

# **Appunti del corso “Introduzione all’intelligenza artificiale”.**

**AA 2017/18**

**Luca Corbucci**

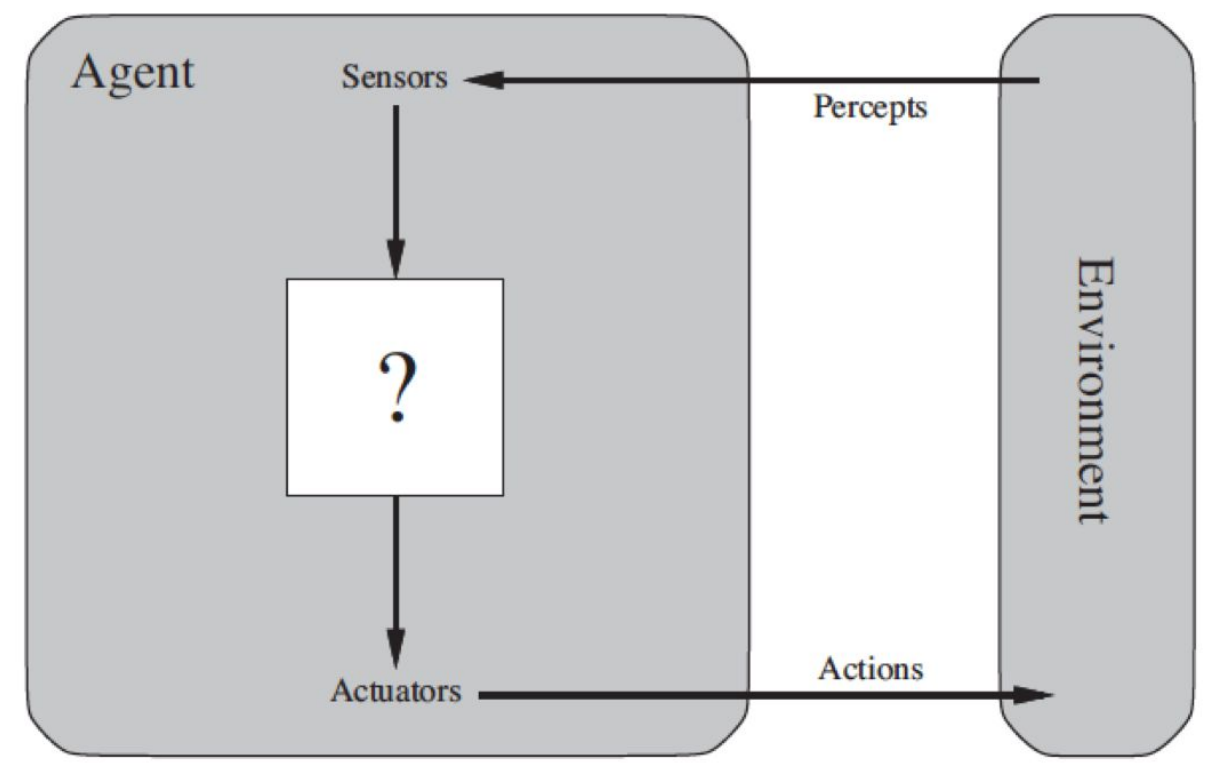
# Prima Lezione

## Agenti Intelligenti

L'attuale approccio all'intelligenza artificiale prevede la costruzione di agenti intelligenti ovvero agenti che devono risolvere dei problemi con una ricerca in uno spazio di stati.

Questi agenti intelligenti devono avere le seguenti caratteristiche:

- Devono avere dei **sensori** in modo da ricevere delle percezioni dal mondo che li circonda
- Devono essere in grado di svolgere delle **azioni** sull'ambiente utilizzando degli **attuatori**
- Gli agenti devono essere in grado di collaborare tra loro e di difendersi da altri agenti



La percezione che si ottiene tramite i sensori è particolarmente importante perché a partire dalle varie percezioni dell'agente si riesce a creare una **sequenza percettiva** in cui memorizziamo tutte le **percezioni** dell'agente.

Quando devo decidere cosa fare devo basarmi sulla sequenza percettiva:

$f(\text{Sequenza percettiva}) \rightarrow \text{Azione}$

La  $f$  che utilizziamo per generare l'azione è la **funzione agente** che va implementata e che caratterizza il funzionamento dell'agente.

Gli agenti che possiamo creare possiamo suddividerli in due macro categorie:

- **Agenti razionali:** è l'agente che interagisce con l'ambiente facendo la cosa giusta. Per capire quanto sia giusto quello che viene fatto mi serve un criterio di valutazione che mi giudica l'effetto delle azioni sull'ambiente. Per dare un giudizio posso basarmi sull'evoluzione del mondo o sull'effetto che voglio ottenere.

La razionalità dipende dalle percezioni e dalla misura di prestazioni, in particolare un agente razionale, per ogni sequenza di percezioni, andrà a scegliere l'azione che **massimizza la misura di prestazione**.

- **Agenti autonomi:** dato che non sempre l'agente può avere fin da subito una conoscenza completa del mondo, si prevedono anche degli agenti che sono **autonomi** perché **imparano da soli basandosi sull'esperienza** e sulle percezioni che ottengono dall'ambiente.

Problema di IA, lo descrivo tramite PEAS:

- P come Performance
- E come Environment

- A come attuatori
- S come sensori

Il problema è caratterizzato da un ambiente (Environment) che ha varie proprietà e caratteristiche che possono influenzare la soluzione.

L'ambiente viene in particolare suddiviso tramite i seguenti attributi:

- **Osservabilità:** un certo ambiente può essere **completamente osservabile** o **parzialmente osservabile**. Una completa osservabilità ci garantisce una conoscenza completa dell'ambiente senza dover salvare uno stato interno. Una osservabilità parziale mi obbliga a salvare uno stato in modo da avere sempre in mente l'ambiente che mi circonda.
- **Quanti agenti ci sono nell'ambiente:** un ambiente può avere al suo interno **un solo agente** oppure **molti agenti**, questi possono essere in competizione tra loro (un gioco) oppure possono cooperare tra loro per raggiungere un obiettivo comune
- **Predicibile:** l'ambiente può essere **deterministico**, in questo caso il prossimo stato lo determino basandomi sullo stato corrente oppure può essere **non deterministico** quando non ci sono legami tra lo stato attuale e il successivo. Un'altra possibilità è lo **stato stocastico** in cui abbiamo lo stato successivo che dipende dalla probabilità
- **Episodico o sequenziale:** l'esperienza può essere divisa in episodi atomici che sono indipendenti tra loro o in episodi sequenziali in cui ogni episodio influenza il successivo.
- **Statico/Dinamico:** nel caso di un ambiente statico, questo rimane sempre uguale per tutto il tempo, il dinamico invece varia. Abbiamo anche il caso di un ambiente semi-dinamico in cui non cambia l'ambiente ma cambia la valutazione dell'agente.
- **Discreto/Continuo:** il numero degli stati nel caso discreto rimane lo stesso per tutto il tempo, nel caso continuo invece il numero di stati viene modificato.
- **Nota/Ignoto:** abbiamo una conoscenza completa dell'ambiente o non conosciamo l'ambiente.

## Agente

L'agente è quello che deve essere costruito per risolvere un problema di AI, questo è formato da:

$$\text{Agente} = \text{Architettura} + \text{Programma}$$

Dove il programma è l'implementazione di una funzione che partendo dalle percezioni mi permette di ottenere delle azioni:

$$\text{Programma} = \text{Percezioni} \rightarrow \text{Azioni}$$

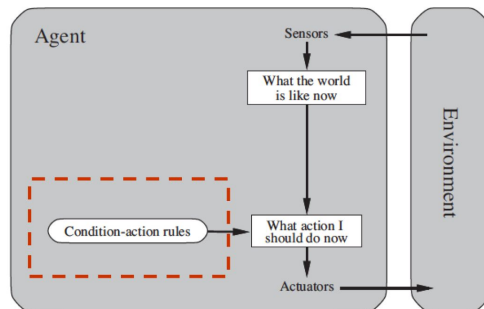
Le azioni potrebbero essere memorizzate in una tabella ma dato che diventerebbe ingestibile bisogna trovare un modo più semplice per fare in modo che da una percezione si possa generare una azione.

Un esempio di programma di un agente che partendo dalle percezioni genera una azione è il seguente (viene mantenuta una memoria con le percezioni e poi viene scelta una azione da eseguire, poi si aggiorna di nuovo la memoria):

```
function Skeleton-Agent (percept) returns action
  static: memory, the agent's memory of the world
  memory ← UpdateMemory(memory, percept)
  action ← Choose-Best-Action(memory)
  memory ← UpdateMemory(memory, action)
  return action
```

## Vari tipi di agenti:

- **Agenti reattivi semplici:**



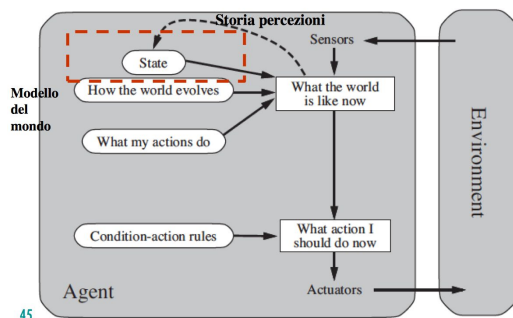
```
function Agente-Reattivo-Semplice (percezione)
  returns azione

  persistent: regole, un insieme di regole
  condizione-azione (if-then)

  stato ← Interpreta-Input(percezione)
  regola ← Regola-Corrispondente(stato, regole)
  azione ← regola.Azione
  return azione
```

Sono gli agenti che si basano unicamente sulla percezione che arriva dall'ambiente per capire quale azione eseguire. La percezione viene utilizzata per generare uno stato (che non viene memorizzato) che viene utilizzato per trovare una regola all'interno di un insieme di <regole, condizioni>. La regola è l'azione da svolgere.

- **Agenti reattivi basati sul modello:**



```
function Agente-Basato-su-Modello (percezione)
  returns azione

  persistent: stato, una descrizione dello stato corrente
              modello, conoscenza del mondo
              regole, un insieme di regole condizione-azione
              azione, l'azione più recente

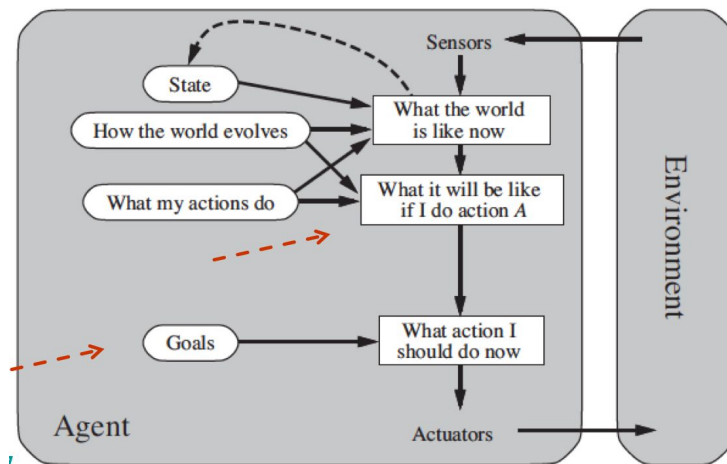
  stato ← Aggiorna-Stato(stato, azione, percezione,
                        modello)
  regola ← Regola-Corrispondente(stato, regole)
  azione ← regola.Azione
  return azione
```

In questo caso viene aggiunto uno stato interno all'agente, questo infatti prende le percezioni dall'esterno tramite i sensori e va subito ad aggiornare uno stato.

All'interno dell'agente poi salviamo una rappresentazione del mondo esterno.

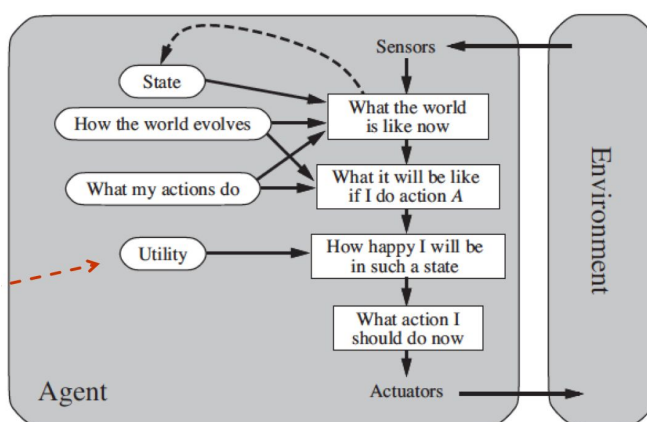
Abbiamo poi la lista di <condizioni, regole> che viene utilizzata per trovare una azione da svolgere in base allo stato attuale.

- **Agenti con obiettivo:**



In questo caso non abbiamo una lista di <condizioni, regole> da seguire per trovare l'azione da svolgere. Si utilizza l'obiettivo che l'agente si prefissa di raggiungere. Per ogni percezione che arriva dall'esterno viene aggiornato lo stato interno dell'agente e si vanno a considerare i possibili cambiamenti che vengono apportati al mondo in base alle varie azioni che posso scegliere. Alla fine viene scelta l'azione migliore per l'obiettivo che ho scelto.

- **Agenti con valutazione di utilità:**

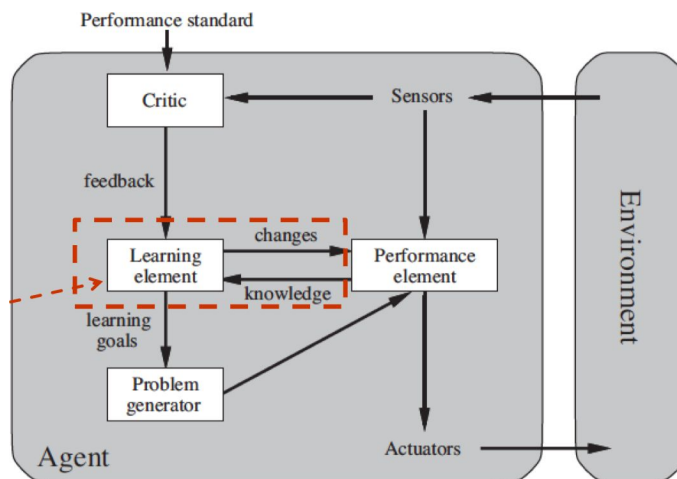


In questo caso non si utilizza l'obiettivo per capire quale azione scegliere ma si utilizza una funzione di utilità. Per ogni percezione agguirno lo stato interno e considero le azioni che posso scegliere

a partire da quello stato. La funzione di utilità mi indica un valore numerico e io scelgo l'azione migliore in base alla funzione di utilità.

La funzione di utilità è la funzione di valutazione di un agente razionale e quindi va massimizzata.

- **Agenti che apprendono:**



All'interno di un agente che apprende ci sono vari componenti:

- Il **Performance element** o elemento esecutivo è il programma vero e proprio dell'agente
- Abbiamo il **componente di apprendimento** che si occupa di cambiare l'elemento esecutivo e migliora se stesso grazie alle percezioni
- L'**elemento critico** si occupa di fornire dei feedback
- Il **generatore di problemi** invece suggerisce le nuove situazioni da esplorare.



# Seconda Lezione

## Agenti risolutori di problemi

In questo caso andiamo a considerare degli agenti che per risolvere i problemi effettuano una ricerca all'interno dello spazio degli stati, prima di agire cercando di determinare l'intera sequenza di azioni che poi verrà eseguita. Questi agenti hanno un obiettivo che li spinge a trovare una soluzione al problema.

Per fare questo è necessario conoscere:

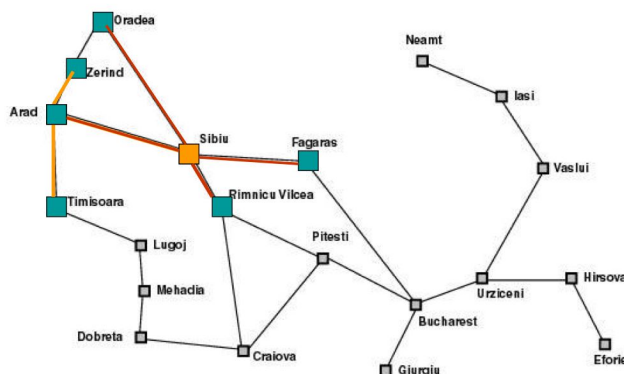
- L'obiettivo con il test obiettivo che controlla se mi trovo nello stato obiettivo
- Il problema con i suoi stati e le azioni possibili che possono essere svolte, queste azioni devono essere un numero finito e devono essere deterministiche
- L'ambiente in cui si svolge il problema deve essere statico, devo poterlo osservare.
- Lo stato iniziale
- Lo spazio degli stati, ovvero l'insieme di tutti gli stati possibili rappresentati dai nodi della rete
- Il modello di transizione ovvero una associazione tra i vari stati e le azioni che mi restituiscono un nuovo stato in cui mi muovo
- I vari cammini che trovo hanno un costo, per passare da uno stato al successivo ho un costo che non è mai negativo

## Dimostrare teoremi

Per arrivare ad una soluzione che mi porta in uno stato obiettivo possiamo utilizzare una dimostrazione formale, in questo caso abbiamo a disposizione delle premesse e vogliamo arrivare a dimostrare una proposizione, per farlo si sfrutta il modus ponens (se vale  $p$  e  $p \implies q$  allora vale  $q$ ).

## Algoritmi di ricerca

Per partire da uno stato iniziale e arrivare allo stato obiettivo è necessario avere a disposizione un algoritmo di ricerca che mi restituiscono un cammino soluzione che mi porta nello stato obiettivo. Trovare un cammino ha un costo che è uguale a costo della ricerca dell'algoritmo + costo del cammino stesso.



La ricerca viene fatta all'interno dello spazio degli stati utilizzando un albero di ricerca (o un grafo), all'interno dello spazio degli stati abbiamo degli stati che sono unici, il concetto di stato è differente dal concetto di nodo durante la ricerca. Il nodo dell'albero fa infatti riferimento ad uno stato dello spazio degli stati, più nodi all'interno dell'albero di ricerca possono fare riferimento allo stesso stato.

L'albero di ricerca si sovrappone allo spazio degli stati e ogni nodo viene espanso per ricercare la soluzione.

