma non garantita.

Ancora, una terza versione della discesa del gradiente è il gradient descent a lotti (batched gradient descent), in cui l'algoritmo calcola i nuovi pesi dopo ogni esempio, ma li applica solamente a fine lotto. Solitamente si prte con lotti piccoli per imparare più rapidamente e si ingrandisce via via la dimensione dei lotti in vista della convergenza.

Funzioni Lineari Appiattite

Si consideri la classificazione binaria, dove il dominio della variabile target è $\{0, 1\}$ e se ci sono più variabili binarie target, esse possono essere apprese separatamente.

L'uso di una funzione lineare non funziona bene per tali compiti di classificazione; un modello non dovrebbe poter effettuare previsioni superiore a 1 o inferiore a 0.

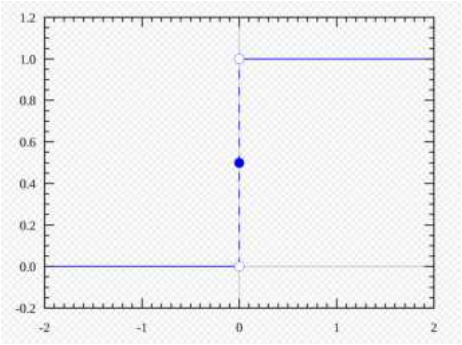


Figura 7.3: Funzione gradino o funzione gradino di Heaviside.

Una *funzione lineare appiattita (o squashed)* è nella forma

$$\widehat{Y}^{\overline{w}}(e) = f(w_0 + w_1 * X_1(e) + \cdots + w_n * X_n(e)) = f\left(\sum_{i=0}^{n} w_i * X_i(e)\right)$$

dove f è una funzione di attivazione definita $\mathbb{R} \to [0, 1]$.

Una predizione basata su una funzione lineare appiattita è un *classificatore lineare*. Una semplice funzione di attivazione è la *funzione gradino* (o funzione gradino) in Figura 7.3, definita come

$$step(x) = \begin{cases} 1 \text{ se } x \geq 0 \\ 0 \text{ se } x < 0 \end{cases}$$

La funzione di attivazione a gradino è alla base del *percettrone*, uno dei primi modelli di apprendimento. Per questa funzione, però, è difficile applicare il gradient descent in quanto la funzione step non è continua, dunque non derivabile, dunque non differenziabile.

Se la funzione di attivazione è differenziabile è possibile utilizzare il gradient descent per aggiornare i pesi. Un esempio di funzione di attivazione differenziabile è il *sigmoide* o *funzione logistica* in Figura 7.4:

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

Questa funzione appiattisce le predizioni portandole nell'intervallo $]0, 1[$.

Il problema di determinare i pesi per il sigmoide di una funzione lineare che minimizza un errore su un insieme di esempi è chiamata *regressione logistica*[2].

[2] Attenzione, si utilizza il termine "regressione" perché si tratta di un modello lineare, ma si applica ai task di classificazione, da non confondere con i task di regressione in cui si dispone di un numero di variabile di input e una variabile target continua e si cerca di trovare una relazione tra queste al fine di prevedere un risultato.

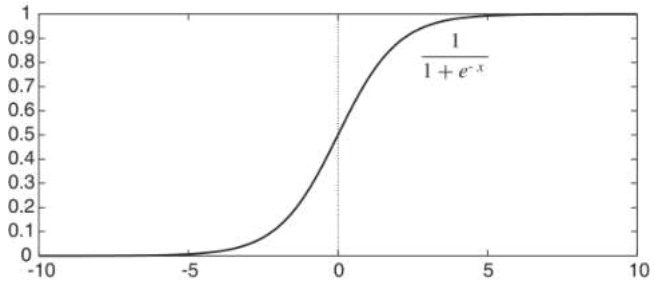


Figura 7.4: Grafico della funzione logistica.

Applicando la regressione logistica alla funzione lineare dei pesi si ottiene il modello di classificazione dato da

$$\widehat{Y}^{\overline{w}}(e) = sigmoid\left(\sum_{i=0}^{n} w_i \cdot X_i(e)\right)$$

Per ottimizzare l'errore *log loss* (quindi determinare $\overline{w}$) con la funzione logistica, si minimizza la log-likelihood negativa

$$LL(Es, \overline{w}) = -\left(\sum_{e \in Es} (Y(e) \cdot \log \widehat{Y}(e) + (1 - Y(e)) \cdot \log(1 - \widehat{Y}(e)))\right)$$

dove il denominatore $|Es|$ viene assorbito come costante moltiplicativa.

La derivata parziale dell'errore log-likelihood negativa rispetto ai pesi w_i è

$$\frac{\partial}{\partial w_i} LL(Es, \overline{w}) = \sum_{e \in Es} -2 \cdot \delta(e) \cdot X_i(e)$$

con $\delta(e) = Y(e) - \widehat{Y}^{\overline{w}}(e)$.

Pertanto, un algoritmo basato su discesa del gradiente incrementale aggiornerà ogni peso secondo l'assegnazione

$$w_i \leftarrow w_i + \eta \cdot \frac{\partial}{\partial w_i} LL(Es, \overline{w})$$

che per le derivate parziali dell'errore log-loss diventa

$$w_i \leftarrow w_i - \eta \cdot \delta(e) \cdot X_i(e)$$

con $\delta(e) = Y(e) - \widehat{Y}^{\overline{w}}(e)$ e costante moltiplicativa 2 assorbita in η.

Se, invece del log-loss, si volesse minimizzare l'errore quadratico, va cambiata la derivata parziale

$$update \leftarrow \eta \cdot error \cdot pred(w) \cdot (1 - pred(e))$$

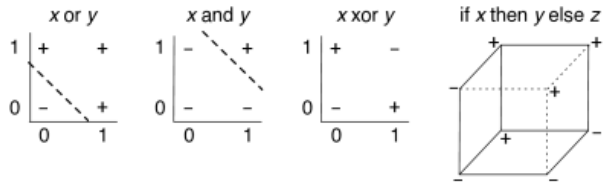


Figura 7.5: Separatori lineari per funzioni Booleane.

Separabilità lineare

Si considera ogni feature di input come una dimensione; se ci sono n feature, avremo uno spazio n-dimensionale. Un *iperpiano* in uno spazio n-dimensionale è un insieme di punti che soddisfano un vincolo espresso come soluzione di una funzione lineare sulle variabili ($\sum_i w_i \cdot X_i = 0$). L'iperpiano forma uno spazio $(n-1)$-dimensionale.

Un classificatore è *linearmente separabile* se esiste un iperpiano per il quale la classificazione è vera da un lato e falsa dall'altro (o maggiore e minore rispetto ad un certo valore).

Esempio 23. *La figura 7.5 mostra i separatori lineari per gli operatori logici "or" e "and". La linea separa i casi positivi da quelli negativi. Una semplice funzione che non è linearmente separabile è lo "xor".*

Spesso è difficile determinare a priori se un insieme di data è linearmente separabile. Inoltre, quando il dominio delle feature target ha più di due valori - ci sono più di due classi - si possono utilizzare le variabili indicatore per convertire il dominio originale in uno binario.

7.4 Overfitting

L'*overfitting* si verifica quando il modello fa previsioni basate su regolarità che compaiono negli esempi di training, ma non negli esempi di test o nel mondo da cui vengono tratti i dati.

Esempio 24. *Consideriamo un sito web in cui le persone inviano valutazioni per ristoranti da 1 a 5 stelle. Supponiamo che i progettisti del sito web vogliano visualizzare i migliori ristoranti. È estremamente improbabile che un ristorante che ha molte valutazioni, non importa quanto sia eccezionale, avrà una media di 5 stelle, perché ciò richiederebbe che tutte le valutazioni siano 5 stelle. Tuttavia, dato che le valutazioni a 5 stelle non sono così rare, è probabile che un ristorante con una singola valutazione abbia 5 stelle. Se i progettisti hanno utilizzato la valutazione media, i ristoranti migliori risulteranno essere quelli con poche valutazioni, nonostante i ristoranti con poche recensioni difficilmente sono effettivamente i migliori. Allo stesso modo, per i ristoranti con poche valutazioni, ma tutte basse, è improbabile che siano così male come indicano le valutazioni.*

Il fenomeno dell'overfitting è legato alla cosidetta *regressione verso la media* in cui i valori sono determinati da qualità e caso, dove l'importanza della casualità può essere diminuita con una grande quantità di dati.

L'overfitting, inoltre, dipende dalla complessità del modello: un modello più complesso si adatta ai dati meglio di un modello più semplice[3].

L'Esempio 24 mostra come più dati possono consentire previsioni migliori. Vorremmo grandi quantità di dati per fare buone previsioni. Tuttavia, anche quando abbiamo i cosiddetti big data, il numero di feature tende a crescere così come il numero di esempi dati. Per esempio, una descrizione dettagliata dei pazienti sarebbe unica al mondo, anche se tutte le persone nel mondo sono state incluse nei dati.

L'errore sul test set è causato da:

- *bias*, errore dovuto all'imperfezione del modello appreso. Il bias è basso quando il modello è vicino al ground truth, il processo che ha generato i dati. Il bias può essere diviso in bias di rappresentazione, causato dalla rappresentazione non contenente un'ipotesi vicina al ground-truth, e un bias di ricerca causato dal fatto che l'algoritmo non cerca esaustivamente lo spazio delle ipotesi.

  Ad esempio, abbiamo visto che un albero di decisione può rappresentare una qualsiasi funzione a variabili discrete, e quindi ha un basso bias di rappresentazione. Il problema è che con un gran numero di feature, lo spazio dei possibili alberi di decisione è troppo grande per potervi applicare una ricerca esaustiva. Da questo segue che un albero di decisione può avere un grande pregiudizio di ricerca.

  Un altro esempio è fornito dalla regressione lineare, la quale, se risolta analiticamente ha un ampio bias di rappresentazione e zero bias di ricerca. Ci sarebbe anche un bias di ricerca qualora venisse utilizzato un algoritmo di discesa del gradiente.

- varianza, l'errore dovuto alla mancanza di dati. Un modello più complicato, con più parametri da regolare, richiederà più dati.

  Quindi con una quantità fissa di dati, c'è un compromesso di bias-varianza; possiamo avere un modello complicato che potrebbe essere accurato, ma non abbiamo abbastanza dati per stimarlo in modo appropriato (basso bias e alta varianza), o un modello più semplice che non può essere accurato, ma di cui possiamo ben stimare i parametri con i dati disponibili (alta polarizzazione e bassa varianza).

- noise, l'errore intrinseco dovuto ai dati in base a caratteristiche non modellate o perché il processo che genera i dati è intrinsecamente stocastico.

L'overfitting si traduce in un eccesso di fiducia (oveconfidence), quando il modello è più sicuro nella sua previsione di quanto i dati lo permettano.

[3] Polinomi di ordine maggiore si adattano meglio ai dati rispetto a polinomi di ordine inferiore, ma questo non significa che siano migliori per il training set.

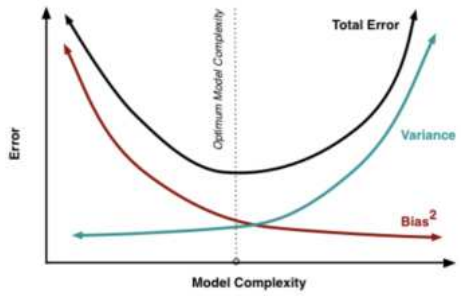


Figura 7.6: Bias-Variance tradeoff

In seguito discutiamo di tre modi per evitare l'overfitting. Il primo permette la regressione alla media, e può essere usato per i casi in cui le rappresentazioni sono semplici. Il secondo fornisce un compromesso tra la complessità del modello e l'adattamento ai dati. Il terzo approccio consiste nell'utilizzare alcuni esempi di training per rilevare l'overfitting.

7.4.1 Pseudoconteggio

Per molte misure di predizione, la predizione ottimale sui dati di training è la media. in caso di dati booleani la media può essere rappresentata come una distribuzione di probabilità.

Tuttavia, la media empirica, la media dell'insieme di training, non è tipicamente una buona stima della probabilità di nuovi casi. Solo perché un agente non ha osservato un certo valore di una variabile non significa che al valore debba essere assegnata una probabilità di zero, il che significa che è impossibile. Allo stesso modo, se dobbiamo fare previsioni per i voti futuri di uno studente, la sua media dei voti può essere appropriata per prevedere i voti futuri dello studente se lo studente ha frequentato molti corsi, ma potrebbe non essere appropriata per uno studente con uno o nessun voto già registrato.

Un modo semplice sia per risolvere sia il problema della *probabilità zero* sia per prendere in considerazione la *conoscenza a priori* è quello di utilizzare uno *pseudo-conteggio* o *conteggio a priori* a cui vengono aggiunti i dati di training (*smoothing additivo*).

Introduciamo il concetto di *media mobile*. Supponiamo di disporre dei valori osservati $v_1, \ldots, v_n$ e che si voglia predire il successivo valore v_{n+1}. La media mobile consiste nell'aggiungere alla media corrente a, cioè la media per gli n valori conosciuti, la quantità $(v_{n+1} - a)/n$. Pertanto, in seguito ad un nuovo valore v, si ha

$$n \leftarrow n + 1$$
$$a \leftarrow a + \frac{(v_n - a)}{n}$$

A tale risultato si arriva tramite la seguente sequenza di semplici calcoli:

$$\begin{aligned} a_n = \frac{v_1 + \cdots + v_{n-1} + v_n}{n} = \frac{v_1 + \cdots + v_{n-1}}{n} + \frac{v_n}{n} = \\ = \frac{v_1 + \cdots + v_{n-1}}{n} \cdot \frac{n-1}{n-1} + \frac{v_n}{n} = \frac{(v_1 + \cdots + v_{n-1})(n-1)}{n(n-1)} + \frac{v_n}{n} = \\ = \frac{n(v_1 + \cdots + v_{n-1}) - (v_1 + \cdots + v_{n-1})}{n(n-1)} - \frac{v_n}{n} = \\ = \frac{(v_1 + \cdots + v_{n-1})}{n-1} - \frac{(v_1 + \cdots + v_{n-1})}{n-1} - \frac{v_n}{n} = \\ a_{n-1} - a_{n-1} \cdot \frac{v_n}{n} = \\ = a_{n-1} + \frac{v_n - a_{n-1}}{n} \end{aligned}$$

Quando $n = 0$ assumiamo di usare una media (predizione iniziale) a_0 che non derivi dai dati. Una predizione che tenga conto della regressione alla media applica la formula

$$v_{n+1} = \frac{v_1 + \cdots + v_n}{n} + \frac{c \cdot a_0}{c} \tag{7.3}$$

dove c è la costante dello pseudoconteggio. Essa controlla la quantità di regressione verso la media. Se $c = 0$ la predizione coincide con la media empirica. Questo può essere implementato nella media mobile inizializzando a con a_0 e n con c.

7.4.2 Regolarizzazione

Il *rasoio di Ockham* specifica che dovremmo preferire modelli più semplici a modelli complessi. Oltre ad ottimizzare l'adattamento ai dati (fit-to-data con lo pseudoconteggio), dovremmo aggiungere un termine alla misura di adattamento in modo da penalizzare la complessità e premiare la semplicità. Questa penalità è detta *regolarizzatore*.

La tipica forma per un regolarizzatore è di trovare una ipotesi h da minimizzare:

$$\left(\sum_{e \in Es} error(e, h) \right) + \lambda \cdot regularizer(h) \tag{7.4}$$

dove $error(e, h)$ è l'errore dell'esempio e per l'ipotesi h e specifica quanto bene h si adatta ad e. Il parametro di regolarizzazione, λ, cerca un compromesso tra adattamento e semplicità del modello, infine $regularizer(h)$ penalizza la complessità del modello, o la deviazione dalla media. Più esempi di prendono in considerazione, maggiore è l'influenza dalla sommatoria rispetto al regolarizzatore.

Per esempio, in un albero di decisione binario, la misura di complessità dipende dal numero di split, o test, effettuati nell'albero (equivale al numero di foglie

meno uno). Quando si costruisce un albero di decisione, potremmo ottimizzare l'errore quadratico aggiungendovi un funzione sulla dimensione dell'albero, minimizzando quindi:

$$\left(\sum_{e\in Es}(Y(e)-\widehat{Y}(e))^2\right)+\lambda\cdot|tree|$$

dove $|tree|$ è il numero di split nell'albero appunto. Uno split in un nodo è utile se riduce l'errore di λ.

Per modelli con parametri a valori reali può essere utilizzato un regolarizzatore L_2 che penalizzi la somma dei quadrati dei parametri. Per ottimizzare l'errore quadratico per la regressione lineare con un regolarizzatore L_2, si minimizza la quantità

$$\left(\sum_{e\in Es}\left(Y(e)-\sum_{i=0}^{n}w_i\cdot X_i(e)\right)^2\right)+\lambda\left(\sum_{i=0}^{n}w_i^2\right)$$

che è conosciuta come *ridge regression*.

Per ottimizzare la log loss error per la regressione logistica con un regolarizzatore L_2, si minimizza la quantità

$$-\left(\sum_{e\in Es}\left(Y(e)\log\widehat{Y}(e)+(1-Y(e))\log(1-\widehat{Y}(e))\right)\right)+\lambda\left(\sum_{i=0}^{n}w_i^2\right)$$

dove $\widehat{Y}(e)=sigmoid(\sum_{i=0}^{n}w_i\cdot X_i(e))$.

Un regolarizzatore L_1, si basa sulla somma dei valori assoluti dei parametri, applicato alla log-loss si minimizza

$$-\left(\sum_{e\in Es}\left(Y(e)\log\widehat{Y}(e)+(1-Y(e))\log(1-\widehat{Y}(e))\right)\right)+\lambda\left(\sum_{i=0}^{n}|w_i|\right).$$

7.4.3 Cross Validation

Il problema con i metodi precedenti è che richiedono di determinare la complessità dei modelli prima di aver visto i dati.

L'idea alla base della *cross validation* consiste nell'utilizzare parte degli esempi di training come esempi di test. Nel caso più semplice, dividiamo il set di training in due partizioni: un insieme di esempi su cui apprendere e un insieme di esempi per la validazione. La previsione sul set di validazione viene utilizzata per determinare quale modello utilizzare.

Si consideri il grafico in Figura 7.7. L'errore sul training set diventa più piccolo come al crescere della dimensione dell'albero. Tuttavia, sul test set l'errore in genere migliora per un po' e poi inizia a peggiorare. La cross validation mira ad una scelta dei parametri, o una rappresentazione di una h, per la quale l'errore sia minimo sul test di validazione. Ipotizzando che questo possa valere anche per il test set.

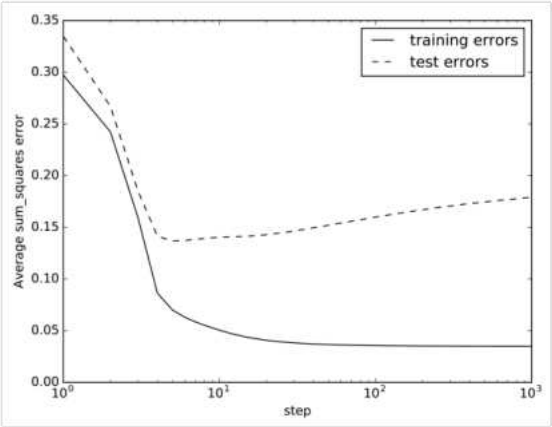


Figura 7.7: Training set error come funzione del numero di step. Sull'asse x è rappresentato il numero di passi di un modello che usa il gradient descent. Sull'asse y è rappresentata la media dell'errore quadratico per il training set (linea continua) e il test set (linea tratteggiata).

Va sottolineato che l'insieme di training e quello di validation devono essere due partizioni separate, non vi devono essere esempi in comune. Usare il set di test come parte dell'apprendimento significa manipolare e invalidare i risultati.

In genere, vogliamo allenarci su più esempi possibili, perché poi otteniamo modelli migliori. Tuttavia, un set di allenamento più grande si traduce in un set di convalida più piccolo e viceversa.

Il metodo della *k-fold cross validation* ci consente di riutilizzare gli esempi sia per il training che per la convalida, utilizzando comunque tutti i dati per training.

I passaggi da effettuare per una cross-validation sono:

- Suddividere casualmente gli esempi di training in k sottoinsiemi, di dimensioni approssimativamente uguali, chiamati fold.
- Per valutare le impostazioni del modello, lo addestro k volte, ognuna con una distinta fold per la validazione e le restanti per l'apprendimento.
- Determinare e restituire le impostazioni ottimali.

Al caso limite, quando k è il numero di esempi di training, la k-fold cross validation diventa una *leave-one-out cross validation*. Con n esempi nel set di training, si apprendono n modelli; per ogni esempio e, utilizza gli altri esempi come set di addestramento e valuta su il modello su e.

7.5 Neural Network e Deep Learning

Le reti neurali (o neural network) sono una rappresentazione popolare per l'apprendimento. Queste reti si ispirano ai neuroni del cervello umano.