Quello che differenzia i vari algoritmi per la ricerca di una soluzione in uno spazio degli stati è il tipo della coda che viene utilizzata per gestire la frontiera, ovvero l'insieme dei nodi espansi, abbiamo infatti vari metodi di implementazione della coda:

- FIFO: il primo elemento inserito viene eliminato dalla coda, questa viene usata con la BF (Breadth First)
- LIFO: l'ultimo elemento aggiunto viene estratto dalla coda, viene usata nella DF (Depth First)

- Coda con priorità: nella coda i vari elementi hanno una priorità e quindi sono ordinati in base a questa priorità, viene utilizzata con Uniform Cost

Ogni nodo dello spazio degli stati ha i seguenti attributi:

- Lo stato
- Il nodo padre
- L'azione che lo ha generato
- Il costo del cammino per arrivare dal nodo iniziale a quel nodo, questo costo viene espresso come  $g(n)$

La ricerca di una soluzione può essere effettuata sia utilizzando la ricerca su alberi sia la ricerca su grafi. In particolare nella ricerca su alberi è possibile avere dei loop perchè posso avere una ripetizione dei nodi all'interno di un percorso.

Allora possiamo utilizzare un grafo per rappresentare la ricerca di una soluzione.

Ci sono tre soluzioni per cercare di migliorare la ricerca di una soluzione evitando la ripetizione di nodi già visti:

- Eliminare il padre dai nodi successori di un nodo
- Controllare se tra i successori di un nodo c'è un nodo che è predecessore del nodo stesso
- Fare una lista con i nodi esplorati, è la soluzione più costosa perchè devo mantenere più nodi in memoria di quanti ne dovrei mantenere. Ad esempio con la ricerca in profondità ho una occupazione di memoria che è pari al fattore di diramazione per la profondità della soluzione, se devo salvarmi tutti i nodi visitati torniamo ad una complessità esponenziale.

## Algoritmi con strategie di ricerca non informate

In questi algoritmi per la ricerca di una soluzione non facciamo delle supposizioni riguardanti la distanza verso lo stato soluzione, l'opposto è

rappresentato dalle strategie di ricerca euristica in cui vengono utilizzate delle stime.

I principali sono:

- Ricerca Breadth First ovvero in ampiezza
- Ricerca Depth First ovvero in profondità
- Ricerca DL ovvero in profondità limitata
- Ricerca con approfondimento iterativo
- Ricerca con costi uniformi

Come valuto la qualità di una strategia:

- Una strategia è completa se trova almeno una soluzione tra quelle esistenti
- Una strategia è ottima se trova la migliore soluzioni tra quelle disponibili
- Deve essere considerato quanto tempo ci vuole per trovare la soluzione, questa è la complessità in tempo
- Poi c'è la complessità in spazio

## Confronto delle strategie (albero)

Criterio	BF	UC	DF	DL	ID	Bidir
Completa?	si	si( $\wedge$ )	no	si (+)	si	si
Tempo	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^d)$	$O(b^d)$	$O(b^{d/2})$
Spazio	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Ottimale?	si(*)	si( $\wedge$ )	no	no	si(*)	si

## Ricerca BF - In ampiezza

Con questo tipo di ricerca partiamo dal nodo di partenza e lo espandiamo, quando otteniamo i nodi figli facciamo anche un controllo e vediamo se quel nodo che ho generato è uno stato obiettivo, nella frontiera vengono messi i nodi figli. Ora prendiamo i nodi figli che stanno in frontiera e li espandiamo in ordine di arrivo.

```
function Ricerca-Ampiezza-A (problema)
  returns soluzione oppure fallimento
  nodo = un nodo con stato il problema.stato-iniziale e costo-di-cammino=0
  if problema.Test-Obiettivo(nodo.Stato) then return Soluzione(nodo)
  frontiera = una coda FIFO con nodo come unico elemento
  loop do
    if Vuota?(frontiera) then return fallimento
    nodo = POP(frontiera)
    for each azione in problema.Azioni(nodo.Stato) do
      figlio = Nodo-Figlio(problema, nodo, azione) [costruttore: vedi AIMA]
      if Problema.TestObiettivo(figlio.Stato) then return Soluzione(figlio)
      frontiera = Inserisci(figlio, frontiera) /* frontiera gestita come coda FIFO
  end
```

*ansione* {

Nota che in questa versione gli stati sono goal-tested momento in cui sono generati. → più efficienti ferma appena trova goal prima di espandere

In questo caso se utilizzo un albero di ricerca posso avere dei cicli e mi posso trovare a controllare più volte lo stesso nodo che ho già controllato. Questo problema si risolve eseguendo la ricerca in ampiezza sul grafo e non sull'albero, in questo caso dobbiamo salvarci l'insieme dei nodi che abbiamo già esplorato in modo da non espanderli di nuovo. Nel momento in cui tolgo un nodo dalla frontiera lo metto tra gli esplorati, quando lo espando controllo che i figli non siano già tra gli esplorati e poi li metto in frontiera.

### Legenda:

- b = fattore di ramificazione
- d = profondità del nodo obiettivo più vicino
- m = lunghezza massima dei cammini nello spazio degli stati

## Analisi:

- La BF è completa
- La BF è ottima se i passaggi tra i vari nodi hanno lo stesso costo
- La complessità in tempo è  $O(b^d)$
- La complessità in spazio è  $O(b^d)$

## Ricerca in profondità

In questo caso l'implementazione della coda è LIFO quindi non si vanno a espandere tutti i nodi per livelli ma si espande un nodo e poi si vanno ad espandere i figli di quel nodo.

## Analisi:

- La DF non è completa su albero perchè potrei avere dei loop
- La DF non è ottima
- La complessità in tempo dipende da  $m$  che sarebbe la lunghezza massima dei cammini nello spazio degli stati ed è  $O(b^m)$
- La complessità in spazio è minore della BF perchè devo mantenere solamente un numero di nodi in frontiera che dipendono dal fattore di diramazione e dalla lunghezza massima dei cammini, quindi è  $O(b \cdot m)$

Quando invece si usa la DF su un grafo salvando quindi i vari nodi che si controllano devo anche salvarmi lo storico dei nodi quindi la complessità torna ad essere  $O(b^d)$  ovvero pari al numero di nodi presenti all'interno dello spazio degli stati

La ricerca in profondità può essere svolta in modo ricorsivo per migliorare ancora di più la complessità in spazio.

In questo caso infatti l'occupazione di spazio è ridotta da  **$b \cdot m$  a  $m$  nodi salvati ogni volta.**

```

function Ricerca-DF-ricorsiva(nodo, problema)
  returns soluzione oppure fallimento
  if problema.TestObiettivo(nodo.Stato) then return Soluzione(nodo)
  else
    for each azione in problema.Azioni(nodo.Stato) do
      figlio = Nodo-Figlio(problema, nodo, azione)
      risultato = Ricerca-DF-ricorsiva(figlio, problema)
      if risultato ≠ fallimento then return risultato
  return fallimento

```

## Ricerca a profondità limitata (DL)

Si tratta di una versione modificata della DF, in questo caso fissiamo un limite (chiamato L) e non andiamo in profondità più di quel limite.

### Analisi:

- Questa strategia è completa se  $d < L$
- Non è una strategia ottima
- La complessità in tempo è  $O(b^L)$
- La complessità in spazio è  $O(b \cdot L)$

## Ricerca con approfondimento iterativo (ID)

Questa strategia si utilizza in coppia con quella a profondità limitata perchè grazie a questa strategia riusciamo a determinare una L a cui fermarsi.

La usiamo con uno spazio ampio quando non conosciamo la profondità della soluzione.

Noi determiniamo ad ogni esecuzione un limite, partiamo da 1 e poi andiamo avanti fino a trovare una soluzione. Si tratta di una visita in profondità a cui mettiamo un limite e ogni volta lo aumentiamo se non troviamo lo stato soluzione.

### **Analisi:**

- E' completa
- E' ottima
- $O(b^d)$
- $O(b^d)$  perchè è comunque una visita in profondità

### **Ricerca Bidirezionale**

Si tratta di una strategia di ricerca che può essere utilizzata quando conosciamo lo stato obiettivo e lo stato di partenza ma non sappiamo come arrivare a destinazione.

Quello che si fa è partire dalla destinazione e dallo stato iniziale e cercare una strada per lo stato iniziale e per la destinazione, a metà strada le ricerche si incontrano con la loro diramazione.

Viene usata la BF ma non si può usare sempre, ad esempio se ho molti predecessori per un nodo non la uso.

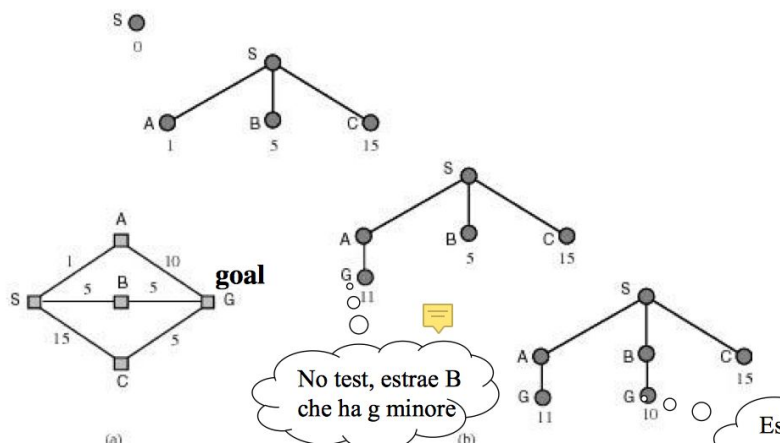
### **Analisi:**

- Complessità in tempo  $O(b^{d/2})$
- Complessità in spazio  $O(b^{d/2})$
- E' una strategia completa
- E' una strategia ottima

### **Ricerca con costo Uniforme**

Si tratta di una generalizzazione della ricerca in ampiezza che prevede l'utilizzo di costi differenti per i vari collegamenti.





Quello che si fa è generare il primo nodo, si fa il controllo per vedere se è il goal e poi se non è il goal vado a generare i vari figli.

Tra tutti i figli prendo quello che nella frontiera ha il costo minore per essere raggiunto.

Quando prendo il primo nodo e lo espando metto nella frontiera i figli del nodo espanso e prendo di nuovo il nodo che ha il costo minore che potrebbe anche essere un nodo generato in precedenza e non uno degli ultimi generati.

**function** Ricerca-UC-A (*problema*)

**returns** soluzione oppure **fallimento**

*nodo* = un nodo con *stato* il *problema.stato-iniziale* e *costo-di-cammino*=0

*frontiera* = una coda con priorità con *nodo* come unico elemento

**loop do**

**if** Vuota?(*frontiera*) **then return** fallimento

*nodo* = POP(*frontiera*)

**if** *problema.TestObiettivo*(*nodo.Stato*) **then return** Soluzione(*nodo*)

**for each** *azione* **in** *problema.Azioni*(*nodo.Stato*) **do**

*figlio* = Nodo-Figlio(*problema*, *nodo*, *azione*)

*frontiera* = Inserisci(*figlio*, *frontiera*) /\* in coda con priorità

**end**

Qui per esaminare pos  
e vedere il costo minor  
(diverso da BF, ma tipico per

Questo tipo di ricerca con costi uniformi può essere svolta anche su grafo, in questo caso ovviamente la differenza sta nel fatto che si devono andare ad inserire all'interno di una seconda lista i nodi che abbiamo già visitato in modo da evitare di andare in loop.

## Analisi:

- Si tratta di una strategia che è completa e ottima solamente se il costo dei vari archi è maggiore di 0
- La complessità di tempo e spazio è  $O(b^{(1+C/e)})$  dove  $C$  è il costo della soluzione ottima e  $C/e$  è il numero di mosse nel caso peggiore.

# Terza Lezione

Utilizzando la ricerca esaustiva ci troviamo ad avere un problema dovuto al fatto che il tempo e lo spazio necessario hanno spesso complessità esponenziale. Quindi si passa ad un metodo differente di ricerca per trovare la soluzione ad un problema, si tratta di una ricerca euristica che si basa su conoscenza che abbiamo già o che otteniamo tramite l'esperienza per trovare una soluzione al problema che non richieda un tempo eccessivo.

Per la valutazione di un cammino si utilizza quindi una euristica, è differente dal caso della ricerca esaustiva in cui noi consideriamo solamente il costo per arrivare ad un certo nodo e scegliamo il minore:  $f(n) = g(n)$ .

In questo caso infatti si considera anche l'euristica espressa come  $h(n)$ , quindi diventa  $f(n) = g(n) + h(n)$ .

Un esempio di euristica nell'esercizio della romania potrebbe essere la distanza in linea d'aria tra le varie città.

- **Best-First:** un primo algoritmo di ricerca che utilizza l'euristica è il Best First, questo si basa su UC, quindi avviene una espansione di un nodo e poi prendiamo quello con la  $f(n)$  migliore tra quelli espansi. Il processo va avanti e viene mantenuta una frontiera con tutti i nodi espansi.
- Best First ha anche una versione particolare chiamata **Greedy Best-First** in cui non andiamo a considerare la  $f(n)$  per decidere quale nodo espandere ma consideriamo solamente l'euristica da quel nodo, quindi in pratica è il caso in cui  $f(n) = h(n)$ .
- **Algoritmo A:** si tratta di Best-First in cui utilizziamo come valutazione la  $f(n) = g(n) + h(n)$  ma abbiamo il vincolo che  $h(n) > 0$  e  $h(goal) = 0$  con  $h(n)$  stima del costo per andare dal nodo  $n$  al goal.

**Teorema:** A è un algoritmo completo se rispetta la condizione  $g(n) \geq d(n) * e$ . Con  $d(n)$  uguale alla profondità del nodo  $n$  ed  $e$  uguale

al costo del minimo arco (un numero  $> 0$ ).

**Dimostrazione:** Consideriamo un cammino soluzione che comprende un certo nodo  $N$  nella frontiera.

Questo nodo  $N$  prima o poi verrà espanso perchè vale la relazione tra  $g(n)$  e  $d(n)$  e quindi ho solamente un numero limitato di nodi  $X$  in cui vale  $f(X) \leq f(N)$ .

Quindi o faccio l'espansione del nodo  $N$  o trovo prima la soluzione.

Posso continuare il ragionamento anche sui nodi successori di  $N$ .

- **Algoritmo A\*:** maggiori dettagli sotto

## Algoritmo A\*

La migliore funzione di valutazione è quella che prende il minimo costo per arrivare dalla radice ad un nodo  $N$  e quella che prende il minimo costo per arrivare dal nodo  $N$  alla destinazione, la possiamo scrivere come:

$$f^*(N) = g^*(N) + h^*(N)$$

Solitamente però non troviamo il cammino minimo verso il nodo  $N$  e la nostra stima  $h(N)$  è una sottostima o una sovrastima.

Se la nostra euristica prevede una  $h(N)$  che sottostima la  $h^*(N)$  allora vuol dire che l'euristica può essere definita ammissibile.

**Nell'algoritmo A\* la funzione euristica deve essere ammissibile**, se così non fosse allora con una sovrastima perderei anche la soluzione ottimale, sottostimando troppo perdo tempo ma trovo la soluzione ottima.

A\* è un algoritmo **ottimale** e quindi lo sono anche Best First con costo costante e UC.

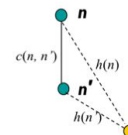
**Su albero** Infatti se uso una euristica ammissibile questo mi dice che A\* sarà anche ottimale.

**Su grafo** invece devo aggiungere la monotonicità per ottenere l'ottimalità, ovvero:

**Euristica monotona o consistente:** una euristica è monotona (o consistente) se dati i nodi  $N$  e  $N'$  abbiamo che  $h(\text{goal}) = 0$  e che  $h(N) < c(N, N') + h(N')$  ovvero  $f(N) \leq f(N')$ .

L'euristica che considero per arrivare da  $N$  al goal deve essere minore di quella che considero per andare dal nodo  $N'$  al goal perchè dato che mi muovo verso il basso conosco un costo preciso di una distanza tra  $N$  e  $N'$ .

Se  $h(n) \leq c(n, \alpha, n') + h(n')$  def. consistenza  
 $g(n) + h(n) \leq g(n) + c(n, \alpha, n') + h(n')$  sommando  $g(n)$   
 ma siccome  $g(n) + c(n, \alpha, n') = g(n')$   
 $g(n) + h(n) \leq g(n') + h(n')$   
 $f(n) \leq f(n')$



**Se una euristica è monotona allora è anche ammissibile e** soprattutto sappiamo con certezza che la soluzione migliore viene trovata subito.

Infatti ogni volta che  $A^*$  seleziona un nodo allora vuol dire che quello è il cammino migliore per arrivare a quel nodo (altrimenti avremmo in frontiera un nodo con  $f(N')$  minore di  $f(N)$  ma non sarebbe possibile).

In  $A^*$  abbiamo anche il pruning ovvero ci troviamo a poter tagliare alcuni rami in cui  $f(N) >$  del costo reale per arrivare a quel nodo, più la  $f$  è vicina alla  $f^*$  più ho la possibilità di tagliare quindi quello che devo fare è prendere la  $h(N)$  più alta possibile purchè questa rimanga ammissibile.

Come costruiamo e valutiamo le euristiche?

Avevamo detto che una euristica è ammissibile se la  $h(n) < h^*(n)$  quindi se è una sottostima del costo reale. Due euristiche potrebbero essere una sottostima e devo poter dire quale è migliore. Per decidere la migliore devo prendere la più informata ovvero quella con il valore più alto, questa infatti mi permette di visitare un numero minore di nodi.

Il contro è che è più complicato calcolare una euristica più informata rispetto all'euristica meno informata.

Se ho varie euristiche ammissibili, prendo quella con la  $h$  più alta.

**Teorema:** Date due euristiche,  $h_1$  più informata e  $h_2$  meno informata, i nodi espansi con  $A^*$  e  $h_1$  sono un sottoinsieme di quelli espansi con  $A^*$  e  $h_2$ .

Per valutare poi un algoritmo che usa una certa euristica posso anche calcolare la diramazione effettiva dell'albero, questa viene indicata con  $b^*$  e dipende dai nodi che genero e dalla profondità a cui sono arrivato.

Per costruire una euristica ci sono varie strategie:

- Rilassamento del problema: dato il problema iniziale, per ottenere una euristica andiamo a eliminare alcuni vincoli del problema in modo da poter arrivare ad una soluzione più velocemente.
- Sottoproblemi: non voglio risolvere subito tutto il problema ma vado a risolvere una parte del problema e poi magari ne risolvo un'altra parte. Prendo una euristica per ogni parte che cerco di risolvere e alla fine scelgo quella con la  $h$  maggiore.  
Non posso prendere i singoli sottoproblemi e sommare le  $h$  in modo da ottenere una euristica che vale per tutti, dobbiamo invece utilizzare dei pattern disgiunti che mi fanno considerare solamente le euristiche su una parte del problema.
- Esperienza: raccogliamo delle coppie  $\langle \text{stato}, h^* \rangle$  e cerchiamo di predire il valore della  $h$  per ottenere un valore più alto possibile della  $h$
- Combinazione: in alcuni casi possiamo unire più euristiche, non è una semplice somma di varie euristiche ma devo andare a dare un peso alle singole euristiche che voglio considerare. In questo modo otterrò una euristica complessiva da utilizzare per il problema.

## Altri algoritmi basati su A\*

- Beam Search: per ridurre la dimensione della frontiera e quindi ridurre il consumo di spazio, l'algoritmo Beam Search va a salvare nella frontiera solamente un numero  $K$  di nodi con il  $f(N)$  più basso. Questo fa sì che non si abbia completezza perché dato che tengo solamente i nodi più promettenti potrei scartarne altri che all'inizio non sembrano buoni ma che invece potrebbero contenere il percorso verso la soluzione
- IDA\*: è una unione tra ID e A\* con la differenza che non ho un limite legato alla profondità ma un limite legato alla  $f$ . Quando la  $f$  diventa troppo alta mi fermo, alzo il limite della  $f$  e poi ricomincio. Vado avanti così fino a quando non trovo la soluzione. Per l'incremento della  $f$  posso fissare un valore fisso che viene utilizzato ogni volta per incrementare il valore attuale. Questo algoritmo è completo e ottimale e ho una occupazione di memoria di  $O(bd)$  dove  $b$  è il fattore di diramazione e  $d$  è la profondità a cui sono arrivato.
- Best First Ricorsivo: si tratta di una versione della BF che procede ricorsivamente occupando una quantità di memoria lineare alla profondità dell'albero. Nel momento in cui scelgo un nodo per l'espansione tengo anche in memoria il valore del secondo nodo migliore (in base alla  $f$ ). Se il nodo che vado ad espandere poi ha come successori dei nodi con valori della  $f$  maggiori di quella che ho in memoria torno indietro e vado ricorsivamente sul secondo nodo migliore, quello che abbandono lo marco modificando il valore della  $f$  in modo da ricordarmi il valore dopo l'espansione.

# Quarta Lezione

Fino ad ora abbiamo fatto delle assunzioni sul tipo di ambiente che viene utilizzato, in particolare l'ambiente è stato sempre considerato statico e osservabile. Nella realtà non è così, l'ambiente non sempre è completamente osservabile e spesso ad una azione non corrisponde solamente una reazione ma ne potrebbero corrispondere più di una.

Dalla ricerca euristica e da quella non informata si passa quindi alla ricerca locale, in questo caso quello che ci interessa non è il modo in cui arriviamo alla soluzione del problema ma la soluzione vera e propria del problema ovvero il goal.

Il percorso non conta in questo caso e quindi dato che non teniamo in memoria il cammino vuol dire che abbiamo anche un risparmio dal punto di vista della memoria.

Da un nodo in cui mi trovo quindi vado a spostarmi in un nodo vicino senza tenere conto del cammino basandomi su una funzione obiettivo (ad esempio voglio minimizzare il costo).

## Hill Climbing

Si tratta di una ricerca greedy perchè quando mi trovo in un punto poi devo capire se muovendomi miglio o no la mia situazione attuale.

```
def hill_climbing(problem): """ Ricerca locale - Hill-climbing. """
    current = Node(problem.initial_state)
    while True:
        neighbors = [current.child_node(problem, action) for action in
                     problem.actions(current.state)]
        if not neighbors: # se current non ha successori esci e restituisci current
            break
        # scegli il vicino con valore piu' alto (sulla funzione problem.value)
        neighbor = (sorted(neighbors, key = lambda x: problem.value(x), reverse = True))[0]
        if problem.value(neighbor) <= problem.value(current):
            break
        else:
            current = neighbor # (altrimenti, se vicino e' migliore, continua)
    return current
```



Allo stato attuale fornisco una valutazione e tra gli stati vicini posso scegliere:

- Il migliore ovvero quello in cui la funzione di valutazione è migliore
- Uno a caso (scelto tra quelli che migliorano la mia soluzione)
- Il primo che capita

Potrei utilizzare Hill Climbing per risolvere il problema delle 8 regine, sia se trovo una soluzione, sia se non la trovo mi ci vogliono poche mosse, il problema è che spesso non arrivo a trovare la soluzione perchè mi si potrebbero creare dei minimi locali ed è difficile trovare una situazione di minimo assoluto partendo dal minimo locale.

Questo è uno dei difetti di Hill Climbing, ce ne sono vari:

- L'algoritmo potrebbe fermarmi su un massimo locale (o su un minimo locale) se si trova nella situazione in cui spostandosi dallo stato attuale peggiora la sua situazione
- L'algoritmo si potrebbe fermare su una plateau se mettiamo come richiesta per lo spostamento che il nuovo stato deve essere migliore dell'attuale, se invece consideriamo anche il caso in cui possa essere uguale allora non mi blocco (mi fermo con  $<$  e non con  $\leq$ )
- Ci possono essere situazioni in cui mi trovo su un crinale da cui è complicato scendere e risalire

Posso risolvere parzialmente questi problemi (a discapito della velocità di esecuzione) utilizzando la versione stocastica dell'Hill Climbing, in questo caso se ho varie mosse che mi portano ad uno stato migliore prendo una casuale tra tutte.

Posso anche effettuare un riavvio casuale, ovvero vado avanti fino ad un certo punto, mi salvo un punto di ripristino e quando sono bloccato vado a ripartire da quel punto di ripristino. Con questo metodo vado a generare tutte le possibili combinazioni quindi il metodo non è molto efficiente ma l'algoritmo in questo modo diventa completo.

## **Simulated Annealing**

Un altro algoritmo di ricerca locale è il simulated Annealing che sfrutta un valore  $T$  di “temperatura” per arrivare a trovare una soluzione per il problema.

Mi trovo in una certa situazione e ho vari possibili successori:

- Considero un successore (preso a caso) se miglioro la situazione attuale allora prendo quel nodo e lo espando
- Se non miglioro la situazione attuale allora devo andare a calcolarmi un delta  $E$ :

$$E = f(n') - f(n)$$

Quando non miglioro questa  $E$  è minore di 0 e quindi associo al nodo una probabilità pari a

$$p = e^{(E/T)}$$

Dove la  $T$  è un valore che diminuisce a mano a mano che vado avanti con l'esecuzione dell'algoritmo

La probabilità diventa quindi inversamente proporzionale al coefficiente che ho calcolato, se non ho nodi che migliorano la situazione, genero un numero casuale compreso tra 0 e 1 e lo confronto con le probabilità associate ai vari nodi che ho, se il numero casuale generato è minore di  $p$  allora prendo quel nodo.

## Local Beam

Una modifica della beam search in cui non vado ad espandere solamente un nodo alla volta ma espando  $k$  nodi alla volta. Quando ho fatto l'espansione, prendo solamente i migliori  $K$  nodi tra quelli espansi e poi vado ad espandere di nuovo questi nuovi  $K$  nodi.

## Beam Search stocastica

Nella versione stocastica della Beam Search non mi limito a tenere i migliori successori come nella Beam Search classica, ad ogni nodo

assegno una probabilità e poi viene scelto il nodo che ha la probabilità migliore tra quelli che ho a disposizione.

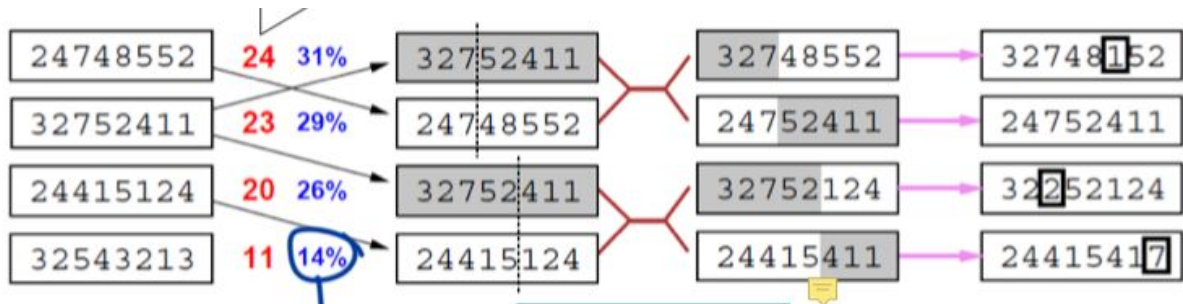
## Algoritmi Genetici

Si tratta di una evoluzione della beam search stocastica, vado ad unire due stati per creare gli stati successivi.

In questi algoritmi gli stati che abbiamo inizialmente rappresentano la popolazione e ogni stato è un nodo che viene giudicato in base ad una funzione di fitness.

In base al valore del fitness vado a selezionare gli stati più promettenti in modo che unendosi possano generare nuovi stati prendendo parte del “patrimonio genetico” degli stati “genitori” e aggiungendo delle mutazioni genetiche. Il patrimonio genetico dei genitori viene “tagliato” in un punto detto punto di crossing over (scelto casualmente) e mescolato.

A mano a mano che si va avanti con l’unione di stati sempre migliori, ci saranno meno differenze tra uno stato e il successivo.

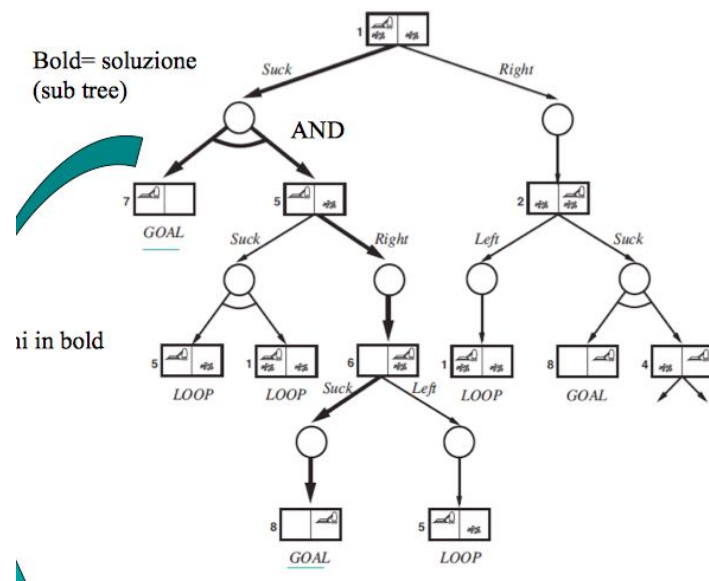


Il funzionamento di questo tipo di problemi è garantito se i vari nodi del problema vengono espressi sotto forma di stringhe.

Come avevamo detto gli ambienti non sono sempre visibili e le azioni non sempre causano solamente un cambiamento, ne possono causare vari.

Quindi quello che si vuole fare è sfruttare le percezioni e fare in modo che quando svolgo una azione questa possa portare comportamenti differenti.

Questi differenti comportamenti li possiamo esprimere con gli alberi AND-OR in cui utilizziamo l'OR quando dobbiamo scegliere tra una azione a l'altra che verrà svolta dall'agente e usiamo l'AND quando invece dobbiamo scegliere tra i possibili stati in cui mi vado a trovare dopo aver svolto una determinata azione.



# Quinta lezione

## Giochi con avversario

Si tratta di problemi in cui intervengono due agenti “competitivi” che cercano di vincere il gioco, l’ambiente è accessibile e deterministico, un giocatore vince e uno perde.

La difficoltà per l’agente è scegliere la mossa migliore nel minor tempo possibile, potrebbe anche esserci un vincolo legato al tempo.

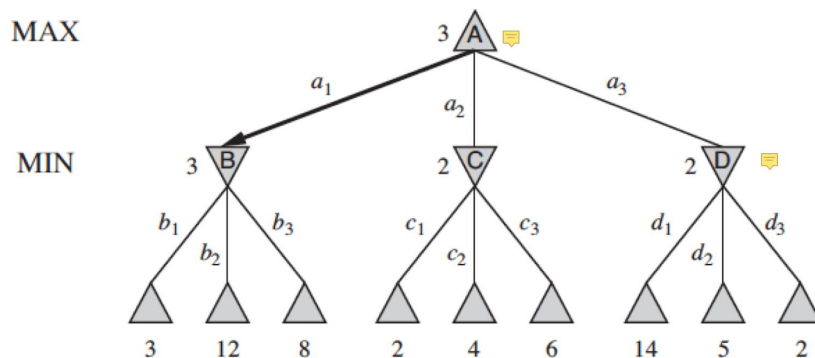
Due agenti che muovono a turno la loro pedina in un certo gioco partono da un certo stato, a partire da questo stato l’agente cerca di capire come si può evolvere lo stato e in che modo può riuscire a vincere, quindi guarda in avanti e cerca la mossa migliore da fare. Poi l’agente avversario fa la stessa cosa e sceglie la mossa e si ricomincia il giro.

In questi giochi con avversari si usano i seguenti termini:

- **Stati**: configurazione di gioco, lo stato iniziale è quello da cui si parte, quello terminale è lo stato in cui si arriva
- **Azioni**: mosse possibili che mi portano da uno stato al successivo
- **Risultato(a,s)**: risultato (stato) che si ottiene quando faccio una azione a in un certo stato s
- **Test di terminazione**: test che svolgo su un certo stato per capire se sono arrivato alla fine del gioco oppure no
- **Funzione di utilità**: funzione che mi fornisce un valore numerico per ogni stato finale dando quindi una valutazione

Per capire la mossa migliore da fare si utilizza l’algoritmo Min Max in cui ad ogni livello dell’albero prendiamo il massimo della funzione di utilità dei successori poi al livello successivo il minimo, poi di nuovo il max e così via. In particolare quando devo prendere il valore massimo vuol dire che è la mossa del primo agente che cerca di fare la mossa migliore per lui, quando invece prendo il valore minimo sto cercando di essere

pessimista e di dire che l'avversario sceglierà la mossa migliore per lui che è peggiore per me, quindi prendo il valore di utilità minore che c'è.



Per questa esplorazione mi conviene utilizzare un algoritmo ricorsivo in profondità in modo da evitare di espandere troppi stati inutili risparmiando quindi spazio e tempo.

Se siamo nel livello in cui devo scegliere il valore massimo faccio così:

- Assegno a  $v$  meno infinito, in  $v$  salverò il valore massimo della funzione di utilità
- Per ogni azione possibile a partire dallo stato in cui mi trovo prendo l'azione e assegno a  $v$  il massimo valore tra  $v$  e il valore che ottengo andando dallo stato in cui mi trovo al successivo usando l'azione  $a$  (Uso Min-Value perchè se sono nello stato Max il successivo è Min)

Per il minimo l'algoritmo è praticamente lo stesso solamente che  $v$  è fissata a infinito all'inizio e poi prendo il minimo tra  $v$  e il valore massimo che ho andando dallo stato attuale al prossimo con l'azione  $a$ .

```

function MINIMAX-DECISION(state) returns an action
  return  $\arg \max_{a \in \text{ACTIONS}(s)} \text{MIN-VALUE}(\text{RESULT}(\text{state}, a))$ 

```

```

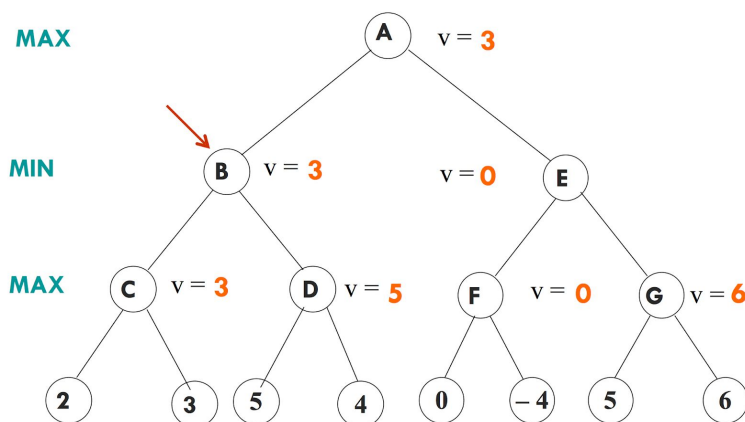
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow -\infty$ 
  for each a in ACTIONS(state) do
     $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(s, a)))$ 
  return v

```

```

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow \infty$ 
  for each a in ACTIONS(state) do
     $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(s, a)))$ 
  return v

```



E' ricorsivo in profondità quindi estraggo: B,C,2,3,D,5,4,E,F,0,-4,G,5,6.  
 Dato che comunque esploro tutti i nodi il costo in tempo sarà  $O(b^m)$   
 quindi dipende da  $b$ =diramazione e da  $m$ =profondità. Alla fine devo  
 esplorare tutti i nodi ed è troppo costoso in tempo anche se risparmio lo  
 spazio che in questo caso è  $O(m)$ .  
 Come si migliora? Usando le euristiche.

## Min-Max Euristico

Si utilizza sempre l'algoritmo Min-Max con la differenza che in questo caso abbiamo un limite  $d$  per la profondità, quando raggiungiamo quel limite allora dobbiamo calcolare per ogni nodo una funzione di valutazione e poi con min max spediamo sopra il valore di quel nodo. Non arriviamo in fondo quindi non abbiamo una vera e propria funzione di utilità, si tratta di una valutazione parziale del valore di uno stato.

Ad esempio nel gioco del filetto posso utilizzare come euristica il numero di righe in cui un giocatore può ancora fare filetto. La funzione di valutazione dei nodi alla profondità  $d$  sarà uguale a (righe aperte per X - righe aperte per Y) in questo modo ho un valore e lo utilizzo nell'albero Max Min.

La funzione di valutazione è comunque una stima della funzione di utilità che avrei andando avanti su quel percorso, deve essere fatta bene e deve essere efficiente, mi deve effettivamente dire chi ha più probabilità di vittoria in un certo stato in cui mi trovo.

La funzione di valutazione può anche essere pesata, ad esempio nel gioco degli scacchi potrei dare un certo valore ai vari pezzi e poi farei la somma pesata di tutti i pezzi che mi rimangono.

Qui però sorge un problema, infatti potremmo trovarci in una situazione in cui ad un livello ho una certa valutazione che mi mostra una situazione in cui io vincerei, al livello successivo però magari mi mangiano la regina e quindi la funzione di valutazione ha un cambiamento repentino, quindi quello in cui ho fatto la valutazione è uno stato non quiescente.

L'idea quindi è quella di calcolare la funzione di valutazione in stati quiescenti ovvero in stati in cui non ci sono cambiamenti repentini della funzione di valutazione.

Un altro problema che potremmo avere è l'effetto orizzonte, ovvero ci troviamo in una situazione in cui il giocatore ha praticamente perso ma delle mosse diversive mi fanno arrivare fino al punto in cui devo fermarmi.

Si possono anche fare dei miglioramenti, in particolare abbiamo la tecnica di potatura alfa beta che si applica all'algoritmo min-max per ridurre il numero dei nodi visitati.

Quando sono in uno stato di minimo e sopra c'è quello di massimo, posso evitare di espandere i nodi che so già che non verranno presi in considerazione.