Le reti neurali hanno avuto un notevole successo nel ragionamento a basso livello per il quale ci sono abbondanti dati di training come per l'image interpretation, il riconoscimento vocale e la traduzione automatica. Uno dei motivi del successo è che sono molto flessibili e possono inventare nuove feature.

Ci sono diversi tipi di reti neurali. Noi considereremo le reti neurali feed-forward che possono essere viste come una gerarchia costituita da funzioni lineari interconnesse con funzioni di attivazione.

Le reti neurali possono avere molteplici feature di input, ma anche molteplici feature target. Tutte le feature sono a valori reali. Le caratteristiche discrete possono essere trasformate in variabili indicatori o caratteristiche ordinali.

Le feature di input alimentano strati di feature nascoste (*hidden layer*), che possono essere considerate come feature che non vengono mai osservate direttamente, ma sono utili alla predizione. In questi strati nascosti ogni unità (o feature nascosta) è una semplice funzione sulle unità degli strati inferiori (o precedenti). Questi strati di unità nascoste alimentano in definitiva le predizioni delle feature target.

Un'architettura tipica è mostrata nella Figura 7.8. Ci sono layer di unità (mostrati come cerchi). Sul livello inferiore ci sono le unità di input per le

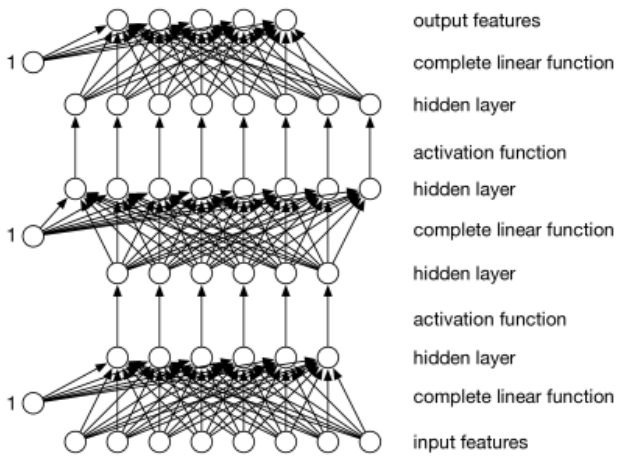


Figura 7.8: Una deep neural network.

feature di input. In alto c'è il livello di output che fa predizioni per le feature target.

Descriviamo tre tipi di livelli in una rete neurale:

- Un *input layer* è costituito da un'unità per ogni feature di input. Ogni unità di questo livello, per restituire un valore al ad ogni unità del layer successivo, viene avvalorato da un esempio.
- Un *complete linear layer*, ogni o_j è una funzione lineare sui valori v_i in input allo stato definita come

  $$o_j = \sum_i w_{ji} v_i$$

  per i pesi w_{ij} da apprendere. C'è un peso per ogni coppia input-output del livello. Nello schema della Figura 7.8 c'è un peso per ogni arco per le funzioni lineari.
- Un layer di attivazione, dove ogni output o_j è funzione del suo valore di input v_i. Ovvero $o_i = f(v_i)$ per una funzione di attivazione f. Una tipica funzione di attivazione utilizzata è il sigmoide.

Per la regressione, dove la previsione può essere qualsiasi numero reale, è tipico che l'ultimo strato sia un livello lineare completo, in quanto ciò consente l'intero intervallo di valori. Per la classificazione binaria, dove i valori di output possono essere mappati a 0,1, è tipico che l'output sia una funzione sigmoide dei suoi input; questo perché, come sempre, non vogliamo mai non prevedere un valore maggiore di uno o inferiore a zero.

La *back-propagation* dell'errore è un algoritmo per l'addestramento delle reti neurali, viene usato, in combinazione con un metodo di ottimizzazione come per

esempio la discesa stocastica del gradiente, per aggiornare il peso w in ragione di $\frac{\partial}{\partial w} error(e)$ per ogni esempio e.

L'apprendimento consiste in due passaggi attraverso la rete per ogni esempio:

1. *Predizione*: per ciascun layer, dati i valori in input, si calcola un valore per l'output.
2. *Backpropagation*: andare indietro attraverso i livelli per aggiornare tutti i pesi della rete (i pesi negli strati lineari).

Trattando ogni strato come un modulo separato, ogni strato deve implementare la predizione "in avanti" e, in fase di back-propagation, aggiornare i pesi nel livello e fornire un termine relativo all'errore per il livello inferiore.

Per rendere questo più modulare, un ulteriore *error layer*, può essere posizionato in cima alla rete. Per ogni esempio, questo livello prende come input la predizione della rete sulle feature target e i valori effettivi delle feature target per quell'esempio, e produce un errore che viene dato in input nel livello finale.

Supponiamo che l'output del layer finale sia un vettore $values$ di previsioni tale per cui $values[j]$ sia la predizione della $j-$esima feature target, ed il valore osservato per l'esempio corrente nell'insieme di training è $Y[j]$. L'*errore* retropropagato per l'errore quadratico è $Y[j] - values[j]$. In fase di back-propagation, l'input per ogni layer è un termine di errore per ognuno delle sue unità di outuput.

Utilizzare layer multipli in una rete neurale può essere visto come una forma di modellazione gerarchica conosciuta come *deep learning.*

7.6 Estensione dei Modelli Lineari (Composite Models)

Gli alberi decisionali e le funzioni lineari (squashed) forniscono la base per molte altre tecniche di apprendimento supervisionato. Le reti neurali ci insegnano che le funzioni lineari diventano più potenti se alternate a funzioni di attivazione non lineari.

Dare in input a funzioni lineari output di funzioni non lineari applicate ai dati di input comporta un aggiunta di feature che incrementa le dimensioni dello spazio rendendo funzioni non lineari (o non linearmente separabili) in uno spazio a meno dimensioni, lineari in uno spazio a più dimensioni.

Esempio 25. *La funzione* xor *non linearmente separabile su un piano, lo diventa in uno spazio a 3-dimensioni, dove le dimensioni sono x_1, x_2 e x_1x_2, con x_1x_2 vera quando entrambe sia x_1 che x_2 sono vere.*

Una *funzione kernel* è una funzione che viene applicata alle feature di input per creare nuove feature che permettano la separabilità lineare dove prima non era possibile.

Un'altra rappresentazione non lineare è un *albero di regressione*, che è un albero di decisione con una funzione lineare (squashed) alle foglie. È anche

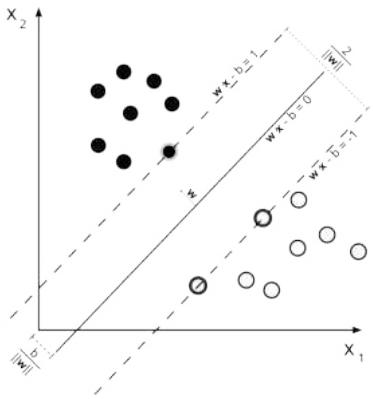


Figura 7.9: Esempio di separazione lineare, usando le SVM.

possibile avere reti neurali o altri classificatori alle foglie dell'albero decisionale. Per classificare un nuovo esempio, l'esempio viene filtrato lungo l'albero e il classificatore alle foglie viene quindi utilizzato per classificare l'esempio.

Un'altra possibilità è quella di utilizzare più di classificatori che sono stati addestrati sui dati e di combinare questi utilizzando alcuni meccanismi come il voto o la media. Queste tecniche sono conosciute come *ensemble learning*.

7.6.1 Support Vector Machine (SVM)

Un *support vector machine*, o classificatore basato su kernel, è un modello utilizzato per task di classificazione che utilizza funzioni di dati di input originali come input della funzioni lineari. Queste funzioni sono chiamate *kernel functions*.

Ad esempio una kernel function è il prodotto delle caratteristiche originali (permette lo xor). Un SVM costruisce una *superficie di decisione*, che è un iperpiano che divide gli esempi positivi e negativi in questo spazio dimensionalmente superiore. Definiamo il *margine* come la distanza minima di uno qualsiasi degli esempi dalla superficie di decisione. Un SVM trova l'iperpiano con il margine massimo. Gli esempi più vicini alla superficie di decisione sono quelli che sostengono la superficie di decisione. In particolare, questi esempi, se rimossi, cambierebbero la superficie di decisione. L'overfitting è evitato perché questi vettori di supporto definiscono una superficie che può essere definita in meno parametri rispetto agli esempi.

7.6.2 Random Forest

Un modello composito semplice ma efficace è quello di avere una media sugli alberi decisionali, nota come *random forest*. L'idea è di avere un certo numero di alberi decisionali, ognuno dei quali può fare una previsione su ogni esempio, e di aggregare le previsioni dei singoli alberi.

Per rendere efficace questa procedura, gli alberi che compongono la foresta devono fare predizioni diverse. Possiamo garantire la diversità in diversi modi:

- Potrebbe essere utilizzato un sottoinsieme di feature per ogni albero.
- Per ogni nodo, si potrebbe scegliere la miglior feature di split all'interno di un sottoinsieme di feature, invece che sull'insieme totale. L'insieme delle feature tra cui scegliere potrebbe cambiare per ogni albero o persino per ogni nodo.
- Ogni albero potrebbe usare un diverso sottoinsieme degli esempi per l'apprendimento. Supponiamo ci siano n esempi con n abbastanza grande, il *bagging* consiste nel suddividere gli n esempi di sottoinsiemi più piccoli ed utilizzarne uno per ogni albero (i sottoinsiemi devono corrispondere a circa il 63% degli esempi originari).

Una volta che gli alberi sono stati addestrati, una previsione può utilizzare la media delle previsioni dell'albero per una previsione probabilistica. In alternativa, ogni albero può votare con la sua classificazione più probabile e potrà essere utilizzata la previsione con il maggior numero di voti.

7.6.3 Ensemble Learning

Nell'*ensemble learning*, un agente prende un numero di modelli e combina le loro previsioni per fare una previsione per d'insieme. Gli algoritmi combinati sono chiamati *algoritmi di base*. Le *random forest* sono un esempio di ensemble learning, dove l'algoritmo di livello base è l'albero di decisione e le previsioni dei singoli alberi sono mediate o sono usate per votare una previsione.

Nel *boosting*, c'è una sequenza di modelli in cui ognuno impara dagli errori dei precedenti. Le caratteristiche di un algoritmo di boosting sono:

- C'è una sequenza di modelli di base che possono essere uguali o diversi tra loro, come piccoli alberi di decisione o funzioni lineari (squashed).
- Ogni modello è addestrato per adattarsi meglio agli esempi sui quali i modelli precedenti non si adattavano bene.
- La predizione finale è un mix (ad esempio, somma, media ponderata o moda) delle previsioni di ogni modello.

I modelli di base possono essere modelli deboli, non hanno bisogno di essere ottimali; hanno solo bisogno di essere meglio di un classificatore casuale. Questi modelli, nel loro insieme avranno prestazioni migliori rispetto ai singoli modelli.

Un semplice algoritmo di boosting è il *functional gradient boosting* che può essere utilizzato per la regressione. Per questo algoritmo la predizione finale, come funzione degli input, è la somma

$$p_0(X) + d_1(X) + d_2(X) + \cdots + d_k(X) \tag{7.5}$$

dove $p_0(X)$ è una predizione iniziale, ad esempio la media, ed ogni d_i è la differenza dalla predizione precedente. Allora l'$i-$esima predizione $p_i = xp_0(X) + d_1(X) + d_2(X) + \cdots + d_i(X) = p_{i-1}(X) + d_i(X)$.

7.7 Case-Based Reasoning (CBR)

I metodi precedenti cercano di trovare una rappresentazione compatta dei dati da utilizzare per la predizione. In caso di ragionamento case-based, gli esempi di training, i "casi", vengono memorizzati per poi essere ritrovati in fase di risoluzione di un nuovo problema. Quindi, per ottenere la predizione per un nuovo esempio, vengono presi gli esempi simili, o *vicini*, e questi ultimi vengono utilizzati per prevedere il valore delle feature target per il nuovo esempio.

Questo è una modalità di apprendimento automatico che si trova sull'estremo opposto rispetto agli alberi di decisione e reti neurali. In questi ultimi modelli infatti, gran parte del lavoro viene fatto offline, mentre nei CBR praticamente tutto il lavoro viene eseguito online, al momento della query.

Il ragionamento case-based è utilizzato sia per task di classificazione che di regressione, ed è applicabile anche quando i casi sono complicati, come per le sentenze giuridiche.

Se i casi sono semplici, un algoritmo che funziona bene è quello di cercare i k esempi più vicini per un certo numero k. Dato un nuovo esempio, vengono utilizzati i k esempi di training con le feature di input più vicine all'esempio dato per predirne le feature target. La previsione potrebbe essere la moda, la media, o una certa interpolazione tra la previsione di questi k esempi di training, magari pesando maggiormente gli esempi più vicini rispetto agli esempi distanti.

Affinché questa metrica funzioni, è necessario definire una certa metrica per la distanza che determini a similarità tra gli esempi. Per prima cosa, va definito un dominio su scala numerica comune per tutte le feature così da renderle confrontabili. Supponiamo che $X_i(e)$ sia la rappresentazione numerica della feature X_i per l'esempio e. Allora $(X_i(e_1) - X_i(e_2))$ è la differenza tra gli esempi e_1 ed e_2 sulla dimensione definita da X_i. Una metrica per la distanza tra due esempi (su tutte le dimensioni) potrebbe essere la *distanza Euclidea*, definita come la radice quadrata della somma dei quadrati delle differenze. Inoltre, introduciamo un peso w_i per ogni $i-$esima dimensione, così da omogeneizzare le scale di feature diverse. La distanza tra due esempi è dunque definita come

$$d(e_1, e_2) = \sqrt{\sum_i w_i \cdot (X_i(e_1) - X_i(e_2))^2}.$$

I pesi delle feature possono essere forniti come input o appresi. L'agente di apprendimento cercherebbe di trovare pesi che minimizzano l'errore nella predizione del valore di ogni esempio di training, per ogni esempio di training. In altre parole, si tratta di un'istanza di cross validation leave-one-out.

Un problema importante nel ragionamento case-based è l'accesso agli esempi rilevanti. A tal proposito si può costruire un *kd-tree* che costruisce un indice per

gli esempi di training e permetta un rapido ritrovamento per similarità. Come gli alberi di decisione, i kd-tree splittano sulle feature di input, ma alle foglie abbiamo sottoinsiemi di esempi di training. Un algoritmo per la costruzione di un kd-tree cerca una feature di input che partizioni il training set in due parti uguali, e ricorsivamente costruisce due kd-tree per le due partizioni. L'algoritmo si ferma quando nelle foglie restano pochi esempi.

Come già detto, il CBR è applicabile anche a quando i casi sono più complicati. Tali casi vengono opportunamente selezionati e modificati per essere utilizzabili.

Capitolo 8

Modelli di Conoscenza Incerta

Gli agenti in ambienti reali sono inevitabilmente costretti a prendere decisioni basate su informazioni incomplete. Ad esempio, un medico non sa esattamente cosa sta succedendo all'interno di un paziente e un insegnante non sa esattamente cosa capisce uno studente.

8.1 Probabilità

Per prendere una buona decisione, un agente non può semplicemente assumere che il mondo sia come gli appare e agire secondo tale presupposto. Deve considerare molteplici ipotesi quando prende una decisione.

Il *ragionamento con incertezza* è stato studiato nei campi della teoria della probabilità e della teoria delle decisioni.

In generale, la probabilità misura il credito, o convinzione, (*belief*) nell'effettuare una certa decisione. La visione della probabilità come misura della convinzione è nota come *probabilità bayesiana* o *probabilità soggettiva*. Il termine soggettivo deriva dal fatto che le decisioni non siano arbitrarie ma basate sulle conoscenze del soggetto.

Per esempio, supponiamo che ci siano tre agenti, Alice, Bob, e Chris, e un dado a sei lati che sia stato lanciato. Supponiamo che Alice osservi che il risultato è un "6" e dice a Bob che il risultato è pari, ma Chris non sa nulla del risultato. In questo caso, per Alice la probabilità che sia uscito un "6" è 1, per Bob la probabilità che sia uscito un "6" è di 1/3 e per Chris la probabilità che sia uscito un "6" è di 1/6. Tutti hanno probabilità diverse perché hanno tutte conoscenze diverse. In questo caso la probabilità si riferisce al particolare lancio del dado in esempio, non di qualche evento generico di lancio dei dadi.

Stiamo assumendo che l'incertezza sia *epistemologica* - che riguarda la conoscenza del mondo da parte di un agente - piuttosto che ontologico - come il mondo è.

La teoria della probabilità è lo studio di *come la conoscenza influisce sulla convinzione*. La credibilità di una proposizione α può essere misurata da un

valore in $[0, 1]$ (per convezione). Se la probabilità di α è 0 significa che α si crede essere sicuramente falsa se la probabilità di α è 1 significa che α si crede essere sicuramente vera.

8.1.1 Proprietà delle distribuzioni e Probabilità Condizionata

Date due qualsiasi proposizioni α e β, sono sempre valide le seguenti proprietà:

- Negazione di una proposizione

$$P(\neg a) = 1 - P(\alpha)$$

- Se $\alpha \leftrightarrow \beta$, allora $P(\alpha) = P(\beta)$. Ovvero, proposizioni logicamente equivalenti hanno la stessa probabilità.
- Ragionamento per casi

$$P(\alpha) = P(\alpha \wedge \beta) + P(\alpha \wedge \neg\beta).$$

- Marginalizzazione: se V è una variabile stocastica con dominio D, allora, per tutte le proposizioni α,

$$P(\alpha) = \sum_{d \in D} P(\alpha \wedge V = d).$$

- Disgiunzione per proposizioni non esclusive

$$P(\alpha \wedge \beta) = P(\alpha) + P(\beta) - P(\alpha \vee \beta)$$

Come detto, la probabilità misura la credibilità, ma quest'ultima va aggiornata quando vi è disponibilità di nuovi fatti.

La misura della credibilità della proposizione h data la proposizione e, o di una ipotesi data una evidenza, è chiamata *probabilità condizionata* di h data e, scritta come $P(h \mid e)$.

Una proposizione e rappresenta la congiunzione di tutte le *osservazioni* dell'agente sul mondo, chiamata *evidenza*. Data una evidenza e, la probabilità condiziona $P(h \mid e)$ è la *probabilità a posteriori* di h. La probabilità $P(h)$ è detta *probabilità a priori* di h, ovvero $P(h \mid true)$, perché è la probabilità di h prima che l'agente osservi qualcosa.

La probabilità condizionata della proposizione h data l'evidenza e è la somma delle probabilità condizionate di tutti in mondi in cui h è vero. Pertanto definiamo

$$P(h \mid e) = \frac{P(h \wedge e)}{P(e)}.$$

La definizione di probabilità condizionata permette di decomporre una congiunzione in un prodotto di probabilità condizionate. Infatti, notiamo che dalla precedente equazione si ottiene

$$P(h \wedge e) = P(e) \cdot P(h \mid e)$$

da cui ricaviamo la seguente *chain rule*:

Proposizione 1. *Per ogni proposizione* $\alpha_1, \ldots, \alpha_n$*:*

$$\begin{aligned} P(\alpha_1 \wedge \alpha_2 \wedge \cdots \wedge \alpha_n) = P(\alpha_1) \cdot \\ P(\alpha_2 \mid \alpha_1) \cdot \\ P(\alpha_3 \mid \alpha_1 \alpha_2) \cdot \\ \cdots \\ P(\alpha_n \mid \alpha_1 \wedge \cdots \wedge \alpha_{n-1}) = \\ \prod_{i=1}^{n} P(\alpha_i \mid \alpha_1 \wedge \cdots \wedge \alpha_{i-1}) \end{aligned} \tag{8.1}$$

Teorema di Bayes

Supponiamo che un agente abbia una belief corrente riguardo una proposizione h basata su un'evidenza k già osservata, data da $P(h \mid k)$, e successivamente osservi e. La sua nuova belief in h è ora $P(h \mid e \wedge K)$. Il teorema di Bayes ci dice come aggiornare la belief dell'agente nell'ipotesi h quando arrivano nuove osservazioni.

Proposizione 2. *Teorema di Bayes*

$$P(h \mid e \wedge k) = \frac{P(e \mid h \wedge k) \cdot P(h \mid k)}{P(e)}.$$

Questo viene spesso scritto con la conoscenza di background k implicita, scrivendo:

$$P(h \mid e) = \frac{P(e \mid h) \cdot P(h)}{P(e)}.$$

Dove $P(e \mid h)$ si chiama verosimiglianza e $P(h)$ è la probabilità a priori dell'ipotesi h. Il teorema di Bayes afferma che la probabilità a posteriori è proporzionale alla verosimiglianza per la probabilità a priori:

$$P(h \mid e) \propto P(e \mid h) \cdot P(h).$$

Spesso il teorema di Bayes si utilizza per il confronto di diverse ipotesi $h_i s$ sulle stesse osservazioni, quindi si tende ad ignorare il denominatore $P(e)$. Inoltre, questo teorema permette di calcolare $P(e \mid h)$ da $P(h \mid e)$ o viceversa in quanto solitamente uno dei due è più facile da stimare dai dati.