Ad esempio qua abbiamo che il primo nodo con 3 è già maggiore del secondo che ha solamente 2 ma dato che devo prendere il minimo sarà anche meno di 2, quindi non lo espando tutto.





```

function MIN-VALUE(state,  $\alpha$ ,  $\beta$ ) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow -\infty$ 
  for each a in ACTIONS(state) do
     $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(s, a), \alpha, \beta))$ 
    if  $v \leq \alpha$  then return  $v$      $\leftarrow$  taglio  $\alpha$ 
     $\beta \leftarrow \text{MIN}(\beta, v)$ 
  return  $v$ 

```

La complessità di alpha beta è  $O(b^{(m/2)})$  e quindi possiamo andare più in profondità rispetto a Min Max.

Si possono anche ordinare le mosse in modo dinamico usando l'approfondimento iterativo in cui ad ogni iterazione ci fermiamo ad una certa profondità ma otteniamo dei dati interessanti da utilizzare nella successiva iterazione.

Oltre all'ordinamento dinamico delle mosse, per migliorare la complessità di esplorazione dell'albero abbiamo anche la possibilità di tagliare le mosse che non sono ritenute promettenti e poi possiamo anche utilizzare delle mosse di apertura e di chiusura.

Fino ad ora abbiamo considerato solamente giochi semplici con lo stato completamente osservabile (scacchi), ci sono però dei giochi in cui interviene il caso o la probabilità, basta pensare al lancio dei dadi o alle carte.

## Problemi CSP - Soddisfacimento di vincoli

Sono problemi particolari in cui ci conviene pensare a degli algoritmi di ricerca specializzati nella ricerca di soluzioni.

I problemi che fanno parte di questa classe sono molti, ad esempio lo scheduling o il layout di circuiti.

Come si formula il problema:

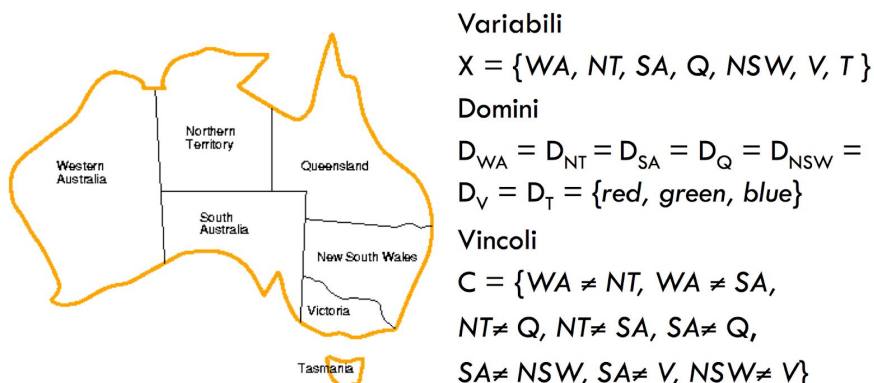
- Serve un insieme di variabili che rappresenta i vari elementi del problema
- Serve un dominio ovvero l'insieme dei valori che possono essere assegnati alle variabili
- Servono i vincoli che devono esistere ed essere validi tra le varie variabili

Lo stato è un assegnamento di valori a variabili, questo può essere completo o parziale.

Quando arriviamo ad un assegnamento che rispetta tutti i vincoli e dove le variabili hanno un valore consistente allora ho raggiunto la soluzione.

Un esempio di soddisfacimento di vincoli è la colorazione di una mappa per fare in modo che zone vicine abbiano colorazioni differenti.

*Si tratta di colorare i diversi paesi sulla mappa con tre colori in modo che paesi adiacenti abbiano colori diversi*



Posso rappresentare i vincoli in un grafo in cui unisco i vari nodi che rappresentano le zone, in particolare due nodi sono uniti se questi due hanno un vincolo che li riguarda. Ad esempio avrò un arco che unisce WA con NT e uno tra WA e SA.

Per risolvere problemi di questo tipo con il soddisfacimento di vincoli si utilizzano diverse strategie, in particolare usiamo delle euristiche e facciamo una inferenza che ci permette di assegnare i valori alle variabili restringendo il dominio, quando mi trovo in stati con vincoli violati devo anche poter fare il backtracking ovvero posso tornare indietro ad una situazione consistente. Il backtracking lo faccio cercando di accorgermi subito quando un vincolo viene violato, in questo modo evito di continuare ad assegnare valori e posso subito tornare indietro.

Quello che si fa in questi problemi quindi è fare un assegnamento di valori alle variabili e cercare di arrivare ad uno stato consistente.

Come scelgo le variabili? Uso delle euristiche:

- MRV: Minimum remaining values, scelgo la variabile in cui ho il minor numero di valori possibili ovvero quella più vincolata
- Euristiche del grado: scelgo la variabile che è coinvolta in più vincoli possibili con altre variabili

Come scelgo i valori?

- Scelgo di assegnare ad una certa variabile un valore che è poco vincolante. Questo vuol dire che scelgo per la mia variabile un valore che non esclude la possibilità di assegnare un altro valore ad una seconda variabile che è vincolata da quella che sto assegnando.

Come utilizzo i vincoli?

- Forward Checking (FC) : vuol dire che quando assegno un valore ad una variabile posso subito rendermi conto che alcuni vincoli non

sono più rispettati, quindi posso eliminare i valori incompatibili in alcune variabili.

Il backtracking lo faccio quando mi blocco in uno stato che viola i vincoli e per farlo in modo intelligente devo tornare ad una situazione precedente in cui l'assegnamento andava bene e non mi basta tornare all'ultimo assegnamento.

Una alternativa per i giochi CSP è il metodo Min-Conflicts che si usa con le 8 regine, posiziono casualmente le regine nella scacchiera poi vado a vedere per ogni posizione della regina nella colonna quanti vincoli sono violati ovvero quante regine mi mangerebbero.

Quindi mi sposto nella casella dove vengono violati meno vincoli e faccio lo stesso anche con le altre regine dove vengono violati vincoli.

Mi sposto sempre dove ho il numero minimo di conflitti.

# Sesta Lezione

## Agenti basati su conoscenza

Vogliamo migliorare gli agenti fornendo una conoscenza basilare che possa rappresentare dei mondi più complessi. Questi agenti sono dotati di una KB (Knowledge Base).

E' necessario avere una rappresentazione del mondo, dato che il mondo è complesso questa rappresentazione sarà parziale e incompleta quindi il linguaggio che si usa per rappresentarla deve essere espressivo e deve dare la possibilità di eseguire inferenza.

L'agente che ha a disposizione una KB può modificarla e aggiornarla in base all'esperienza e quindi le azioni che svolge possono variare nel tempo, se invece avessi un agente scritto in modo procedurale avrei solamente una serie di cose che vengono fatte per sempre senza possibilità di aggiornamenti.

KB: insieme di enunciati espressi in un certo linguaggio e rappresentano ciò che l'agente crede. L'agente può interagire con questa KB andando ad aggiungere nuova conoscenza nella KB o a prendere la conoscenza interrogando la KB.

Le risposte alle interrogazioni devono essere conseguenze della KB stessa, questo si scrive come  $KB \models a$  ovvero  $a$  è conseguenza logica della KB.

Il problema è capire quando un fatto è conseguenza logica e quando non lo è.

La base di conoscenza è differente dalla base di dati, la base di dati infatti riguarda solamente dei fatti specifici mentre la base di conoscenza KB è in grado di fare inferenza e di ottenere delle regole generali dai fatti specifici di cui è a conoscenza.

Per rappresentare la conoscenza si deve fare una mediazione tra l'espressività del linguaggio e la complessità dell'inferenza, più è semplice il linguaggio e più si complica l'inferenza.

Il linguaggio che usiamo per la rappresentazione ha una sintassi e una semantica e un meccanismo che mi consente di fare inferenza ovvero ottenere nuovi dati a partire da quelli che abbiamo a disposizione all'interno della KB.

Per rappresentare la conoscenza della KB si utilizza la logica con linguaggi come il FOL e il PROP.

## Calcolo proposizionale

Il linguaggio che si utilizza per rappresentare la KB ha la seguente sintassi:

$$\begin{aligned}
 \text{formula} &\rightarrow \text{formulaAtomica} \mid \text{formulaComplessa} \\
 \text{formulaAtomica} &\rightarrow \mathbf{True} \mid \mathbf{False} \mid \text{simbolo} \\
 \text{simbolo} &\rightarrow \mathbf{P} \mid \mathbf{Q} \mid \mathbf{R} \mid \dots \\
 \text{formulaComplessa} &\rightarrow \neg \text{formula} \\
 a &\quad \mid \quad ( \text{formula} \wedge \text{formula} ) \\
 &\quad \mid \quad ( \text{formula} \vee \text{formula} ) \\
 &\quad \mid \quad ( \text{formula} \Rightarrow \text{formula} ) \\
 &\quad \mid \quad ( \text{formula} \Leftrightarrow \text{formula} )
 \end{aligned}$$

Non si utilizzano parentesi per rappresentare le varie frasi perchè utilizziamo una precedenza tra gli operatori che evita situazioni ambigue:



La semantica di una certa frase mi indica se questo enunciato è vero o falso basandosi su una interpretazione. L'interpretazione dipende dal mondo e indica un valore di verità per ogni simbolo presente nella frase.

**Devo dare una interpretazione alla formula ovvero assegnare un valore ai vari elementi della formula, un assegnamento è detto modello, se la formula A è vera con un certo modello M allora diciamo che M è un modello per A.**

**Conseguenza logica: una formula A è conseguenza logica della KB se e solo se in ogni modello in cui è vera la KB è vera anche A.**

$$KB \models \alpha \text{ sse } M(KB) \subseteq M(\alpha)$$

Nella formula abbiamo:

- $M(A)$  = l'insieme delle interpretazioni che rendono vera la A
- $M(KB)$  = insieme di formule che sono un modello per KB

Due formule A e B sono equivalenti se B è conseguenza logica di A e se A è conseguenza logica di B.

Una formula A è **valida** se è vera con tutte le interpretazioni (tautologia) ed è **soddisfacibile** se è vera almeno in una interpretazione.

Per fare inferenza su una KB scritta in Prop si utilizza il Model Checking ovvero si crea una tabella di verità in cui proviamo tutti i possibili modelli (possibili assegnamenti di valori alle variabili che rende vera la formula). Una strategia per trovare una soluzione consiste nell'inserire nella KB anche il goal negato e dimostrare che questa formula non è soddisfacibile.

## **TT Entails**

Esiste anche l'algoritmo TT Entails per capire se una certa formula A è conseguenza logica della KB.



Come funziona:

- Enumero tutte le possibili interpretazioni della KB assegnando i possibili valori alle variabili (T o F)
- Se ottengo una interpretazione che soddisfa la KB allora controllo se soddisfa anche il goal, se non lo soddisfa allora non posso dire che il goal è conseguenza della KB. Se invece con ogni interpretazione che soddisfa KB ho anche che è soddisfatto il goal allora sappiamo che è conseguenza logica.

## Algoritmi SAT

In questi algoritmi si utilizza la KB scrivendola in forma a clausole, la forma a clausole consiste nello scrivere nelle parentesi graffe i simboli, divisi da una virgola che rappresenta l'OR. Poi se metto vicine vari gruppi di parentesi graffe, tra questi gruppi ci sono gli AND.

Se ho una formula che non è in forma a clausole devo farcela diventare e ci sono dei passi da seguire:

1. Eliminazione della  $\Leftrightarrow$ :  $(A \Leftrightarrow B) \equiv (A \Rightarrow B) \wedge (B \Rightarrow A)$
2. Eliminazione dell'  $\Rightarrow$ :  $(A \Rightarrow B) \equiv (\neg A \vee B)$
3. Negazioni all'interno:  
 $\neg(A \vee B) \equiv (\neg A \wedge \neg B)$  (de Morgan)  
 $\neg(A \wedge B) \equiv (\neg A \vee \neg B)$
4. Distribuzione di  $\vee$  su  $\wedge$ :  
 $(A \vee (B \wedge C)) \equiv (A \vee B) \wedge (A \vee C)$

Anche l'algoritmo SAT è per la soddisfacibilità di una certa formula e mi permette di dire che A è conseguenza logica della KB.

## DPLL

Altro algoritmo per la soddisfacibilità in cui la KB è espressa in forma a clausole. È un miglioramento rispetto a TT Entails, infatti enumera in profondità le possibili interpretazioni cercando un modello.

Come funziona:

- Si usa la terminazione anticipata che è una euristica per fare meno mosse, in particolare nella forma a clausole quando arrivo ad una formula in cui ho due simboli  $\{A, B\}$  e uno dei due è vero allora automaticamente quella formula tra le parentesi è vera indipendentemente dal valore di B. Se invece sia A che B sono negative allora tutta la formula sarà falsa e quindi quella interpretazione non è un modello
- Si usano i simboli puri: un simbolo puro compare con lo stesso segno in tutte le clausole, in particolare se una clausola è già verificata possiamo non considerare quel simbolo puro. Il simbolo puro viene assegnato a True o False (se non è negato o se è negato)
- Clausole unitarie: sono le clausole in cui ho un solo letterale o in cui ho più letterali ma gli altri hanno valore false. Quindi assegno valore true all'unico simbolo rimasto

In sat abbiamo da dimostrare sempre che  $KB \models A$  quindi quello che si fa è prendere A e inserirlo nella formula della KB assegnando anche valori ad A, se non arrivo ad una formula soddisfacibile allora vuol dire che non è conseguenza logica.

DPLL è completo e termina sempre.

## Metodi locali per SAT - WalkSat

- Partiamo con un assegnamento casuale di T e F
- Valuto lo stato in cui mi trovo contando il numero delle clausole che non sono soddisfatte
- Tra tutte le clausole che non sono soddisfatte ne scelgo una e prendo un simbolo all'interno della clausola, a questo simbolo gli cambio il valore e lo flippo da T a F o viceversa
- Il simbolo da flippare o lo scelgo casualmente oppure posso sceglierlo in modo da soddisfare più clausole possibili
- Se non riesco ad arrivare ad un assegnamento che soddisfi la formula mi fermo dopo un certo numero di flip

Non posso usare questo metodo per dire che un certo insieme di clausole non è soddisfacibile, l'algoritmo WalkSat non è completo però termina se ho un numero infinito di flip e ho un insieme di clausole soddisfacibili.

Walksat trova velocemente la soluzione in problemi con molti vincoli e in problemi con pochi vincoli, nel mezzo invece ha delle difficoltà.

### **Inferenza per dimostrare $KB \models A$**

Si tratta di un altro metodo per dimostrare la conseguenza logica, a partire dalla KB in questo caso devo dedurre A e questo lo scrivo in questo modo:  $KB \vdash A$ , questa deduzione di A dalla KB avviene con delle regole di inferenza.

- Correttezza: se deduco A da KB allora A è conseguenza logica della KB ( $KB \vdash A \Rightarrow KB \models A$ )
- Completezza: se A è conseguenza logica della KB allora è anche deducibile dalla KB ( $KB \models A \Rightarrow KB \vdash A$ )

Per l'inferenza si utilizzano le regole della logica come il modus ponens o l'eliminazione dell'And e della doppia implicazione:

Le regole sono schemi deduttivi del tipo:

$$\frac{\alpha \Rightarrow \beta, \quad \alpha}{\beta} \quad \begin{array}{l} \text{Modus ponens oppure} \\ \text{Eliminazione dell'implicazione} \end{array}$$

$$\frac{\alpha \wedge \beta}{\alpha} \quad \text{Eliminazione dell'AND}$$

*Eliminazione e introduzione della doppia implicazione*

$$\frac{\alpha \Leftrightarrow \beta}{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)} \quad \text{and} \quad \frac{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}{\alpha \Leftrightarrow \beta}$$

Decidere ad ogni passo la regola da applicare non è un problema semplice perchè se scelgo male potrei avere troppe combinazioni da considerare. Per questo motivo quello che si fa è una dimostrazione all'indietro ovvero partiamo da quello che vogliamo dimostrare e torniamo indietro cercando di dimostrare le premesse.

Le regole di deduzione sono regole di inferenza, meno regole ho e meglio è perchè la complessità del problema aumenta con l'aumentare delle regole.

In particolare nel PROP abbiamo la regola di risoluzione che consiste nell'eliminare i letterali di segno opposto e fare l'unione dei rimanenti:

$$\frac{\{P, Q\} \quad \{\neg P, R\}}{\{Q, R\}}$$

Se ho due letterali ad esempio A e B e nell'altra clausola ci sono sempre A e B ma con segno opposto, non posso eliminare tutto e arrivare a  $\{\}$ . Invece se ho  $\{A\}$  e  $\{\text{not } A\}$  allora posso arrivare a dire che ho  $\{\}$  perchè sono arrivato ad una contraddizione.

**Teorema di risoluzione:**

Se la KB non è soddisfacibile allora  $KB \models \{ \}$

**Teorema di refutazione:**

La  $KB \models A$  solamente se la  $(KB \cup \{\text{not } A\})$  non è soddisfacibile. In pratica aggiungo il goal negato all'interno della formula e vedo se raggiungo la clausola vuota. Se la raggiungo allora vuol dire che non è soddisfacibile.

# Settima Lezione

Agenti Logici: la logica del prim'ordine

La logica del prim'ordine viene espressa con il linguaggio FOL, anche in questo caso abbiamo un dominio del problema, poi abbiamo delle funzioni e delle relazioni tra gli elementi del dominio. In particolare la sintassi del linguaggio è la seguente:

- **Connettivo**  $\rightarrow \wedge \mid \vee \mid \neg \mid \Rightarrow \mid \Leftrightarrow \mid \Leftarrow$
- **Quantificatore**  $\rightarrow \forall \mid \exists$
- **Variabile**  $\rightarrow x \mid y \mid \dots a \mid s \dots$  (lettere minuscole)
- **Costante**  $\rightarrow$  Es.  $A \mid B \mid \dots$  Mario  $\mid$  Pippo  $\mid 2 \dots$
- **Funzione**  $\rightarrow$  Es.  $Hat \mid Padre-di \mid + \mid - \mid \dots$   
(con arità  $\geq 1$ )    1            1        2    2
- **Predicato**  $\rightarrow$  Es.  $On \mid Clear \mid \geq \mid < \dots$   
(con arità  $\geq 0$ )    2            1        2    2

Costanti (lettere maiuscole), variabili (lettere minuscole) e funzioni sono definite termini, in particolare le funzioni prendono come parametri una quantità di termini che dipende dalla arità della funzione stessa.

I termini del linguaggio devono corrispondere ad oggetti del mondo. Per questo viene creata una interpretazione, questa interpretazione serve per stabilire una corrispondenza tra gli elementi del linguaggio e gli elementi della concettualizzazione.