8.1.2 Valori Attesi

Il valore atteso di una funzione numerica su più mondi è la media ponderata sui mondi dei valori della funzione

$$\varepsilon_p(f) = \sum_{\omega \in \Omega} f(\omega) \cdot P(\omega)$$

In maniera analoga alla definizione di probabilità condizionata, il *valore atteso condizionato* per f condizionata dall'evidenza e, si definisce come

$$\varepsilon(f \mid e) = \sum_{\omega \in \Omega} f(\omega) \cdot P(\omega \mid e).$$

8.1.3 Informazione ed Entropia

La teoria dell'informazione tratta come rappresentare l'informazione usando i bit. Per un messaggio $x \in \mathit{domain}(X)$, è possibile costruire un codice che identifichi x utilizzando $-\log_2 P(x)$ bit. Il numero di bit atteso per trasmettere un valore x di X è allora

$$H(X) = \sum_{x \in dom(X)} -P(X = x) \cdot \log_2 P(X = x)$$

Questo è il *contenuto informativo* o l'*entropia* di una variabile stocastica X.
L'entropia di X data l'osservazione $Y = y$ è

$$H(X \mid Y = y) = \sum_{x} -P(X = x \mid Y = y) \cdot \log_2 P(X = x \mid Y = y).$$

Prima di osservare Y, l'aspettazione su Y:

$$H(X \mid Y) = \sum_{y} P(Y = y) \cdot \sum_{x} -P(X = x \mid Y = y) \cdot \log_2 P(X = x \mid Y = y)$$

è chiamata *entropia condizionata* di X dato Y.

L'*information gain* per un test che determina il valore di Y è $H(X) - H(X \mid Y)$, ovvero il numero di bit, sempre positivo, risparmiato nel descrivere X noto il valore di Y.

La nozione di informazione è utile in diversi campi. Ad esempio, negli alberi di decisione, la teoria dell'informazione fornisce un criterio utile per scegliere la proprietà su cui splittare l'insieme degli esempi. Nell'apprendimento Bayesiano, la teoria dell'informazione permette di stabilire il miglior modello rispetto ai dati osservati.

8.2 Indipendenza Condizionata

Per gli assiomi di probabilità, date n variabili booleane, ci sono $2^n - 1$ valori da assegnare per dare una distribuzione di probabilità completa da cui possono

essere derivate probabilità condizionate. Per determinare qualsiasi probabilità, potrebbe essere necessario iniziare con un enorme database di probabilità.

Un modo utile per limitare la quantità di informazioni richieste è assumere che ogni variabile dipenda direttamente solo da poche altre variabili. Si costruisce, quindi, una ipotesi di *indipendenza condizionale*. In questo modo, non solo si riduce notevolmente il numero di dati necessari, ma anche, si rende più efficiente il ragionamento.

Una variabile stocastica X è *condizionatamente indipendente* da un'altra variabile stocastica Y dato l'insieme di variabili stocastiche Zs se

$$P(X \mid T, Zs) = P(X \mid Zs).$$

Sono invece *incodizionatamente indipendenti* se

$$P(X, Y) = P(X)P(Y)$$

ossia condizionatamente indipendenti ma senza osservazioni date.

8.3 Belief Network

La nozione di indipendenza condizionale è utilizzata per dare una rappresentazione coincisa a diversi domini. L'idea è che, data una variabile stocastica X, ci sono poche variabili che influenzano direttamente il valore di X, le quali costituiscono una *Markov blanket*.

Una *belief network* sfrutta proprio le Markov blanket per costruire un grafo di probabilità condizionate.

Per definire una belief network su un insieme di variabili stocastiche, $\{X_1, \ldots, X_n\}$, stabiliamo prima un ordinamento totale sulle variabili, ad esempio $X_1, \ldots, X_n$. Applichiamo la chain rule per decomporre una probabilità congiunta in un prodotto di probabilità condizionate:

$$P(X_1 = v_1 X_2 = v_2 \wedge \cdots \wedge X_n = v_n) = \prod_{i=1}^{n} P(X_i = v_i \mid X_1 = v_1 X_2 = v_2 \wedge \cdots \wedge X_{i-1} = v_{i-1}). \quad (8.2)$$

Oppure in termini di variabili stocastiche e distribuzioni di probabilità

$$P(X_1, X_2, \ldots, X_n) = \prod_{i=1}^{n} P(X_i \mid X_1 \cdots \wedge X_{i-1}).$$

Definiamo i *parents* di una variabile stocastica X_i, scritto $parents(X_i)$, come l'insieme minimale di predecessori di X_i nell'ordinamento totale tali per cui tutti gli altri predecessori di X_i sono condizionatamente indipendenti da X_i dato $parent(X_i)$. Quindi X_i dipende probabilisticamente dai suoi genitori ma è

indipendente da tutti gli altri predecessori. Questo significa che $parents(x_i) \subseteq \{X_1, \ldots, X_{i-1}\}$ tale che

$$P(X_i \mid X_1, \ldots, X_{i-1}) = P(X_i \mid parents(X_i)).$$

Unendo la chain rule con la definizione di genitore otteniamo:

$$P(X_1, X_2, \ldots, X_n) = \prod_{i=1}^{n} P(X_i \mid parents(X_i)).$$

La probabilità su tutte le variabili, $P(X_1, X_2, \ldots, X_n)$, è chiamata *distribuzione di probabilità congiunta*. Una rete bayesiana (o belief network) definisce una *fattorizzazione* della distribuzione di probabilità congiunta in un prodotto di probabilità condizionate. Graficamente, una rete bayesiana è un grafo aciclico diretto (DAG), dove i nodi sono le variabili stocastiche e gli archi orientati vanno dai nodi $parents(X_i)$ in X_i. Associato alla belief network vi è un insieme di distribuzioni di probabilità condizionata che specificano la probabilità condizionata di ogni variabile dati i suoi genitori.

Il compito più comune di *inferenza probabilistica* è quello di calcolare la *distribuzione a posteriori* di una o più variabile di query, data qualche evidenza, dove l'evidenza è una congiunzione di assegnazioni di valori ad alcune delle variabili.

8.3.1 Costruire una Belief Network

Per modellare un dominio attraverso una belief network, il progettista dovrebbe prendere in considerazione:

- Quali sono le variabili rilevanti
  - Cosa può essere osservato del dominio. Ogni feature che un agente può osservare nel dominio dovrebbe essere una variabile.
  - Per quali informazioni l'agente potrebbe voler conoscere la probabilità a posteriori. Ognuna di queste feature dovrebbe essere una potenziale variabile di query.
  - Controllare la presenza di eventuali variabili latenti che non vengono osservate o richieste, ma semplificano il modello.
- Quali valori dovrebbero popolare il dominio di queste variabili.
- Quali relazioni sussistono tra le variabili? Questo dovrebbe essere modellato mediante gli archi nel grafo che definiscono la relazione di *parent*.
- Come dovrebbe dipendere la probabilità di una variabile in base ai suoi *parent*. Questo viene espresso in termini di distribuzioni di probabilità condizionate.

8.4 Inferenza Probabilistica

Il compito più comune di inferenza probabilistica è quello di calcolare la distribuzione a posteriori di una variabile di query o variabili data qualche evidenza. Purtroppo, il problema di stimare una probabilità a priori o a posteriori in una belief network è NP-hard.

I principali approcci per l'inferenza probabilistica nelle belief network sono:

- *Inferenza esatta*: dove le probabilità sono calcolate esattamente.
- *Inferenza approssimata*: dove le probabilità sono solo approssimate. Questi metodi sono caratterizzati in base alle diverse garanzie che forniscono:
  - forniscono *limi garantiti* $[l, u]$ di variazione per la probabilità esatta p.
  - producono *limiti probabilistici* sull'errore prodotto, per esempio, garantendo un basso errore di predizione per una certa percentuale di volte. Potrebbero inoltre garantire che, nel tempo, la probabilità di predizione converga alla predizione esatta. La simulazione stocastica è un metodo che fornisce queste garanzie.
  - Potrebbero fare il possibile (best effort) per produrre un'approssimazione abbastanza buona, anche se ci possono essere casi in cui non funzionano bene. Una tale classe di tecniche è chiamata *inferenza variazionale*, dove l'idea è di trovare un'approssimazione al problema che sia facile da calcolare. Per prima cosa scegliete una classe di rappresentazioni facili da calcolare. Poi cerca di trovare un membro della classe che è più vicino al problema originale. Quindi, il problema si riduce ad un problema di ottimizzazione per la minimizzazione dell'errore, seguito da un semplice problema di inferenza.

8.4.1 Conditional Probability Tables (CPT)

Una tabella di probabilità condizionata (CPT) è una tabella che rappresenta la distribuzione di probabilità condizionata di una variabile stocastica, data una o più altre variabili stocastiche. Ogni riga della tabella rappresenta una possibile combinazione di valori per le variabili dipendenti e le voci in quella riga danno la probabilità che la variabile stocastica di interesse assuma ogni possibile valore, dati quei valori specifici delle variabili dipendenti. I CPT sono comunemente usati nelle reti bayesiane.

8.5 Modelli Probabilistici Sequenziali

Esistono particolari tipologie di belief network con strutture ripetute che vengono utilizzate per ragionare su concetti sequenziali come il tempo o le parole in una frase.

Fire	$P(smoke \mid Fire)$
true	0.9
false	0.01

Fire	*Tampering*	$P(alarm \mid Fire, Tampering)$
true	*true*	0.5
true	*false*	0.99
false	*true*	0.85
false	*false*	0.0001

X	Y	$P(Z = t \mid X, Y)$
t	t	0.1
t	f	0.2
f	t	0.4
f	f	0.3

Figura 8.1: Conditional probability tables.

8.5.1 Markov Chain

Una Markov Chain è una belief network con variabili aleatorie in sequenza, dove ogni variabile dipende solo dal suo predecessore nella sequenza. Dunque, l'indipendenza è definita come

$$P(S_{i+1} \mid S_0, \ldots, S_i) = P(S_{i+1} \mid S_i),$$

ed è detta *assunzione di Markov*. In una catena di Markov, ogni punto nella sequenza è detto *stato* e costituisce un *modello stazionario* se il dominio è unico per tutte le variabili e la probabilità di transizione è la stessa per ogni stato. Questo porta ad una *distribuzione stazionaria*, ovvero una distribuzione per la quale se la transizione da uno stato all'altro vale una volta, allora vale sempre.

Infine una catena di Markov si dice *ergordica* se per ogni coppia di stati s_1 ed s_2 vi è una probabilità non nulla di raggiungere s_2 da s_1. Si dice *periodica* con periodo p se la differenza tra le volte in cui si visita uno stesso stato è sempre divisibile per p. Se l'unico p possibile è 1, allora la Markov chain è *aperiodica*.

8.5.2 Modelli di Markov Nascosti

Un *hidden Markov model (HMM)* è un'espansione della Markov chain caratterizzata dall'aggiunta di stati per le osservazioni.

8.5.3 Belief Network Dinamica

Una *belief network dinamica (DBN)* è una belief network con una struttura regolare ripetuta nel tempo. È come una hidden Markov chain, ma gli stati e le osservazioni sono presentate in termini di feature.

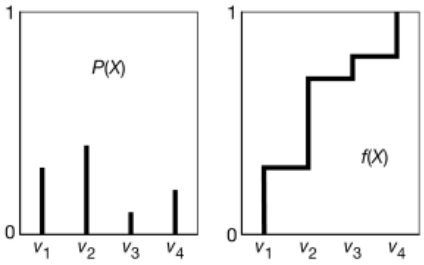


Figura 8.2: Una distribuzione di probabilità cumulativa.

8.6 Simulazione Stocastica

Molti problemi sono troppo grandi per pretendere un'inferenza esatta, quindi si deve ricorrere ad un'inferenza approssimata. Uno dei metodi più efficaci si basa sul generare campioni random dalla distribuzione a posteriori specificata dalla rete.

La *simulazione stocastica* si basa sull'idea che un insieme di campioni possa essere mappato su e da probabilità. Ad esempio, la probabilità $P(a) = 0.25$ significa che su 1000 campioni, circa un quarto avranno a vero. In questo modo, quindi, l'inferenza può essere effettuata passando da probabilità in campioni e da campioni in probabilità.

Esaminiamo come affrontare i seguenti problemi: come generare campioni, come inferire probabilità da campioni, e come incorporare osservazioni.

8.6.1 Campionare una Variabile

Il caso più semplice è quello di generare la distribuzione di probabilità per una singola variabile.

Dalle probabilità ai campioni

Per generare campioni da una singola variabile (discreta o reale), X, per prima cosa ordiniamo totalmente i valori nel dominio di X, l'ordine può essere arbitrario se non ve ne fosse uno naturale. Dato questo ordinamento, definiamo la *distribuzione di probabilità cumulativa* su x come $f(x) = P(X \leq x)$.

Per generare un campione casuale (o random sample) v per X, generiamo un numero casuale y da una distribuzione uniforme su $[0, 1]$. Ricaviamo $v \in dom(X)$ che abbia y come immagine nella distribuzione cumulativa, cioè $f(v) = y$.

Esempio 26. *Consideriamo una variabile stocastica X con dominio $\{v_1, v_2, v_3, v_4\}$. Supponiamo $P(X = v_1) = 0.3$, $P(X = v_2) = 0.4$, $P(X = v_3) = 0.1$, $P(X = v_4) = 0.2$. Quindi ordiniamo il dominio, ad esempio $v_1 < v_2 < v_3 < v_4$. La*

figura 8.2 mostra $P(X)$, la distribuzione per X, e $f(X)$, la distribuzione cumulativa per X. Consideriamo il valore v_1; lo 0.3 *del dominio di f riporta a v_1. Quindi, se un campionamento è uniformemente selezionato dall'asse Y, v_1 ha* 0.3 *chance di essere selezionato, v_2 ha lo* 0.4*, e così via.*

Dai campioni alle probabilità

Le probabilità possono essere stimate da un insieme di campioni usando la media campionaria. La *media campionaria* di una proposizione α è il numero di campioni in cui α è vera (s) diviso per il numero di totale dei campioni (n). Per la *legge dei grandi numeri*, la media campionaria converge alla probabilità esatta al crescere del numero di campioni considerati.

La stima dell'errore ε della media campionaria, dati gli n campioni, è fornita dalla *disuguaglianza di Hoeffding*

$$P(|s - p| > \varepsilon) \leq 2e^{-2n\varepsilon^2}.$$

8.6.2 forward Sampling su BN

Il *forwardsampling* è un modo per generare un campione di ogni variabile in una belief network così che ogni campione sia generato in proporzione alla sua probabilità. Questo ci permette di stimare la probabilità a priori di qualsiasi variabile.

Supponiamo che $X_1, \ldots, X_n$ sia un ordinamento totale sulle variabili in modo che ogni variabile sia preceduta dai suoi *parents*. Il forward sampling estrae un campione di tutte le variabili, estraendo un campione per ogni variabile $x_1, \ldots, X_n$ in ordine. Prima, campione X_1 usando la distribuzione cumulativa. Poi, per ciascuna delle altre variabili, quando arriva il turno di X_i, per via dell'ordinamento totale delle variabili, ha già i valori di tutti i genitori di X_i.

Quindi campiona un valore per X_i dalla distribuzione di X_i dati i valori già assegnati ai genitori di X_i. Ripetere questo per ogni variabile genera un campione contenente valori per tutte le variabili. La distribuzione di probabilità di una variabile query è stimata considerando la proporzione di campioni che sono stati assegnati a ogni valore della variabile.

8.6.3 Markov Chain Monte Carlo

Questo metodo prescinde dall'ordine di campionamento. L'idea dietro al metodo *Markov chain Monte Carlo (MCMC)* per generare campioni a partire da distribuzioni consiste nel costruire una catena di campioni con la distribuzione desiderata come sua distribuzione stazionaria e quindi campionare dalla Markov chain. Solitamente, in fase di *burn-in*, si scartano i primi campionamenti in quanto lontani dalla distribuzione stazionaria.

Un modo di creare una Markov chain a partire da una belief network con osservazioni è di usare il campionamento di Gibbs. L'idea è di bloccare le variabili osservate ai valori che sono stati osservati e campionare le altre variabili.

Ogni variabile è campionata dalla distribuzione della variabile dati i valori correnti delle altre variabili. Si noti che ogni variabile dipende solo dai valori delle variabili nella sua coperta Markov. La Markov blanket (o cperta di Markov) per una variabile X in una belief network contiene i genitori di X, i figli di X e gli altri genitori dei figli di X.

Capitolo 9

Apprendimento e Incertezza

9.1 Modelli di Apprendimento Probabilistico

Uno dei modi principali per scegliere un modello, dato un insieme di esempi Es, è scegliere quel modello m che massimizza la probabilità del modello dati gli esempi, $P(m \mid Es)$. Il modello che massimizza tale probabilità è detto *maximum a posteriori* model (MAP model).

La probabilità del modello m dati gli esempi Es si calcola con la regola di Bayes come segue

$$P(m \mid Es) = \frac{P(Es \mid m) \cdot P(m)}{P(Es)}.$$

La verosimiglianza $P(Es \mid m)$ indica quanto bene il modello di adatta agli esempi, un valore alto indica che il modello ha un buon *fit* con gli esempi, un valore basso indica che il modello m avrebbe predetto dati diversi. La probabilità a priori $P(m)$ codifica il *learning bias* e specifica quali modelli sono più probabili a priori, viene utilizzato per indurre a scegliere modelli più semplici. Il denominatore $P(Es)$, chiamata *funzione di partizione*, è una costante di normalizzazione che assicura una probabilità a domma 1. Il modello MAP massimizza il numeratore, trascurando il denominatore essendo indipendente da m.

Una alternativa consiste nello scegliere il modello *maximum likelihood model* (ML model), ovvero il modello che massimizza $P(Es \mid m)$, che equivale a scegliere il modello MAP con una distribuzione uniforme sulle ipotesi $P(m)$.

9.1.1 Apprendere Probabilità

Esempio 27. *Consideriamo il problema di predire il prossimo lancio di una puntina da disegno che ha una* testa *ed una* coda*.*

Supponiamo che la puntina venga lanciata diverse volte e le osservazioni in Es riportano n_0 volte testa *e n_1 volte* coda*. Assumiamo che i lanci siano indipendenti e che la* testa *occorra con una probabilità p.*

La verosimiglianza per questo caso è

$$P(Es \mid p) = p^{n_1} \cdot (1-p)^{n_0}$$

la cui log-likelihood è

$$\log P(Es \mid p) = n_1 \cdot \log p \cdot n_0 \cdot \log(1-p)$$

da cui, facilmente, notiamo che se uno tra n_0 ed n_1 fosse 0, risulterebbe con probabilità zero un evento comunque possibile.

Un semplice modo per risolvere il problema della probabilità zero e tenere in conto la conoscenza a priori è usare uno *pseudoconteggio* aggiunto ai dati di training.

Supponiamo che un agente debba predire un valore per Y con dominio $\{y_1, \ldots, y_k\}$. L'agente inizia con uno pseudoconteggio c_i per ogni y_i, stabilito prima che l'agente veda qualsiasi esempio. Inoltre, dal training set, il numero di "data points" con $Y = y_i$ risulta essere n_i. Allora la probabilità di Y è stimata usando:

$$P(Y = y_i) = \frac{c_i + n_i}{\sum_{i'} c_{i'} + n_{i'}}$$

Se non è disponibile conoscenza a priori, Laplace ci consiglia di utilizzare uno pseudoconteggio $c_i = 1$. Questo metodo con conteggio a priori di 1 è detto *aggiustamento di Laplace*.

Opinione degli Esperti

Lo pseudoconteggio ci fornisce anche un modo per combinare i dati all'opinione degli esperti. Spesso un agente potrebbe non avere molti dati a disposizione, ma potrebbe avere accesso a diversi esperti che in base alla propria esperienza possono fornire differenti distribuzioni di probabilità.

Ci sono, però, diversi problemi nell'ottenere probabilità dagli esperti:

- Riluttanza nel fornire valori esatti di probabilità che non possano essere modificati in seguito.
- Diversi valori possono provenire da diversi esperti.

Per porre rimedio a questo problema, invece di fornire un numero reale per quantificare una probabilità per A, un esperto potrebbe fornire una coppia di valori $< n, m >$ intesi come n occorrenze di A su m prove. In questo modo, l'esperto non solo fornisce una probabilità, ma anche una stima delle dimensioni dei dati su cui fonda il parere. Inoltre, mentre la proporzione tra i due valori fornisce una stima della probabilità, i valori assoluti forniscono una stima del grado di *confidenza* dell'esperto. La confidenza aumenta all'aumentare dei valori assoluti.