All'interno del linguaggio troviamo anche le formule, queste possono essere:

- Atomiche: ovvero formate da un solo elemento tra quelli elencati prima ad esempio true o un termine
- Complesse: formate da più formule unite tra loro ad esempio due formule in and o una implicazione

Notare la differenza tra variabili e costanti, le prime sono espresse con le lettere minuscole e le seconde con le lettere maiuscole.

Le variabili nel linguaggio possono essere libere o legate, se sono libere vuol dire che non vengono usate con un quantificatore, se invece sono legate vuol dire che vengono usate in un quantificatore.

Una formula che non contiene variabili libere viene detta chiusa, se invece ce ne sono allora la formula è aperta.

Data una formula composta o un termine, possiamo determinare il significato in base al valore dei suoi componenti.

La semantica:



- Il per ogni  $X$  viene utilizzato come un grande AND in cui mettiamo in and tutti i possibili valori, solitamente questo per ogni implica qualcosa
- L'esiste  $X$  invece è un grande OR e di solito si usa in AND con altri termini

Nella semantica del FOL dobbiamo andare a specificare la differenza tra oggetti distinti, questo non avviene nella semantica da database in cui se due oggetti hanno nomi diversi allora sono diversi tra loro.

Nella semantica da database poi non esistono altri oggetti oltre a quelli di cui si parla e se c'è qualcosa di cui non si parla allora è falso.

Regole di inferenza:

- La regola di inferenza per il per ogni mi permette di eliminare il per ogni andando a istanziare, ad esempio se ho per ogni  $A(x)$ , questo diventa  $A(b)$ , quindi sostituiscono  $x$  con un certo valore
- Per l'esistenziale si utilizza la skolemizzazione, in particolare posso utilizzare in presenza di un esistenziale la sostituzione con una variabile di skolem, invece se ho un esistenziale all'interno di un quantificatore universale allora devo usare una funzione di skolem.

$\exists x \text{ Padre}(x, G)$	diventa	$\text{Padre}(k, G)$ 
$\forall x \exists y \text{ Padre}(x, y)$ 	diventa	$\forall x \text{ Padre}(x, p(x))$
	e non	$\forall x \text{ Padre}(x, k)$

## Teorema di Herbrand

Se A è conseguenza logica della KB allora esiste una dimostrazione che coinvolge un sottoinsieme finito della KB proposizionalizzata.

Se A non è conseguenza logica il processo non termina.

Per fare questo dobbiamo prendere la KB e trasformarla andando a creare le istanze, prima quelle singole e poi quelle annidate.

Come si fa la risoluzione con il FOL?

Si può utilizzare un sistema molto simile a quello che abbiamo usato con il PROP aggiungendo però l'unificazione alla trasformazione in forma a clausole.

**Teorema:** data una formula A è possibile trovare un insieme di clausole che sono soddisfacibili se A era soddisfacibile e insoddisfacibili se A non era soddisfacibile.

**Unificazione:** con questa operazione si determina se due espressioni possono essere rese identiche tramite una sostituzione di termini a variabili.

Se due espressioni sono identiche dopo la sostituzione allora abbiamo l'unificatore altrimenti un fallimento.

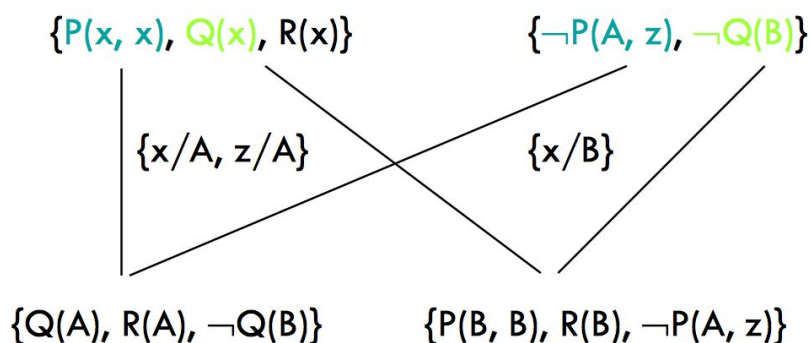
**Sostituzione:** associazione tra variabili e termini, ogni variabile compare solamente una volta, ad esempio  $\{x / A\}$  è la sostituzione della variabile A con il valore x.

Le sostituzioni tra variabili e valori vanno eseguite tutte in contemporanea.



Se una sostituzione rende identiche due espressioni allora vuol dire che le espressioni sono unificabili, possiamo creare vari unificatori, alcuni saranno più generali altri meno generali, ne esiste però solamente uno che è il più generale di tutti.

La risoluzione deve essere applicata in questo modo, quelli che ottengo sono dei fattori e posso unificarli per ottenere la clausola vuota (quando possibile).



Questo metodo di risoluzione è corretto ma non è completo, la correttezza in particolare è garantita da:

**Correttezza:** Se  $\Gamma \vdash_{\text{RES}} A$  allora  $\Gamma \models A$

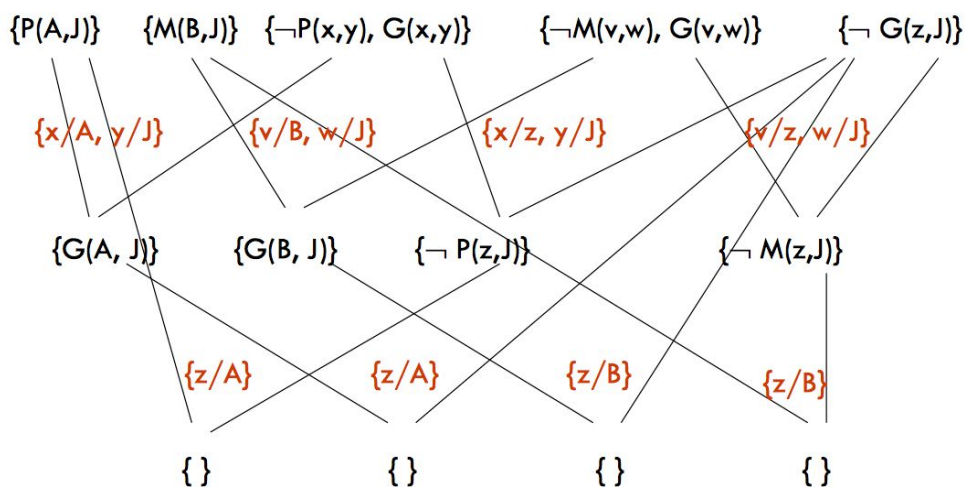
**Refutazione:** un metodo alternativo alla risoluzione standard che non è completa è la risoluzione per refutazione, questo metodo è completo. Se la formula unita con il goal negato è insoddisfacibile allora vuol dire che il goal è conseguenza logica della formula.

Con la refutazione posso anche rispondere alle domande del tipo “chi è il figlio di...?”.

Per farlo devo andare a trovare tutti i possibili assegnamenti della variabile (che voglio scoprire) che mi portano alla clausola vuota.

In questo caso voglio sapere chi sono i genitori di J, quindi cerco tutti gli assegnamenti di z che portano alla clausola vuota.

Notare che date due clausole, non posso sostituire x con A e y con A, i due valori che assegno devono essere differenti tra loro’.



Le risposte sono: A, B

# Ottava Lezione

## Introduzione

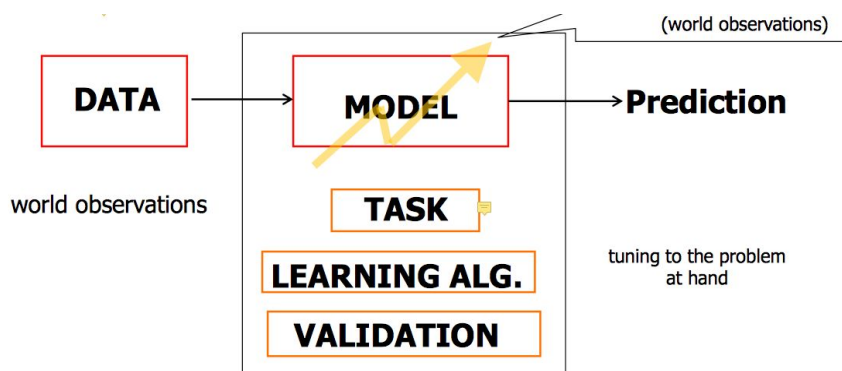
Il machine learning è una nuova area di ricerca che combina l'idea di creare computer che possono imparare con nuovi tool statistici adattivi.

Si vogliono creare sistemi intelligenti o creare sistemi in grado di predire, in generale si affrontano problemi per cui non abbiamo una soluzione ma per cui è facile trovare una sorgente di esperienza.

Per il machine learning è necessario avere un set di dati che devono essere rappresentativi e significativi per il fenomeno. In alcuni contesti è più complicato avere dei dati che possono essere significativi come nella medicina.

Il risultato che otteniamo con il machine learning può anche essere una approssimazione e non un risultato preciso, in alcuni casi questo può andare bene, in altri campi (medicina) invece non va bene.

Avere delle soluzioni approssimate non vuol dire che il metodo è approssimato, il metodo è rigoroso e preciso ma trova soluzioni approssimate a problemi complessi.



Il sistema predittivo è formato da tutti questi elementi, abbiamo a disposizione dei dati che sono le osservazioni del mondo, poi abbiamo il modello che in accordo al sistema di validazione e all'algoritmo di learning è in grado di fornire una predizione.

In questo sistema la parte soggetta a cambiamenti è la parte del model che deve essere modificata in base al problema che si deve affrontare.

**Modello:** il modello ha lo scopo di catturare le relazioni tra i dati, il modello definisce una classe di funzioni che possono essere implementate, questo è chiamato lo spazio delle ipotesi.

**Task Supervisionato:** in questo caso abbiamo a disposizione dei dati di ingresso del tipo <input,output> che qualcuno ha etichettato, l'uscita è la risposta che deve dare il modello quando prende in input quel dato.

L'obiettivo è trovare una approssimazione della funzione  $F$  che mi fornisce un risultato nel momento in cui do in input un nuovo valore.

La funzione  $F$  può funzionare in due modi:

- Classificazione: nel caso della classificazione abbiamo la funzione  $F$  che mi restituisce una classe di appartenenza (categoria) per quel particolare input
- Regressione: nella regressione invece dato il dato in input si restituisce un valore numerico

Nel caso del task supervisionato gli esempi di training sono nella forma  $\langle x, f(x) \rangle$  e la funzione target è la  $f$ . Quello che vogliamo fare è trovare una funzione  $h$  che approssima il funzionamento della funzione  $f$ . La funzione  $h$  può essere espressa con i linguaggi della logica o con equazioni numeriche.

In questo caso lo spazio delle ipotesi è l'insieme delle ipotesi che possono essere output dell'algoritmo di learning.

**Task non supervisionato:** non abbiamo i dati già etichettati abbiamo solamente un insieme di dati in input  $\langle X \rangle$  che sono quelli su cui vogliamo fare apprendimento.

Quello che si fa è raggruppare i dati e quindi c'è il problema del clustering in cui cerchiamo di raggruppare i dati secondo la metrica euclidea, all'interno del gruppo abbiamo un centroide che mi identifica tutto il gruppo.

Con il raggruppamento riusciamo a capire anche se ci sono dati che riusciamo a comprendere senza avere necessità di etichettarli.

**Algoritmo di learning:** basandosi sui dati, sul task e sul modello si fa una ricerca euristica all'interno dello spazio delle ipotesi  $H$  e si trova la migliore approssimazione della funzione target che al momento non conosciamo.

Si cerca una buona funzione all'interno dello spazio delle funzioni partendo da dati conosciuti.

Si deve utilizzare una euristica per cercare nello spazio delle ipotesi perchè non posso fare una ricerca esaustiva di tutte le possibili funzioni.

**Errore di generalizzazione:** indica quanto accuratamente il modello predice soluzioni nel momento in cui arrivano nuovi dati.

La generalizzazione è un punto fondamentale del machine learning perchè è difficile capire se un modello generalizza bene o generalizza male.

La performance del machine learning si misura in base all'accuratezza nella previsione delle successive risposte ai dati che inseriremo.

**Calcolo delle derivate parziali:** La derivata parziale di una funzione rispetto ad una variabile  $X$  si calcola derivando la funzione considerando solamente quella variabile  $X$  e considerando quindi le altre variabili come costanti.

**Gradiente:** quando una funzione con due variabili  $f(x,y)$  ha delle derivate parziali nei due punti  $(x,y)$  allora possiamo associare il vettore delle due derivate parziali  $(\frac{df}{dx}, \frac{df}{dy})$  e questo è chiamato gradiente.

Il gradiente è quindi un vettore che mi punta nella direzione del “pendio più ripido”.  
Nei minimi locali il gradiente vale 0.

## Nona Lezione

### Concept Learning

L’idea del learning consiste nel migliorare l’esecuzione di alcuni task con l’esperienza.

**Concept Learning:** si deve inferire una funzione booleana a partire da esempi di training positivi e negativi.

In generale l’insieme  $H$  che rappresenta lo spazio delle ipotesi è grande  $2^{2^n}$ , dato che è un numero esagerato allora si considera uno spazio delle ipotesi decisamente più piccolo.

Per semplificare possiamo utilizzare delle regole congiuntive in un insieme finito e discreto  $H$  e delle funzioni lineari in un insieme infinito e continuo  $H$ .

Per organizzare la ricerca nello spazio delle ipotesi esistono alcuni algoritmi efficienti come le regole congiuntive.

Ad esempio data la tabella con i vari attributi si deve andare a fare una previsione:

Sky	Temp	Humid	Wind	Water	Fore- cast	Enjoy Sport
Sunny	Warm	Normal	Strong	Warm	Same	Yes
Sunny	Warm	High	Strong	Warm	Same	Yes
Rainy	Cold	High	Strong	Warm	Change	No
Sunny	Warm	High	Strong	Cool	Change	Yes

instance

In questo caso vogliamo una funzione EnjoySport che mi dice in quali giorni posso andare in acqua e quando invece non posso. Questo set di dati viene descritto come una congiunzione di vincoli.

Per ogni valore della tabella creiamo una rappresentazione del tipo  $\langle 1, 0, ?, \dots \rangle$ .

Per ogni attributo possiamo scrivere:

- ? se ogni valore è accettabile per questo attributo
- Possiamo specificare un valore preciso
- Possiamo scrivere 0 per indicare che nessun valore è accettabile

**Dati:** istanza  $X$  che mi rappresenta un possibile giorno con il corrispondente valore di enjoy sport

**Funzione target:** funzione  $c$  tale che  $c(x) \rightarrow \{0, 1\}$

**Spazio delle ipotesi:** un set di letterali  $\langle \text{soleggiato}, ?, ?, \text{forte}, ? \rangle$

Si assume che ogni funzione che approssima correttamente la funzione target e che quindi mi fornisce una soluzione corretta con gli esempi di training, funzionerà correttamente anche con i successivi esempi non ancora osservati.

Dizionario:

- L'insieme delle istanze su cui viene definito il problema viene indicato con  $X$ , nell'esempio sopra la  $X$  è l'insieme delle varie condizioni atmosferiche nei vari giorni
- Il concept (funzione da imparare) è chiamata target concept e viene indicata con  $c$   
l'unica informazione che abbiamo riguardante  $c$  è il valore che assume quando viene utilizzata sugli esempi di training.
- Spazio delle ipotesi  $H$ : insieme di tutte le possibili funzioni che possono essere realizzate dal sistema di apprendimento, ognuna è  $h$  e rappresenta una funzione definita su  $X$  tale che mi restituirà 0 o 1.  $H$  non può coincidere con tutte le funzioni possibili e la ricerca non può essere esaustiva.
- Esempi di training: una istanza  $x$  appartenente a  $X$ , gli esempi positivi sono quelli in cui  $c(x) = 1$ , i negativi  $c(x) = 0$

L'obiettivo è trovare una ipotesi  $h$  appartenente a  $H$  tale che  $h(x) = c(x)$  per ogni  $x$  in  $X$ .

Ogni ipotesi  $h$  che approssima la funzione target  $c$  su un insieme sufficientemente largo di  $x$  appartenenti a  $X$ , approssimerà anche la funzione target su esempi non ancora osservati in modo sufficiente.

## Concept Learning come ricerca

Il concept learning può essere visto come una ricerca all'interno dello spazio delle ipotesi con l'obiettivo di trovare l'ipotesi che meglio fitta gli esempi di training.

Il problema è che lo spazio delle ipotesi è enorme e quindi si devono trovare degli algoritmi che cercano in modo intelligente all'interno di  $H$  in modo da trovare una  $h$  utilizzabile.

### **Definizione di funzione booleana più generale:**

Molti algoritmi per il concept learning cercano di ricercare in  $H$  mediante l'organizzazione delle ipotesi dalla più generale alla più specifica.

Una ipotesi  $h$  che classifica più istanze come vere rispetto ad una ipotesi che invece ha meno ?.

Date le funzioni booleane  $h_1$  e  $h_2$ ,  $h_1$  è più generale di  $h_2$  (ovvero  $h_1 > h_2$ ) se e solo se per ogni  $x$  appartenente a  $X$  allora  $(h_2(x) = 1) \rightarrow (h_1(x) = 1)$

Esempio:

$h_1 = \langle \text{Sunny}, ?, ?, \text{Strong}, ?, ? \rangle$

$h_2 = \langle \text{Sunny}, ?, ?, ?, ?, ? \rangle$

In questo caso ad esempio  $h_2$  è più generico di  $h_1$  perchè  $h_2$  classifica come positive più istanze rispetto ad  $h_1$ .

## Algoritmo Find-S:

Si tratta di un algoritmo per trovare una  $h$  consistente senza però enumerare tutte le  $h$  presenti nell'insieme  $H$ .

- Prima di tutto inizializziamo  $h$  come più specifico possibile, un esempio è una istanza con tutti 0.
- Per ogni istanza di training positiva  $X$ :
  - Per ogni attributo  $a_i$  di  $h$ 
    - se  $a_i$  è soddisfatto da  $x$  non fare niente
    - Altrimenti sostituisci  $a_i$  con il valore più generale che è soddisfatto da  $X$ , ad esempio possiamo anche rimuovere  $a_i$  sostituendolo con  $?$  che è più generale

```
x1=<Sunny,Warm,Normal,Strong,Warm,Same>+ h0=<∅,∅,∅,∅,∅,∅,>
x2=<Sunny,Warm,High,Strong,Warm,Same>+ h1=<Sunny,Warm,Normal,
Strong,Warm,Same>
x3=<Rainy,Cold,High,Strong,Warm,Change>- h2,3=<Sunny,Warm,?,
Strong,Warm,Same>
x4=<Sunny,Warm,High,Strong,Cool,Change>+ h4=<Sunny,Warm,?,
Strong,?,?>
```

All'inizio la  $h$  è troppo specifica, la generalizziamo e quindi abbiamo una  $h$  un po' più generale ma comunque sono fissati tutti gli attributi quindi mi dice che solamente le istanze fatte in quel modo sono positive.

Va generalizzata ancora quindi confrontiamo con gli altri esempi di training positivi e generalizziamo mettendo  $?$  ad alcuni attributi che non sono soddisfatti dal nuovo esempio.

Alla fine della procedura abbiamo la  $h$  che è l'output.

Find-S ignora tutti gli esempi di training negativi ma comunque la  $h$  è consistente con gli esempi negativi senza bisogno di fare altri controlli. Ad ogni passaggio l'ipotesi  $h$  è quella più specifica consistente con gli esempi di training.

Problemi:



- Non sappiamo se abbiamo trovato l'unica ipotesi consistente con i dati e se ce ne sono altre
- Non sappiamo se è meglio trovare l'ipotesi più specifica o se invece trovarne una che è meno specifica e quindi un po' più generale

## Candidate elimination algorithm

L'idea dell'algoritmo candidate elimination è quella di mandare in output l'insieme di tutte le ipotesi che sono consistenti con gli esempi di training, non enumera tutte le possibili ipotesi ma utilizza l'ordinamento "più generale di".

Definizione di ipotesi consistente:

Una ipotesi  $h$  è consistente con un set di esempi di training  $D$  se e solo se  $h(x) = c(x)$   
per ogni esempio  $\langle x, c(x) \rangle$  in  $D$

L'algoritmo candidate Elimination mi da in output il set delle ipotesi consistenti con tutti gli esempi di training, questo insieme di ipotesi è chiamato Version Space.

Version Space:

Il version space che rispetta lo spazio delle ipotesi  $H$  e gli esempi di training  $D$ , è il sottoinsieme delle ipotesi  $h$  di  $H$  consistenti con gli esempi di training in  $D$ .

Un modo per rappresentare il version space è utilizzare l'algoritmo List-Then-Eliminate che parte con un version space che contiene tutte le ipotesi di  $H$  poi elimina tutte le ipotesi che sono trovate inconsistenti con gli esempi di training. Se non riusciamo a trovare solamente una  $h$

all'interno del version space, allora verrà dato in output tutto il contenuto del version space.

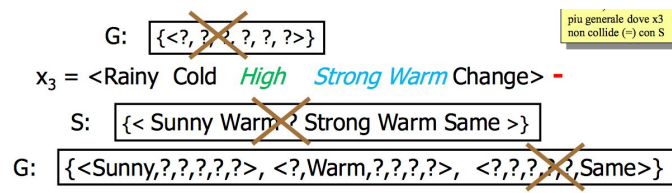
Un modo migliore per rappresentare il version space è utilizzare l'algoritmo Candidate-Elimination, in questo caso il version space è rappresentato con l'elemento più generale e con l'elemento più specifico che formano i confini del version space ordinato.

L'algoritmo Candidate-Elimination Learning:

Partiamo con  $G$  che è l'insieme delle ipotesi più generali e con  $S$  che è l'insieme delle ipotesi più specifiche,

L'algoritmo dice questo:

- Per ogni esempio  $d$  di training positivo:
  - Rimuoviamo da  $G$  tutte le ipotesi che non sono consistenti con  $d$
  - Generalizziamo  $S$ , se ci sono delle ipotesi che non sono consistenti andiamo ad eliminarle ed aggiungiamo al suo posto una ipotesi più generale che deve essere consistente con  $d$  e che creiamo da quella eliminata.
  - Se all'interno di  $S$  ora ci sono ipotesi più generali di altre, quelle più generali le eliminiamo
- Per ogni esempio  $d$  di training negativo:
  - Rimuoviamo da  $S$  le ipotesi che non sono consistenti con  $d$
  - Per ogni ipotesi in  $G$  che non è consistente con  $d$ :
    - Rimuoviamo l'ipotesi da  $G$  e poi aggiungiamo a  $G$  una ipotesi più specifica che deve essere consistente con  $d$ , per essere consistente con  $d$  prendiamo il negato di quello che troviamo in  $d$



- Rimuoviamo da G le ipotesi che sono meno generali di altre ipotesi di G

## Inductive Bias

L'algoritmo Candidate Elimination converge verso un target concept corretto, ma cosa succede se invece il target concept non è contenuto nello spazio delle ipotesi? Possiamo utilizzare uno spazio delle ipotesi che le include veramente tutte? Quanto crescerebbe lo spazio delle ipotesi e in che modo questa crescita influenzerebbe il numero di esempi di training da osservare?

## Uno spazio delle ipotesi parziale (Biased)

Nel caso dell'algoritmo candidate elimination abbiamo creato lo spazio delle ipotesi permettendo solamente una congiunzione degli attributi, ad esempio non permettiamo di scrivere che il cielo può essere soleggiato o nuvoloso ma solamente uno dei due.

Example	Sky	AirTemp	Humidity	Wind	Water	Forecast	EnjoySport
1	Sunny	Warm	Normal	Strong	Cool	Change	Yes
2	Cloudy	Warm	Normal	Strong	Cool	Change	Yes
3	Rainy	Warm	Normal	Strong	Cool	Change	No

Con questo metodo c'è un problema perchè con un insieme di esempi di training come il precedente, la h che viene trovata non sarà consistente con il terzo esempio ma solamente con i primi due e quindi non avremo niente all'interno del version space.

Sarebbe quindi necessario uno spazio delle ipotesi che includa anche espressioni disgiuntive e quindi un “learner” che non deve essere indotto a considerare solamente ipotesi congiuntive.

Una soluzione al problema potrebbe essere quella di creare uno spazio delle ipotesi capace di rappresentare tutti i concetti che possono essere imparati, questo insieme delle ipotesi sarebbe l'insieme di tutti i possibili sottoinsiemi di  $X$  viene chiamato Power Set.

Avremmo in questo caso due ipotesi unite con un OR per accettare i primi due esempi.  $\langle \text{Sunny}, ?, ?, ?, ?, ? \rangle \vee \langle \text{Cloudy}, ?, ?, ?, ?, ? \rangle$

Praticamente utilizzando il Candidate Elimination algorithm si andrebbe a creare un version space in cui il limite  $S$  sarebbe la disgiunzione di tutti gli esempi positivi mentre il limite  $G$  sarebbe la disgiunzione degli esempi negativi negati.

Questo vuol dire che i soli esempi di training che verrebbero classificati sono gli esempi di training stessi, Per produrre un solo target concept finale bisogna presentare tutte le istanze di  $X$  come un esempio di training.

Per le istanze che non sono state osservate però ho un problema perchè queste verranno caratterizzate per metà come positive e per metà come negative.

Quindi un learner “unbiased” non è in grado di generalizzare e sono necessarie delle assunzioni a priori per poter far funzionare l'algoritmo candidate elimination e farlo anche generalizzare.

Il bias (vincolo) quindi può essere l'assunzione che il target concept può essere rappresentato da una congiunzione, questo in alcuni casi potrà una classificazione di alcune istanze come positive anche se sono negative.

E' quindi necessario una forma di assunzione a priori o anche detta inductive bias.

L'idea chiave è quella di permettere al learner di inferire la classificazione di nuove istanze:

- Abbiamo un algoritmo di learning  $L$
- Abbiamo un set di dati di training  $D_c = \{x, c(x)\}$
- Abbiamo una nuova istanza  $x_i$

Quello che vogliamo fare è classificare  $x_i$  ovvero:

$$(D_c \text{ and } x_i) \rightarrow L(x_i, D_c)$$

La parte a destra viene inferita induttivamente da quella a sinistra.

### Definizione di Inductive Bias

L'inductive Bias di  $L$  (algoritmo) è un qualsiasi insieme di asserzioni  $B$  tale che per ogni target concept  $c$ , e un corrispondente insieme di esempi di training  $D_c$  si ha:

$$(\forall x_i \in X)[B \wedge D_c \wedge x_i] \vdash L(x_i, D_c)$$

L'inductive Bias dell'algoritmo candidate-elimination è che il target concept  $c$  è contenuto nello spazio delle ipotesi  $H$ .

L'inductive bias viene quindi utilizzato perchè nell'insieme  $H$  non posso fare una ricerca esaustiva in modo da filtrare le funzioni.

Altri esempi di algoritmi con il loro bias (vincolo iniziale??):

- Rote learner (lookup table): Store examples, classify  $x$  if and only if it matches a previously observed example.
  - No inductive bias  $\rightarrow$  no generalization
- Version space candidate elimination algorithm.
  - Bias: The hypothesis space contains the target concept (conj. of attributes)  $|H| = 973$  versus  $10^{28}$ .
- Find-S
  - Bias: The hypothesis space contains the target concept and all instances are negative instances unless the opposite is entailed by its other knowledge (seen as positive examples).

# Decima Lezione

## Regressione

Processo che consiste nello stimare una funzione basandosi su un insieme di dati.

### Regressione lineare univariata

Consideriamo il caso della regressione lineare univariata, abbiamo una variabile in input ( $x$ ) e una variabile in output ( $y$ ).

Il fatto che il modello sia lineare è una cosa positiva perchè è molto semplice e perchè tutte le informazioni e i dati sono in  $w$ , è anche semplice da interpretare e possono essere presenti dei dati con errori.

Il modello  $h(x)$  viene espresso come  $out = w_1 * x + w_0$  dove i  $w$  sono detti pesi.

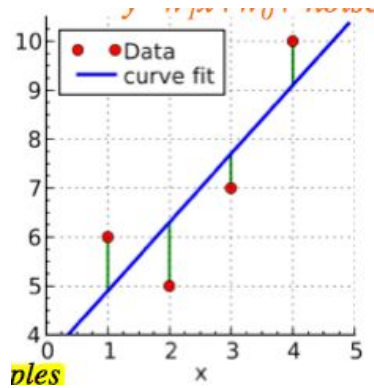
In queste definizioni assumiamo che la variabile  $y$  sia legata linearmente con  $x$  tramite la relazione  $y = w_1 * x + w_0 + noise$  dove con noise intendiamo l'errore che possiamo avere.

In questo caso otteniamo una linea retta, per fittare la linea con i dati che abbiamo a disposizione, dobbiamo trovare i corretti valori per i pesi.

Lo spazio dei pesi in questo caso è a due dimensioni.

Metodo dei minimi quadrati (**Least mean square LMS**): l'obiettivo è trovare una  $W$  che minimizzi la somma residua dei quadrati per fare in modo che la linea blu sia vicina il più possibile ai punti rossi.

In pratica apprendere la funzione è diventato trovare i pesi che minimizzano l'errore.



Nel disegno gli errori sono le distanze tra la linea blu e i punti rossi. La loss quindi è calcolata come la sommatoria su tutti i pattern (detti anche esempi) della distanza tra il punto  $y_p$  effettivo e il punto che sta sulla linea blu  $h(x_p)$  (calcolato come  $w_1 \cdot x_p + w_0$ ):

$$Loss(h_w) = E(w) = \sum_{p=1}^l (y_p - h_w(x_p))^2 = \sum_{p=1}^l (y_p - (w_1 x_p + w_0))^2$$

$\rightarrow$  pattern/example

Se vogliamo la media dell'errore, basta dividere questa sommatoria per  $n$ .

Come facciamo a capire che abbiamo trovato una soluzione che può essere utilizzata? Ci dobbiamo ricordare che il gradiente nei punti stazionari è nullo.

Quindi prima calcoliamo la loss  $E(w)$  e poi calcoliamo la derivata rispetto a  $w_0$  e rispetto a  $w_1$  ovvero il gradiente per ogni pattern  $p$ .

Quello che si fa è sfruttare il gradiente che mi quantifica lo spostamento ma non mi fa una ricerca tra tutti i possibili successori.

Costruiamo un algoritmo iterativo, partiamo con un qualsiasi  $W$  ed eseguiamo una ricerca locale muovendoci verso il minimo seguendo il gradiente che decresce.

Dato un certo  $W_0$ , per trovare il nuovo  $W_1$  che utilizzerò calcolo:

$$W_1 = W_0 + \text{frazione} * \text{gradiente}$$

Gradiente decrescente:

Questo metodo è un metodo che va a correggere l'errore, è chiamato regola del delta e cambia la W in modo proporzionale all'errore.

In questo caso le due derivate che vengono calcolate rispetto a  $w_0$  e  $w_1$  devono avere un segno negativo davanti.

$$\Delta w_0 = -\frac{\partial E(\mathbf{w})}{\partial w_0} = 2(y - h_{\mathbf{w}}(x)) \quad \Delta w_1 = -\frac{\partial E(\mathbf{w})}{\partial w_1} = 2(y - h_{\mathbf{w}}(x)) \cdot x$$

Ci sono 3 possibilità:

- Caso in cui abbiamo che la differenza tra il target e l'output è 0, in questo caso l'errore è 0 e quindi non ho errori
- Caso in cui l'output è maggiore del target, in questo caso l'output è troppo alto e stiamo dando un eccesso di valutazione.

Quindi qua la derivata rispetto a  $w_0$  è negativa e questo mi porta a ridurre  $w_0$  e se la  $x$  in input è maggiore di 0 con la derivata rispetto a  $w_1$  negativa allora riduco anche  $w_1$ , altrimenti lo aumento

- Output > target  $\rightarrow (y-h) < 0$  (output is too high)
  - $\rightarrow \Delta w_0$  negative  $\rightarrow$  reduce  $w_0$  and
  - if (input  $x > 0$ )  $\Delta w_1$  negative  $\rightarrow$  reduce  $w_1$  (else increase  $w_1$ )

- La terza possibilità è che l'output è troppo basso e quindi ho output < target.

Per la modifica della W se devo considerare I pattern differenti:

- **Algoritmo batch**: calcoliamo la sommatoria dell'errore quadratico su I pattern diversi e quindi abbiamo il gradiente, dopo I dati di training sommati abbiamo una “epoca” e aggiorniamo la W
- **On-line**: calcoliamo il gradiente su un pattern p e poi aggiorniamo subito w.



## Caso multidimensionale

Fino ad ora abbiamo considerato solamente il caso in cui abbiamo una variabile in ingresso  $x$  e una in uscita  $y$  (esempio del prezzo delle case  $x$  = dimensione della casa,  $y$  = prezzo stimato).

Bisogna però considerare anche il caso in cui abbiamo più di una sola variabile, in questo caso  $X$  diventa un vettore di  $x_i$  (dove le  $x_i$  potrebbero essere nell'esempio delle case, il numero delle stanze o l'età della casa). Se  $X$  è diventato un vettore di variabili, anche  $W$  diventa un vettore di pesi e quindi il calcolo della  $h(X)$  viene modificato e viene ora utilizzato il vettore trasposto delle  $W$  per il vettore delle  $X$ :

$$\mathbf{W}^T \mathbf{X} + w_0 = w_0 + w_1 x_1 + w_2 x_2 + \dots + w_n x_n$$

In questo caso  $w_0$  è l'offset e spesso è uguale alla costante 1.

Un'equazione di questo genere con la  $X$  e la  $W$  che sono dei vettori necessita di una rappresentazione geometrica su un iperpiano, quindi fino a che sono due piani è anche visibile ma poi non riusciamo più a visualizzarlo.

Dato un set di esempi di training dove ogni esempio è fatto così:  $(x_p, y_p)$  con  $x_p$  vettore allora per calcolare la  $W$  che minimizza la loss devo calcolare:

$$E(\mathbf{w}) = \sum_{p=1}^l (y_p - \mathbf{x}_p^T \mathbf{w})^2 = \|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2$$

### L'algoritmo del gradiente decrescente:

Si tratta dell'algoritmo che mi fa trovare i  $w$  da utilizzare nella mia funzione in modo da ottenere una funzione che fitti i dati il più possibile.

- Partiamo con una certa  $w$  e settiamo un valore di  $\eta$ , è frazionario compreso tra 0 e 1

- Calcoliamo il gradiente  $E(w)$ , questo calcolo va fatto calcolando la derivata parziale di  $E(w)$  per ogni  $w_i$  che abbiamo a disposizione.
- Ora per ogni  $w_i$  dobbiamo calcolare il nuovo  $w$ , questo è dato dal vecchio  $w$  + la frazione che abbiamo scelto per il gradiente che abbiamo calcolato:

$$w_{\text{new}} = w_{\text{old}} + \text{eta} * \Delta w$$

- Ora calcoliamo di nuovo  $E(w)$  utilizzando però i nuovi pesi che sono stati calcolati. Se la convergenza di  $E(w)$  è sufficientemente piccola allora mi fermo, altrimenti continuo cercando una nuova  $W$ .

## Problemi non lineari, generalizzazione per la regressione: Linear Basis Expansion

Non tutti i problemi possono essere affrontati con un modello lineare, potremmo avere dei casi in cui non mi basta una semplice retta per approssimare la funzione che cerco e per fittare i dati, in questo caso dovremmo passare all'utilizzo di un modello non lineare che quindi comporta un aumento del grado del polinomio.

L'alternativa è utilizzare comunque un modello lineare andando però a "vedere" i dati in uno spazio che non è quello originale ma uno più complesso in cui mi basta una funzione lineare per approssimare la funzione che cerco.

Se per esempio sono in  $R^2$  e non mi basta una funzione lineare per fittare i dati, passo in  $R^3$  dove magari riesco a fittare i dati con una retta semplice.

Questa trasformazione è chiamata linear basis expansion:

$$h_w(\mathbf{x}) = \sum_{k=0}^K w_k \phi_k(\mathbf{x})$$

Dove la funzione  $\phi(x)$  serve per complicare il piano in cui mi trovo e rendere lineare il problema che sarebbe invece non lineare.

$$(\phi: \mathbb{R}^n \rightarrow \mathbb{R})$$

EXAMPLES:

■ [1-dim  $x$ ]  $\phi_j(x) = x^j$ .

$$h(x) = w_0 + w_1x + w_2x^2 + \dots + w_Mx^M = \sum_{j=0}^M w_jx^j$$

1-dim polynomial regression ( $K=M$ )

■ Or “any other”, e.g.  $\phi(x) = \phi([x_1, x_2, x_3])$

$$h(x) = w_1x_1 + w_2x_2 + w_3\log(x_2) + w_4\log(x_3) + w_5(x_2x_3) + w_0$$

Il numero dei parametri è maggiore di  $n$ .

La linear basis expansion può essere più complicata rispetto al semplice modello lineare e questo comporta la necessità di avere dei metodi per trattare la complessità del modello perchè se cresce troppo ci sono dei problemi.

Se aumenta troppo la complessità ( $M=9$ ) abbiamo overfitting, i dati vengono fittati troppo bene e abbiamo  $E(w) = 0$ , in questo caso il modello segue tutti i rumori e abbiamo una varianza alta.