9.1.2 Classificatore Probabilistico

Un *Bayes classifier* è un modello probabilistico usato per l'apprendimento supervisionato. Questo modello si basa sull'idea che una *classe* aiuta a predire i valori delle feature per i suoi membri. In effetti gli esempi vengono classificati in classi perché hanno valori comuni per alcune feature. Tali classi sono chiamate *natural kinds*. Quindi, il modello impara la dipendenza delle feature dalla classe e si usa il modello risultante per predire la classificazione di nuovi esempi.

Il caso più semplice di classificatore probabilistico è il *Naive Bayes*, che fonda il suo ragionamento sull'assunzione di indipendenza, assume, cioè, che le feature di input siano condizionatamente indipendenti le une dalle altre. L'indipendenza del naive Bayes è incarnata in una belief network in cui le feature sono i nodi, le feature target (la classificazione) non hanno genitori e le target feature sono i soli genitori delle feature di input. Questa belief network richiede le distribuzioni di probabilità $P(Y)$ per la feature target Y (o classe Y) e $P(X_i \mid Y)$ per ogni feature di input X_i.

Diversamente da molti altri modelli di apprendimento supervisionato, il classificatore naive Bayes può maneggiare data mancanti dove non tutte le feature sono osservate. Chiaramente, la condizione *ottimale* per questo modello sarebbe avere una sola feature di input X_i, così da evitare l'assunzione di indipendenza. All'aumentare delle X_i l'accuratezza dipende dalla reciproca indipendenza delle X_i rispetto alla Y data.

Apprendimento di Classificatori Bayesiani

Per apprendere un classificatore, le distribuzioni $P(Y)$ e $P(X_i \mid Y)$ possono essere apprese dai dati. Ogni distribuzione di probabilità condizionata $P(X_i \mid Y)$ può essere trattato come un diverso sotto-problema per ogni valore di Y.

Nel caso più semplice si stima la maximum likelihood $P(X_i = x_i \mid Y = y)$ come proporzione empirica osservata nei dati di training, ovvero come il rapporto tra il numero di casi in cui $X_i = x_i \wedge Y = y$ ed il numero di casi in cui $Y = y$.

Anche in questo caso, incorporare lo pseudoconteggio risolve il problema di probabilità nulle che renderebbero probabilisticamente impossibili determinate combinazioni di osservazioni. Un'alternativa sarebbe utilizzare frequenze empiriche.

Il naive Bayes può essere esteso per permettere ad alcune feature di input di essere genitori delle feature target e ad altre di essere figlie delle feature target. I figli della classificazione non devono essere necessariamente indipendenti. Una rappresentazione dei figli è la *Tree Augmented Naive Bayes (TAN)*, dove ai figli è permesso di avere esattamente un altro genitore oltre la classificazione. Questo permette un modello semplice che tiene conto delle interdipendenze tra i figli. Un'alternativa è mettere la struttura nella variabile di classe. Un modello ad *Albero Latente* decompone la variabile di classe in un numero di variabili latenti collegate insieme in una struttura ad albero. Ogni variabile osservata è figlia

di una delle variabili latenti. Le variabili latenti permettono un modello della dipendenza tra le variabili osservate.

9.1.3 Apprendimetno MAP di Alberi di Decisione

Per l'apprendimento di alberi di decisione serve un bias, che tipicamente favorisce alberi più brevi ed è imposto attraverso la distribuzione a priori sui modelli.

Se non ci sono esempi che abbiamo gli stessi valori nelle feature di input, ma valori diversi nelle target feature, diversi alberi di decisione possono adattarsi perfettamente ai dati. In caso di *rumore* nei dati però, dove il rumore è dato da esempi con X_i uguali ma valori di Y diversi, o da valori mancanti per alcune X_i, nessuno degli alberi che si adatta perfettamente potrà risultare ottimale, rendendo conveniente scegliere anche tra quelli che si adattano meno perfettamente. L'apprendimento MAP fornisce un modo per comparare tali modelli. Ricordando che la probabilità a priori fornisce un bias (learning bias) in base al quale preferire un modello rispetto ad un altro, tendenzialmente il più semplice, vediamo come il teorema di Bayes fornisce un modo per contemperare semplicità e abilità nel trattare il rumore.

Gli alberi di decisione gestiscono il rumore tramite probabilità alle loro foglie. In presenza di dati rumorosi, alberi più grandi si adattano meglio in quanto riescono a seguire meglio le irregolarità causate dal rumore. Quindi, se la probabilità a priori favorisce alberi più semplici, la verosimiglianza favorisce alberi grandi perché adattandosi meglio ai dati avranno una likelihood maggiore. Allora utilizziamo Bayes per ottenere una distribuzione a posteriori del modello proporzionale al loro prodotto:

$$P(m \mid Es) \propto P(Es \mid m) \cdot P(m) \tag{9.1}$$

9.1.4 Lunghezza delle Descrizioni

Applicando i logaritmi alla formula 9.1 del MAP model, si ottiene

$$(-\log_2 P(Es \mid m)) + (-\log_2 P(m))$$

Questo risultato può essere interpretato in termini di teoria dell'informazione. Il primo termine rappresenta il numero di bit per descrivere i dati (Es), dato il modello m, il secondo termine rappresenta il numero di bit per descrivere (m). Un modello che minimizza questa somma è il *minimum description lenght (MDL)* model. Il principio alla base del modello MDL consiste nello scegliere il modello che trasmette i dati ed il modello nella maniera più breve possibile.

Poiché descrivere la complessità di un codice in termini di bit non è semplice, è spesso preferibile ricorrere a misure approssimate per la complessità. Un modo per farlo è considerare i suoi parametri probabilistici. Dato $|m|$ il numero di parametri (numero delle foglie per gli alberi e numero di parametri numerici per una funzione lineare o una rete neurale) e $|Es|$ il numero di esempi di training.

Il problema di trovare il modello MDL può essere approssimato a minimizzare la quantità

$$-\log_2 P(Es \mid m) + |m| \cdot \log_2 P(|Es|).$$

Questo valore è l'indice *BIC* (Bayesian Information Criteria).

9.2 Apprendimento non Supervisionato

Nell'*apprendimento non supervisionato* nel training set non sono presenti le feature target. L'obiettivo è costruire una classificazione naturale dei dati.

Un metodo generale per l'apprendimento non supervisionato è il *clustering*, che partiziona gli esempi in *cluster* o *classi*. Ogni classe predice i valori delle feature per gli esempi che contiene. Ogni clustering (sistema di cluster o raggruppamento) presenta un certo *errore di predizione* associato. Il clustering migliore è quello che minimizza l'errore.

Nell'*hard clustering*, ogni esempio viene assegnato ad una classe precisa e la classe può essere utilizzata per predire i valori delle feature di quell'esempio. L'alternativa all'hard clustering è il *soft clustering*, in cui ogni esempio ha una distribuzione di probabilità sulle classi. In questo caso la predizione dei valori delle feature di un esempio è data dalla media ponderata delle previsioni delle classi in cui si trova l'esempio, ponderata dalla probabilità che l'esempio sia nella classe.

9.2.1 k-Means

L'algoritmo *k-Means* è utilizzato per l'hard clustering. In input all'algoritmo sono dati gli esempi ed il numero di classi k. Quindi costruisce k classi, una predizione del valore di ogni feature per ogni classe ed una funzione di assegnazione degli esempi alle classi.

Formalmente, supponiamo che Es sia l'insieme degli esempi di training e siano $X_1, \ldots, X_n$ le feature di input. Sia $X_j(e)$ il valore della feature di input X_j per l'esempio e. Identifichiamo ogni classe con un intero $c \in 1, \ldots, k$.

L'algoritmo costruisce:

- una funzione $class : Es \to \{1, \ldots, k\}$ che associa ad ogni classe un esempio. Se $class(e) = c$, diremo che l'esempio e è nella classe c.
- per ogni feature X_j, una funzione $\widehat{X}_j$, che va dalle classi al dominio di X_j, tale che, per ogni feature, $\widehat{X}_j(c)$ è la predizione del valore della feature X_j per la classe c, cioè per ogni esempio che contiene. Quindi per un esempio e, $\widehat{X}_j(class(e))$ è il valore predetto per la sua feature X_j.

  Il k-means deve trovare $class$ e $\widehat{X}_j$ che minimizzino l'errore quadratico definito come

$$\sum_{e \in Es} \sum_{j=1}^{n} (\widehat{X}_j(classe(e)) - X_j(e))^2.$$

Per minimizzare questa quantità, la predizione di una classe dovrebbe essere la media delle predizioni dei suoi esempi. Se gli esempi sono molti, ricercare la migliore assegnazione (di ogni esempio ad ogni classe) tra tutte quelle possibili è infattibile. Quindi il k-means "si limita" a migliorare iterativamente l'errore quadratico partendo da un'assegnazione casuale degli esempi alle classi e ripetendo i seguenti due passi:

1. Per ogni classe c ed ogni feature X_j, assegna ad $\widehat{X}_j(c)$ il valore medio di $X_j(e)$ per ogni e in c:
$$\widehat{X}_j(c) \leftarrow \frac{\sum_{e:class(e)=c} X_j(e)}{|\{e : class(e) = c\}|}$$
dove il denominatore conta il numero di esempi in c.
2. Riassegna ogni esempio ad una classe assegnando ogni esempio e alla classe c ceh minimizza
$$\sum_{j=1}^{n} (\widehat{X}_j(c) - X_j(e))^2$$

Questi due passi vengono ripetuti fino a quando non avvengono riassegnazioni (si raggiunge la stabilità). Solitamente converge in poche iterate.

L'inizializzazione random potrebbe avvenire in diversi modi:

- attraverso l'assegnazione casuale di ogni esempio ad una classe,
- selezionando k esempi a caso che rappresentino le classi,
- o assegnando solo alcuni esempi per costruire le *statistiche sufficienti* iniziali.

Gli ultimi due metodi possono essere più utili se il set di dati è grande, in quanto evitano di assegnare l'intero set di dati per l'inizializzazione.

Un'assegnazione è *stabile* se un iterazione di k-means non modifica le assegnazioni. Alcune assegnazioni potrebbero essere migliori di altre, in termini di errore quadratico. Per trovare l'assegnazione migliore è necessario applicare un algoritmo di ricerca che ci permetta di confrontare le assegnazioni stabili per trovare un minimo locale. A tal proposito è spesso utile provare configurazioni iniziali diverse applicando il *random restart*, finendo per scegliere l'assegnazione stabile con l'errore minimo.

Un problema che affligge il k-means è al sua sensibilità alla scala delle dimensioni. Feature su scale diverse potrebbero sbilanciare il confronto (si pensi alle diverse scale di misura di età, altezza, genere). È necessaria una fase di normalizzazione.