Al contrario con  $M=1$  abbiamo underfitting, il modello non fitta bene i dati e la soluzione è troppo vincolata.

Serve un bilanciamento tra varianza e vincoli e questo si ottiene con un metodo che è chiamato regolarizzazione e che mi evita l'overfitting, tra le varie possibilità preferiamo la soluzione più semplice che fitta i dati.

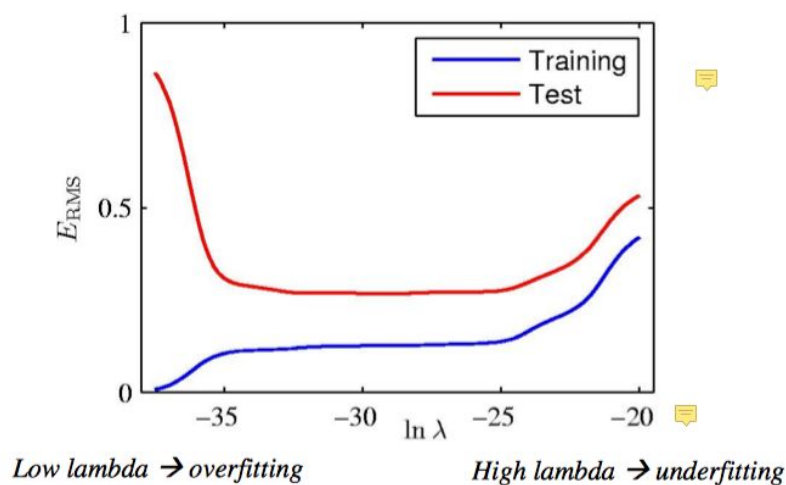
**Regolarizzazione di Tikhonov:** chiamata anche Ridge regression, è un metodo che mi permette di mantenere il controllo sulla funzione aggiungendo dei vincoli sui parametri  $w$  in modo da favorire un modello con meno termini.

Calcoliamo l'errore sui dati e lo sommiamo ad un termine penalty che deve essere calcolato in base a  $w$  moltiplicato per  $\lambda$ .

$$E(\mathbf{w}) = \sum_p (y_p - h_{\mathbf{w}}(\mathbf{x}_p))^2 + \lambda \|\mathbf{w}\|^2 \quad \text{coefficient} \quad \sum w_i^2$$

Il termine lambda è il coefficiente di regolarizzazione, il valore di questo lambda mi dice quanto peso deve avere la regolarizzazione.

Se il lambda è molto alto stiamo dando importanza alla regolarizzazione, se invece è basso lasciamo la possibilità di considerare l'errore con un potenziale overfitting.



Il calcolo dei pesi da utilizzare quindi varia e viene aggiunto anche qua il lambda.

$$\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} + \text{eta} * \Delta \mathbf{w} - 2 \lambda \mathbf{w}_{\text{old}}$$

## Classificazione

Lo stesso modello usato per la regressione può essere utilizzato anche per la classificazione ovvero, dato un certo dato in input, dobbiamo classificarlo (true/false o cose simili).

In questo caso utilizziamo un iperpiano in cui abbiamo una divisione tra la zona positiva e la zona negativa.

Il punto  $x$  che sta nell'iperpiano viene classificato in base alla posizione all'interno dell'iperpiano (zona positiva o zona negativa).

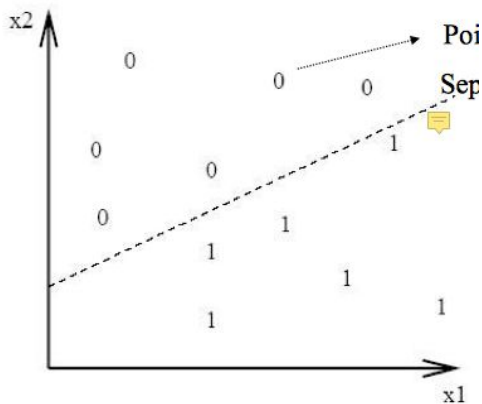
Nell'iperpiano la linea che divide la parte dove ci sono i valori positivi da quella dove ci sono i valori negativi è detta decision boundary e qui il valore dell'iperpiano è uguale a 0.

Per calcolare il valore del punto all'interno dell'iperpiano devo calcolare la moltiplicazione tra il vettore dei pesi  $W$  trasposto per il vettore degli  $X$ . Anche in questo caso va calcolato il peso  $w$  in modo da ottenere una accuracy accettabile.

La funzione  $h(x)$  per l'iperpiano restituisce 1 o 0 a seconda della posizione nel piano, per il calcolo di  $h(x)$  non mi porto dietro tutto l'iperpiano ma solamente il decision boundary. In pratica per dare il valore 1 o 0 viene utilizzata la funzione segno su  $wx + w_0$ .

$$h(x) = \text{sign}(wx + w_0)$$

Un esempio con uno spazio in due dimensioni:



In questo caso abbiamo la linea tratteggiata che mi fa da divisore tra la classe 0 e la classe 1, la linea tratteggiata è quella in cui  $wx + w_0$  vale 0.

$$\mathbf{w}^T \mathbf{x} = w_1 x_1 + w_2 x_2 + w_0 = 0$$

*Decision boundary* Def

Anche per i problemi di classificazione dato un set di esempi di training possiamo trovare una  $W$  che minimizzi  $E(w)$ , in questo caso nella formula non inseriamo  $h(w)$  perchè per il caso della classificazione equivale all'equazione segno e non funzionerebbe come nel caso della regressione.

La formula da utilizzare per calcolare  $E(w)$  è quindi:

$$\Delta w_i = -\frac{\partial E(\mathbf{w})}{\partial w_i} = \sum_{p=1}^l (y_p - \mathbf{x}_p^T \mathbf{w}) \cdot x_{p,i}$$

E possiamo applicare anche in questo caso l'algoritmo iterativo per il calcolo del gradiente in modo da trovare la  $w$  che mi vada a diminuire l'errore per fittare meglio i dati.

Il training del modello viene fatto dato un set di esempi utilizzando LSM e cercando un valore di  $w$  adeguato, poi successivamente il modello viene utilizzato per la classificazione applicando la funzione  $h(x) = \text{segno}(wx)$ .

L'errore in questo caso è rappresentato dal numero di pattern che non vengono classificati nel modo corretto dal modello.

Limitazioni:

Due insiemi di punti in un sistema a due dimensioni sono **separabili linearmente** quando i due insiemi di punti possono essere separati da una singola linea. In uno spazio con  $n$  dimensioni, per fare una separazione lineare sarebbe necessario avere un iperpiano divisore di  $n-1$  dimensioni.

La limitazione quindi è legata al fatto di avere una separazione lineare dei punti.

**Conclusioni:**

Abbiamo un metodo ben fondato per la regressione e la classificazione e un modo per rappresentare la conoscenza con grandi assunzioni per quel che riguarda le relazioni tra i dati.

Abbiamo a disposizione l'algoritmo LMS per la correzione dell'errore.

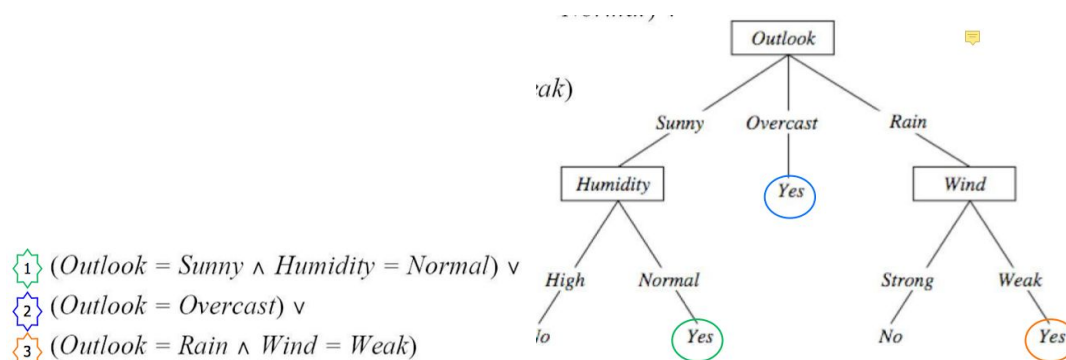
Quando arrivo ad avere modelli più flessibili vengono inseriti dei controlli sulla complessità come la regolarizzazione.

# Undicesima Lezione

## Introduzione agli alberi di decisione

Il metodo degli alberi di decisione è un metodo per approssimare le funzioni in cui la funzione “imparata” è rappresentata da un albero di decisione. Questi alberi possono anche essere rappresentati come una regola if then.

In pratica noi abbiamo un training set iniziale che mi fornisce delle risposte basandosi sugli attributi e un albero che è formato da nodi che rappresentano l'attributo e da archi che rappresentano il valore dell'attributo. Tra i nodi c'è un AND mentre tra nodi e archi c'è un OR. Ogni percorso corrisponde ad una regola e ogni nodo in un percorso corrisponde ad una preconditione, ogni classificazione della foglia corrisponde ad una post condizione.



## Come si generano i Decision Tree?

Luca Corbucci - Questi appunti non vogliono sostituire il materiale fornito dal docente. Sono solamente un riassunto di slide e libri e potrebbero contenere errori. Ho inserito il mio nome in tutte le pagine con la speranza che gli appunti vengano usati per studiare e non a fini di lucro.



Per generare i decision tree si deve utilizzare un procedimento top down, ovvero si parte dalla radice e poi si generano i nodi successivi. Abbiamo a disposizione gli esempi di training  $X$ , l'attributo target  $T$  e tutti gli attributi  $Attr$ .

Nell'albero il nodo rappresenta il nome dell'attributo, poi sull'arco abbiamo il valore corrispondente.

Per capire come generare i nodi ci si deve chiedere qual è l'attributo da testare e da inserire all'interno dell'albero che ci da maggior guadagno in termine di informazione.

Un possibile algoritmo è ID3, devo partire chiedendomi quale attributo dovrebbe essere testato come radice dell'albero.

ID3 effettua una ricerca greedy per creare l'albero di decisione in cui non viene mai fatto backtracking.

Deve essere scelto il miglior attributo possibile.

Il procedimento è ricorsivo e una volta scelto un nodo vado a prendere i suoi possibili valori e scelgo poi il successivo.

**ID3( $X, T, Attrs$ )**      $X$ : training examples:  
                                   $T$ : target attribute (e.g. *PlayTennis*),  
                                   $Attrs$ : other attributes, initially all attributes

Create Root node

If all  $X$ 's are +, **return** Root with class +

If all  $X$ 's are -, **return** Root with class -

If  $Attrs$  is empty **return** Root with class most common value of  $T$  in  $X$

else

$A \leftarrow$  **best attribute**; decision attribute for Root  $\leftarrow A$

For each possible value  $v_i$  of  $A$ :

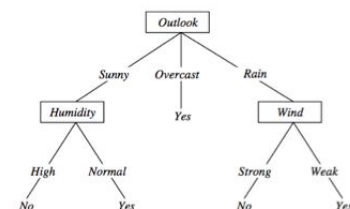
- add a new branch below Root, for test  $A = v_i$

-  $X_i \leftarrow$  subset of  $X$  with  $A = v_i$

- If  $X_i$  is empty **then** add a new leaf with class the most common value of  $T$  in  $X$

**else** add the subtree generated by **ID3**( $X_i, T, Attrs - \{A\}$ )

**return** Root



Se negli esempi di training ho sempre risultato positivo o sempre negativo allora restituisco subito quel risultato

Se invece non ho più attributi da controllare allora restituisco il risultato più comune per i valori che ho selezionato

Per ogni possibile valore di  $A$  devo fare un sottoinsieme delle  $X$  in cui vale che l'attributo selezionato ha il valore selezionato. Una volta che ho creato l'insieme devo andare a vedere se questo sottoinsieme è pieno o no. Se è vuoto prendo + o - a seconda del valore che compare maggiormente, altrimenti devo richiamare la procedura.

Che vuol dire selezionare il miglior attributo, su quali basi scelgo il miglior attributo tra quelli che ho a disposizione?

Per capire quale attributo è migliore viene utilizzata l'entropia, questa entropia misura il grado di impurità in un insieme di esempi ovvero mi dice quanto il set di dati è omogeneo.

L'entropia viene calcolata su un insieme di esempi di training e deve andare a dividere gli esempi positivi dagli esempi negativi in un colpo solo. Più l'entropia è bassa e meglio è.

$$\textbf{Entropy (S)} \equiv -p_+ \log_2 p_+ - p_- \log_2 p_-$$

Il valore dell'entropia varia tra 0 e 1, 1 vuol dire che la situazione è molto omogenea tra positivi e negativi e quindi ci sono molte impurità, 0 vuol dire che la suddivisione tra positivi e negativi è netta, quindi magari ho tutti positivi e 0 negativi o viceversa.

## Information Gain

Per la scelta dell'attributo si utilizza l'information gain che misura quanto bene un certo attributo separa gli esempi di training utilizzando l'entropia.

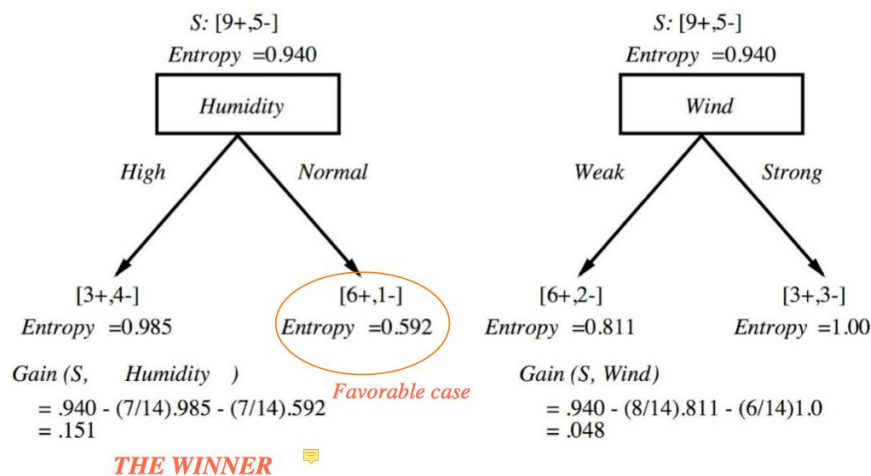
Questo viene definito come la riduzione attesa dell'entropia causata dal partizionamento degli esempi sull'attributo che vogliamo utilizzare.

$$\textbf{Gain(S, A)} = \textbf{Entropy(S)} - \sum_{v \in \text{Values(A)}} \frac{|S_v|}{|S|} \textbf{Entropy(S_v)}$$

In pratica io mi trovo su un certo nodo e ho calcolato il valore dell'entropia per i possibili valori di quell'attributo. Quindi ad esempio se ho due valori possibili ho due figli e per ognuno ho una certa entropia. Ora per ogni possibile attributo che posso scegliere devo andare a calcolare l'entropia che avrei prendendo quello specifico attributo con i suoi possibili valori. Se anche questo ha due figli ho due valori di entropia. Ora prendo il valore dell'entropia che ho calcolato all'inizio e sottraggo i valori dell'entropia che ho calcolato ora.

Se avevo due possibili scelte, otterrò due information gain differenti e quello vincente sarà quello che avrà il valore maggiore perchè vuol dire

che ho avuto una divisione migliore tra casi positivi e negativi e questo comporta che quell'attributo sarà maggiormente effettivo nel momento in cui deve classificare i dati di training.



Problemi con l'information gain:

All'interno del training set potremmo avere dei dati che non sono effettivamente utili per la decisione finale ma che comunque sono presenti e che potrebbero essere scelti come nodi.

Un esempio potrebbe essere la data, questa non influisce sulla decisione, cambia in ogni esempio e quindi avremo sempre una entropia pari a 0, prendendo questo attributo quindi avremo una entropia molto alta ma andremmo a scegliere un attributo non significativo.

La soluzione a questo problema è il **gain ratio** ovvero si deve andare a penalizzare in qualche modo gli attributi che suddividono i valori in tanti sottoinsiemi. Per farlo si va a dividere l'information gain per la splitInformation ovvero:

$$\text{Where} \quad \text{SplitInformation}(S, A) \equiv - \sum_{i=1}^c \frac{|S_i|}{|S|} \log_2 \frac{|S_i|}{|S|}$$

Questa splitinformation può essere bassa se  $S_i$  è quasi uguale a  $S$ , in questo caso possiamo utilizzare due euristiche:

- Calcolare il gain per ogni attributo

- Applicare Gain Ration sugli attributi che hanno il gain sotto la media.

Gli alberi di decisione possono essere utilizzati anche per la ricerca nello spazio delle ipotesi, partendo dalla radice vediamo un albero inizialmente piccolo che poi però cresce a mano a mano che andiamo verso il basso quando troviamo i possibili nodi.

Lo spazio delle ipotesi è completo, non si fa backtracking e non abbiamo garanzia dell'ottimalità.

Decision tree e bias induttivo:

Le uniche "regole" o vincoli che vengono utilizzate nei decision tree con bias induttivo sono le seguenti:

- Gli alberi meno profondi sono da preferire rispetto a quelli più profondi.
- Gli alberi in cui abbiamo gli attributi con l'information gain più in alto vicino alla radice sono da preferire agli altri.

**Bias di ricerca:** ID3 ricerca in uno spazio di ipotesi completo ma con una strategia di ricerca che è incompleta

**Bias di linguaggio:** è dovuto al linguaggio che utilizziamo, tipicamente si utilizza un approccio flessibile nel senso che all'inizio si usano dei modelli molto espressivi e poi ci si sposta verso un modello con un language bias più forte.

Un problema che si verifica utilizzando i decision tree è l'overfitting.

**Overfitting:** si tratta del problema che consiste nel creare un albero che si adatta troppo agli esempi di training.

La nostra ipotesi  $h$  tende a fare overfitting se abbiamo una  $h_1$  che è tale che :

Numero di errori sul training set( $h$ ) < Numero di errori sul training set( $h_1$ )  
e  
errori sui dati successivi( $h_1$ ) < errori sui dati successivi( $h$ )

Più aumenta la complessità dell'albero e più l'accuracy aumenta sui dati di training ma allo stesso tempo diminuisce l'accuracy sui dati ovvero la capacità di generalizzare.

L'overfitting non si può evitare ma si può tenere sotto controllo e ci sono due strategie percorribili:

- Fermare la crescita dell'albero prima di avere una classificazione perfetta
- Far crescere l'albero ma fare successivamente una operazione di pruning

Per fare questo possiamo suddividere il training set in due parti separate, il training set e il validation set, con il training set ci alleniamo il modello e poi con il validation set vediamo di valutare il pruning.