Un altro dettaglio da curare in questo algoritmo è la scelta nel numero k di classi. Un modo per trovare il suo valore naturale consiste nel partire da un valore basso ed incrementarlo man mano fino a quando si ottiene una riduzione di errore significativa rispetto al precedente valore per k, ossia in corrispondenza del *gomito* della curva sul grafico che plotta l'errore quadratico con il numero di clusters (esempio in Figura 9.1).

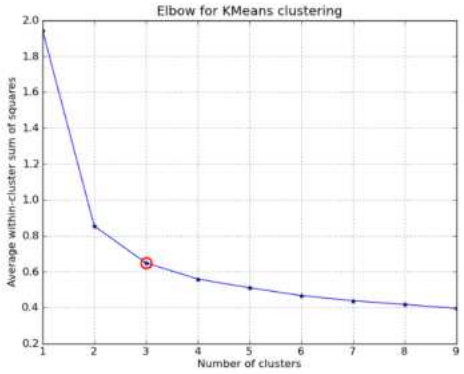


Figura 9.1: Esempio di k ottimale con il metodo del gomito.

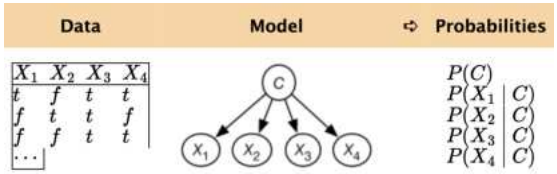


Figura 9.2: Algoritmo EM: classificatore di Bayes con classe nascosta.

9.2.2 Expectation Maximization e Soft Clustering

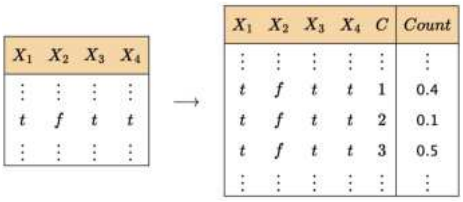
Una *variaible nascosca (latente)* è una variabile che non viene osservata nei dati. Un classificatore Bayesiano, nell'ambito dell'apprendimento non supervisionato, può trattare le classi come variabili latenti. L'algoritmo della *Expectation Maximization* o *EM algorithm* può essere utilizzato per apprendere modelli probabilistici con variabili latenti. Combinato con il classificatore naive Bayes può, quindi, essere utilizzato per fare soft-clustering, similmente al k-means, ma con esempi probabilisticamente assegnati alle classi. Come nel k-means in input è dato il numero k di classi e gli esempi Es.

Dati gli esempi, l'algoritmo costruisce un modello naive Bayes in cui vi è una variabile per ogni feature di input e una variabile nascosta per la classe. La variabile di classe è l'unico genitore di tutte le feature, come mostrato in Figura 9.2. La variabile di classe ha dominio $\{1, \ldots, k\}$ dove k è il numero di classi.

Le probabilità su cui questo modello si basa per funzionare sono la probabilità a priori della classe $P(C)$ e la probabilità condizionata di ogni feature data la classe $P(X_i \mid C)$. L'obiettivo di questo algoritmo è apprendere le distribuzioni di probabilità che meglio si adattano ai dati.

L'algoritmo EM concettualmente è come se estendesse la tabella dei dati con due colonne virtuali, la colonna classe C, in cui si inserisce il numero della classe di appartenenza dell'esempio, e una colonna *count* di contatori normaliz-

zati. Ogni esempio originale viene replicato k volte quante sono le classi in k righe diverse. Per ogni replica dello stesso esempio, nella colonna *count*, viene assegnato un valore, la cui somma con i valori delle altre repliche sia 1. Per esempio, per quattro feature e tre classi, avremmo L'algoritmo EM parte con



counts randomici e ripete due step:

1. *E step*: aggiorna i conteggi in *count* in base alle distribuzioni correnti de modello. Per ogni esempio $< X_1 = v_1, \ldots, X_n = v_n >$ nei dati originali, il contatore normalizzato associato a $< X_1 = v_1, \ldots, X_n = v_n, C = c >$ nei dati estesi viene aggiornato, $\forall c \in 1, \ldots, k$, assegnandogli il valore $P(C = c \mid X_1 = v_1, \ldots, X_n = v_n)$.

   Questo passo è di *expectation* perché calcola il valore atteso.

2. *M step*: inferisce la probabilità per modello dalla tabella estesa. Poiché i dati estesi hanno valori associati ad ogni variabile, si tratta dello stesso problema di apprendere le probabilità dai dati per un classificatore naive Bayes. Questo è uno step di *maximization* perché si massimizza o la maximum likelihood (ML) o la maximum a posteriori probability (MAP) per la stima delle probabilità.

EM converge a un massimo locale e non ha bisogno di memorizzare i dati estesi, ma mantiene un insieme di *statistiche sufficienti* che forniscono abbastanza informazione per calcolare le probabilità richieste. Queste statistiche sufficienti sono:

- cc, il class count, ovvero un vettore k-dimensionale tale che $cc[c]$ sia la somma dei contatori degli esempi nei dati estesi con $class = c$.
- fc, feature counter, una matrice a tre dimensioni tale che $fc[i, v, c]$ per i da 1 a n, per ogni valore $v \in domain(X_i)$, e per ogni classe c, sia la somma dei conteggi per l'esempio esteso t, con $X_i(t) = val$ e $class(t) = c$.

Le probabilità richieste per il modello possono quindi essere calcolate da cc ed fc:

$$P(C = c) = \frac{cc[c]}{|Es|}$$

$$P(X_i = v \mid C = c) = \frac{fc[i, v, c]}{cc[c]}.$$

L'algoritmo può fermarsi quando cc e fc cambiano poco tra un'iterata e l'altra. In alternativa si potrebbe prevedere un numero massimo di iterate.

9.3 Apprendimento di Belief Network

Una belief network dà una distribuzione di probabilità su un insieme di variabili casuali. Non possiamo sempre aspettarci che un esperto sia in grado di fornire un modello accurato; spesso vogliamo imparare una belief network dai dati.

Imparare una belief network dai dati si può fare in diversi modi in base alla conoscenza a priori conosciuta e a quanto sia completo il dataset. Nel caso più semplice, viene data la struttura, tutte le variabili sono osservate in ogni esempio, e solo le probabilità condizionali di ogni variabile data ai suoi genitori devono essere apprese. All'altro estremo, l'agente può non conoscere la struttura o anche quali variabili esistono, e ci possono essere dati mancanti, che non possono essere considerati mancanti a caso.

Nel caso più semplice la struttura è nota, cioè tutte le variabili sono osservate in ogni esempio, e l'unico obiettivo è calcolare le distribuzioni condizionate di ogni variabile dati i genitori. Dall'altro estremo, la struttura delle variabili non è nota o non si sa quali variabili esistano, e i dati sono incompleti.

9.3.1 Apprendere le Probabilità

Per il caso più semplice, stabilire $P(X_i \mid parents(X_i))$ per ogni X_i è un problema di apprendimento supervisionato. Si potrebbe utilizzare un albero di decisione per variabili discrete o regressione logistica e reti neurali per distribuzione di variabili binarie.

9.3.2 Variabili Latenti

Il caso un pò più difficile è quando il modello è noto, ma non tutte le variabili sono osservate, vi sono variabili latenti.

In questo caso, l'algoritmo EM per l'apprendimento di una rete bayesiana con variabili nascoste è essenzialmente la stessa dell'algoritmo EM per il clustering. Nel passo E si utilizza l'inferenza probabilistica, per ogni esempio, per derivare la distribuzione di probabilità della variabili nascoste date le variabili osservate per quell'esempio. Il passo M deriva le probabilità del modello dai dati estesi (come visto nella sezione precedente, ma con conteggi non necessariamente interi).

9.3.3 Dati Mancanti

Oltre ai dati latenti, i dati possono essere incompleti anche per altre cause. In presenza di dati mancanti il dataset va usato con cautela poiché la loro mancanza potrebbe essere correlata al fenomeno di interesse. Ad esempio, una terapia potrebbe sembrare funzionante solo perché fa ammalare i pazienti su cui non funziona facendoli allontanare dalla sperimentazione ed escludendoli dai risultati.

I dati sono mancanti in modo casuale (*missing at random*) quando il motivo per cui mancano non è correlata ad alcuna variabile del modello. Questo tipo di mancanza può essere ignorata o completata con algoritmo come EM, ma è

una forte assunzione. Sarebbe preferibile un modello che spieghi la mancanza dei dati o che capisca il motivo della mancanza dal mondo reale.

9.3.4 Apprendimento della Struttura

Supponiamo che un agente disponga di dati completi e non vi sono variabili nascoste, ma la struttura della della belief network non è disponibile. Per *apprendere la struttura* si possono applicare tue metodi:

- Il primo metodo consiste nell'applicare la definizione di belief network. Quindi dato un ordinamento totale delle variabili, determinare i sottoinsiemi *parents* per codificare le relazioni di indipendenza condizionata. Ma questo comporta due complicazioni

  1. determinare il miglior ordinamento
  2. misurare l'indipendenza

- Il secondo metodo consiste in una ricerca basata su uno *score* per le belief network, ad esempio utilizzando il MAP model, che prende in considerazione l'adattamento ai dati e la complessità del modello.

Presentiamo, quindi, un metodo appartenente al secondo metodo, chiamato *search and score*. Aassumendo la disponibilità di un dataset di esempi Es completo, l'obiettivo del metodo search and score è trovare un modello che massimizzi

$$P(m \mid Es) \propto P(Es \mid m) \cdot P(m).$$

dove la likelihood $p(Es \mid m)$ è il prodotto delle probabilità per ogni esempio, ulteriormente fattorizzabili in ragione delle distribuzioni di ogni X_i dati i genitori di X_i nel modello m, cioè $parents(X_i, m)$.

Questa considerazione ci conduce, trovato un buon ordinamento totale sulle variabili attraverso ricerca locale o branch-and-bound, alla risoluzione di due problemi supervisionati da risolvere:

- determinare le distribuzioni $P^e_m(X_i \mid parents(X_i, m))$
- determinare $\log P(m(X_i))$ approssimabile come nel BIC.

9.3.5 Caso Generale di Apprendimento di una Belief Network

Il caso generale prevede una struttura del modello sconosciuta, variabili latenti e dati mancanti. Per il problema dei dati mancanti è stato discusso precedentemente. Il secondo problema è di carattere computazionale, è proibitivo provare tutte le combinazioni possibili di ordinamenti e variabili latenti. Per risolvere, si potrebbe selezionare il miglior modello secondo il criterio MAP oppure facendo una media su tutti i modelli.