Se scegliamo di fare il pruning consideriamo che ogni nodo può essere eliminato e se viene eliminato perdiamo anche il sottoalbero di quel nodo. L'operazione viene fatta in modo iterativo.

Quello che facciamo è provare l'albero con i dati del validation set, se il risultato è migliore e l'accuracy sul validation set aumenta allora vuol dire che quella parte di albero possiamo eliminarla davvero.

Il pruning si blocca se non ci sono più nodi da tagliare che possano migliorare l'accuracy.

Regole per il pruning:

- Creiamo il decision tree partendo dal training set
- Convertiamo l'albero in una serie di regole del tipo:
- 

*$(Outlook=Sunny) \wedge (Humidity=High) \Rightarrow (PlayTennis=No)$*

- Facciamo il pruning ovvero andiamo ad eliminare le regole che sono presenti nella preconditione e che, con la rimozione, comportano un aumento della accuracy
- Ordiniamo le regole in base all'accuracy e le consideriamo in sequenza quando dobbiamo classificare nuove istanze

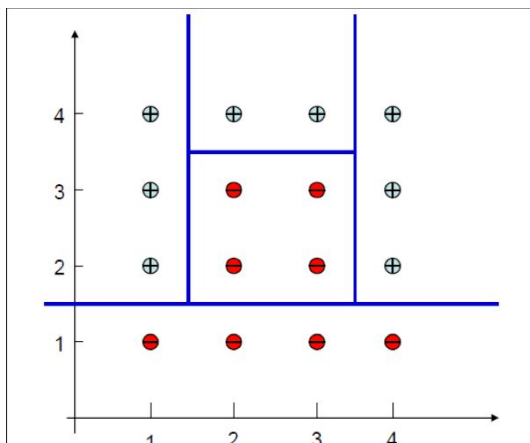
Come ci comportiamo quando abbiamo dati incompleti? Ci sono delle strategie:

- Possiamo utilizzare altri dati di esempio per ottenere dati per l'attributo che manca. Per ogni dato che abbiamo forniamo un valore di probabilità e poi in base alla distribuzione di probabilità andiamo a dare il valore all'attributo che manca.

Possiamo anche avere attributi con costi diversi e magari preferiamo degli alberi in cui si scelgono degli attributi che costano meno.

Quello che si fa è prendere il gain e dividerlo per i costi che abbiamo.

Parlando geometricamente, i Decision tree dividono lo spazio di dati di input in modo differente rispetto al modello lineare che divideva tra 0 e 1. In questo caso infatti abbiamo la possibilità di dividere l'input space in rettangoli, quindi abbiamo una suddivisione più flessibile.



# Dodicesima lezione

## Introduction to Validation and Theoretical Issues

Per usare bene il machine learning è importante comprendere nel modo giusto quando il modello che stiamo utilizzando è valido e quando non è valido.

In particolare per capire se un modello è valido si deve andare a considerare l'errore di generalizzazione, questo indica quanto accuratamente il modello è in grado di predire correttamente sui dati nuovi diversi dai dati di test.

Con generalizzazione si intende che se una  $h$  approssima bene la  $f$  sui dati di training, allora approssima bene anche sui nuovi dati.

Come facciamo a misurare se un modello generalizza bene o no?

- Per la classificazione possiamo utilizzare la MSE per calcolare la loss e possiamo calcolare l'accuracy per capire quante volte il modello fornisce una risposta corretta
- Per la regressione invece possiamo usare MSE e Root MSE e anche misure statistiche.

In generale un alto errore corrisponde ad una accuracy bassa e viceversa.

Quindi è necessaria una fase di validazione che viene suddivisa in due fasi:

- **Model Selection:** se ho più modelli a mia disposizione e devo sceglierne uno, devo cercare di scegliere il migliore per il caso in cui devo utilizzarlo. Il miglior modello viene scelto in base agli iperparametri, ad esempio l'ordine del polinomio è un iperparametro e anche la  $\lambda$  è un iperparametro.

Per il model selection devo andare ad utilizzare i dati di training E NON I DATI DI TEST.

Questa fase mi restituisce un modello

- **Model assessment:** ora che ho scelto il modello da utilizzare devo andare a dichiarare anche una stima dell'accuratezza del modello, quindi questa fase mi va a restituire una stima.  
La stima che devo restituire la vado a calcolare utilizzando i dati di test e NON I DATI DI TRAINING.

Quello che si va a fare è suddividere i dati che abbiamo a disposizione in dati di training e dati di test, i primi li usiamo per allenare il modello e i secondi nel model assessment. Per fare anche model selection devo andare a dividere la parte dei dati di training in una parte di dati di validazione da utilizzare per selezionare il modello migliore.

**Dati di training:** per trovare un modello che fitta i dati ovvero per il training

**Dati Validation:** per il model selection

**Dati di Test:** per il model assessment

E' importante non mischiare i dati, non posso utilizzare i dati di test sia per il model assessment e sia per il model selection perchè andrei a fare una selezione del modello corretta ma una stima del rischio non corretta perchè sarebbe ottimistica rispetto alla realtà.

Algoritmo che unisce model selection e model assessment:

- Per prima cosa dobbiamo separare i dati di training TR dai dati di validazione VL e da quelli per il test TS
- La prima cosa che facciamo è cercare la  $h$  migliore cambiando gli iper parametri del modello (quindi  $\lambda$  o l'ordine del polinomio), in questo caso la  $h$  migliore è quella che causa meno errori con i dati VL. Ogni volta che cambiamo  $\lambda$  allora ricalcolo anche il best.



- Ora cerchiamo la  $h$  migliore che minimizza l'errore utilizzando i dati del training set TR, in questo modo troviamo i parametri  $w$  migliori
- Usando il miglior modello che abbiamo trovato fino ad ora facciamo il fitting sui dati TR e VL
- A questo punto facciamo una valutazione del modello con i dati TS

Se ho pochi dati a disposizione per fare il training, la validation e il test si può usare una tecnica che è detta **K-fold cross validation** in cui si va a fare una suddivisione dell'insieme di dati in vari sottoinsiemi.

Di questi sottoinsiemi ne useremo uno per il training, uno per la validation e uno per i test.

Ogni volta i sottoinsiemi vengono modificati ma non si va contro alla regola che non posso usare gli stessi dati per training e per test perchè sono tutte prove diverse con dati diversi tra loro.

## Simplified Formal Setting

Quello che dobbiamo fare è approssimare una funzione non nota  $f(x)$ , questo può essere fatto andando a minimizzare la funzione del rischio  $r$  che rappresenta l'errore vero su tutti i dati che abbiamo a disposizione, data la loss e una distribuzione di probabilità.

Ricercaire la  $h$  all'interno dello spazio delle ipotesi  $H$  corrisponde a trovare il minimo  $R$  ovvero la minima funzione del rischio  $R$ , questo non possiamo farlo perchè abbiamo solamente un insieme di dati a disposizione, quindi invece di calcolare  $R$  calcoliamo il rischio empirico e troviamo i valori del nostro modello.

Possiamo usare  $R$  empirico per approssimare  $R$  considero che noi vogliamo una funzione che sia in grado di predire non solo sui dati di training ma anche su tutti gli altri?

VC-DIM: misura la complessità di  $H$  ovvero la flessibilità a fittare i dati. Epsilon è la funzione che prende come parametri il numero dei dati, VC ovvero la complessità del modello e il delta che è un fattore scelto per calcolare il rischio.

Quindi il calcolo è inversamente proporzionale alla dimensione dei dati e al fattore di rischio.

$$R \leq R_{emp} + \underbrace{\varepsilon(1/l, VC, 1/\delta)}_{VC\text{-confidence}}$$

Se minimizziamo  $R$  empirico allora vuol dire che minimizziamo anche  $R$  perchè la relazione è di minore o uguale dato che abbiamo un upper bound che mi dice che il rischio non potrà essere maggiore di un certo valore, se si abbassa il rischio empirico allora riusciamo ad abbassare anche il rischio normale.

Controllo della complessità:

La statistical learning theory permette di inquadrare meglio il problema dell'overfitting e della generalizzazione fornendo dei limiti superiori al rischio  $R$ .

Nei modelli lineari la complessità del modello viene controllata grazie al numero dei parametri e grazie al  $\lambda$ , negli alberi di decisione viene controllata tramite il numero dei nodi.

# Tredicesima Lezione

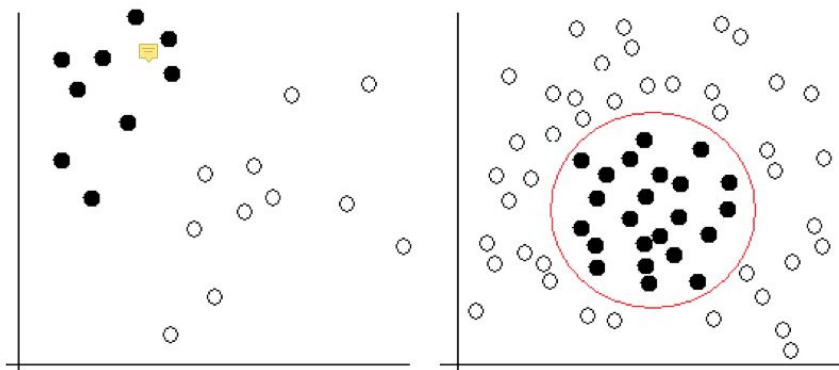
## Introduzione all'uso di SVM

SVM è un altro modello utilizzato nel machine learning, si riparte dai modelli lineari con l'interesse anche verso problemi non lineari.

L'introduzione all'SVM serve per:

- Capire come controllare la complessità dei modelli cercando di fare delle ottimizzazioni utilizzando i classificatori con max margin.
- Cerchiamo di utilizzare una linear basis expansion utilizzando il kernel, un modo diverso dal solito che però mi serve comunque per i problemi in cui ho un learning non lineare
- Evitare usi scorretti dell'SVM

Nel disegno si può vedere un problema linearmente separabile e un problema non linearmente separabile. Nel primo posso dividere perfettamente con una linea le due zone, nel secondo non mi basta una retta quindi serve un polinomio di grado più alto.

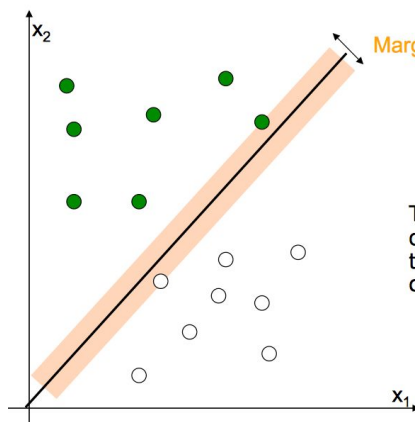


**1° obiettivo controllare la complessità ed utilizzare un margine.**

## Hard Margin SVM

Abbiamo un problema di classificazione, è un problema linearmente separabile e abbiamo il nostro iperpiano separatore che mi divide la zona con i valori +1 da quelli -1.

Per la classificazione vogliamo utilizzare un margine, non tutti gli iperpiani separatori sono uguali, modificandoli infatti varia anche il margine.



Prendiamo la retta separatrice, abbiamo una zona safe in cui posso spostare l'iperpiano senza fare errori, questa zona è il margine.

La definizione di margine: è la doppia distanza tra l'iperpiano separatore e il punto più vicino.

L'obiettivo è trovare il margine più grande ovvero la massima distanza tra l'iperpiano e il punto più vicino.

Per trovare il massimo spostiamo la retta in un certo range e otteniamo vari iperpiani separatori che poi differenziamo in base al margine scegliendo alla fine quello più grande.

Tutti i punti all'interno dell'iperpiano del margine soddisfano l'equazione:

$$wx + b = 0$$

La  $h(x)$  in questo caso sarà data da  $h(x) = \text{segno}(wx + b)$ .

I vettori che mi delimitano il margine sono significativi e sono chiamati vettori di supporto, sono nella forma:

$$|W x_i + b| = 1$$

Ovvero sono quelle  $x_i$  in cui il valore assoluto del piano vale 1, quindi vuol dire che da una parte dell'iperpiano separatore abbiamo valore -1 e da una parte 1.

I punti sono classificati correttamente se  $(w^*x + b) y \geq 1$ .

[se la parte dentro la parentesi è positiva e  $y$  è positivo abbiamo classificato bene, se invece è negativo abbiamo fatto un errore e non abbiamo classificato nel modo giusto].

**Training problem:** in questo caso il nostro obiettivo è trovare il vettore  $w$  e la  $b$  tali che tutti i punti sono classificati in modo corretto e il margine è massimizzato.

Questo training problem ha una forma primale che consiste nel:

$$\text{minimize } |w|^2/2 \quad (\text{i.e. } w^T w)$$

$$\text{such that } (wx_i + b) y_i \geq 1 \text{ for all } i$$

Nel problema primale però non si vedono i vettori di supporto, quindi è necessario passare dal problema primale al problema duale in cui riesco a vedere anche questi vettori. Il problema alla fine si risolve solamente con il duale.

Nel duale possiamo utilizzare un  $\alpha$  che è una costante e calcoliamo la  $h(x)$  in modo differente.

Per prima cosa riscriviamo  $w$  come una sommatoria:

$$w = \sum \alpha_i y_i x_i$$

Poi riscriviamo anche  $h(x)$  come:

$$h(\mathbf{x}) = \text{sign}(\mathbf{w}^T \mathbf{x} + b) = \text{sign}\left(\sum_{i=1}^l \alpha_i y_i \mathbf{x}_i^T \mathbf{x} + b\right) = \text{sign}\left(\sum_{i \in SV} \alpha_i y_i \mathbf{x}_i^T \mathbf{x} + b\right)$$

Dove gli  $\mathbf{x}_i$  sono i dati del training set.

Se le  $\alpha$  sono diverse da 0 allora abbiamo un vettore di supporto.

In questo modo l'iperpiano dipende solamente dai vettori di supporto.

$$h(\mathbf{x}) = \text{sign}\left(\sum_{i \in SV} \alpha_i y_i \mathbf{x}_i^T \mathbf{x} + b\right)$$

**Soft Margin:** l'hard margin può essere troppo restrittivo e quindi non permette alcun errore, in alcuni casi invece può essere utile permettere degli errori ed avere quindi un margine più largo.

Per avere un soft margin viene aggiunta una variabile chiamata "slack variable" che permette di avere alcuni errori di classificazione o del rumore.

Anche se utilizziamo il soft margin possiamo affrontare il problema di training nel primale, in questo caso abbiamo la stessa formula di prima ma viene aggiunta una parte in cui troviamo:

- una  $\psi$  che se è positiva mi indica un errore o un margine troppo piccolo
- $C > 0$  che guida il numero degli errori che possono essere permessi.  $C$  diventa un iper parametro definito dall'utente, se è troppo basso c'è il rischio di avere underfitting, se invece è troppo alto c'è il rischio di avere un margine troppo piccolo quindi c'è possibilità di andare in overfitting. Quindi si deve giocare con il parametro  $C$  e con la  $\psi$ :

### Primal training problem:

minimize  $\frac{1}{2} \|\mathbf{w}\|^2 + C \cdot \sum_i \xi_i$

such that  $(\mathbf{w} \mathbf{x}_i + b) y_i \geq 1 - \xi_i$  and  $\xi_i \geq 0$  for all  $i$

## 2° obiettivo: per i casi non lineari?

Vorremmo ottenere un approccio flessibile anche per i casi di learning non lineare e supervisionato.

Abbiamo dei dati a disposizione che nello spazio a due dimensioni sono mappati in modo da non essere linearmente separabili però se li disegniamo in uno spazio a più dimensioni diventano separabili linearmente.

In pratica in Z3 abbiamo un separatore lineare che corrisponde ad un separatore non lineare nello spazio originale Z2.

Se usassimo la linear basis expansion avremmo un problema legato al possibile overfitting che si verificherebbe, il calcolo diventerebbe anche più complesso però per il machine learning il vero problema sarebbe l'overfitting.

Quindi l'idea è di non usare la formula della classica linear basis expansion ma di usare una nuova forma in cui viene utilizzato il kernel in modo che il margine sia massimizzato e il modello regolarizzato.

La classica  $h(\mathbf{x})$  dell'SVM è questa:

$$h(\mathbf{x}) = \text{sign}\left(\sum_{i \in SV} \alpha_i y_i \mathbf{x}_i^T \mathbf{x} + b\right)$$

Quello che facciamo è avere una funzione K (kernel) che mi funziona in modo simile alla phi che avevo nella linear basis expansion.

Quindi la  $h(\mathbf{x})$  per i modelli non lineari diventa:

$$h(\mathbf{x}) = \text{sign}\left(\sum_{i \in SV} \alpha_i y_i K(\mathbf{x}_i, \mathbf{x})\right)$$

La funzione K può essere usata in vari modi differenti e quindi si deve scegliere quella migliore per il nostro caso.

Avere il kernel K non è la stessa cosa di avere la phi.

**Linear:**  $K(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^T \mathbf{x}_j$

- Mapping  $\Phi: \mathbf{x} \rightarrow \phi(\mathbf{x})$ , where  $\phi(\mathbf{x})$  is  $\mathbf{x}$  itself

**Polynomial** of power  $p$ :  $K(\mathbf{x}_i, \mathbf{x}_j) = (1 + \mathbf{x}_i^T \mathbf{x}_j)^p$

- Mapping  $\Phi: \mathbf{x} \rightarrow \phi(\mathbf{x})$ , where  $\phi(\mathbf{x})$  has exponential dimension in  $p$

**RBF (radial-basis function) Gaussian:**  $K(\mathbf{x}_i, \mathbf{x}_j) = e^{-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{2\sigma^2}}$

- Mapping  $\Phi: \mathbf{x} \rightarrow \phi(\mathbf{x})$ , where  $\phi(\mathbf{x})$  is *infinite-dimensional*

Riassunto:

- Dobbiamo scegliere un parametro C, la funzione del kernel K con i suoi parametri
- Risolviamo il problema di ottimizzazione per trovare alpha, quindi abbiamo risolto il duale del problema di training dell'SVM
- Dopo questi calcoli otteniamo il modello definitivo dell'SVM:

$$h(\mathbf{x}) = \text{sign}\left(\sum_{i \in SV} \alpha_i y_i K(\mathbf{x}_i, \mathbf{x})\right)$$

### 3° obiettivo: evitare un uso sbagliato dell'SVM



L'overfitting nell'SVM possiamo averlo se non scegliamo nel modo giusto il  $C$ , il kernel e i parametri del kernel.

La validazione invece può essere effettuata con il model selection e il model evaluation come visto già nelle slide precedenti.

L'SVM è un modello molto utile e popolare.

Sfrutta il max margin e la linear basis expansion con il kernel per ottenere un modello che sia allo stesso tempo flessibile e che controlli anche la complessità.

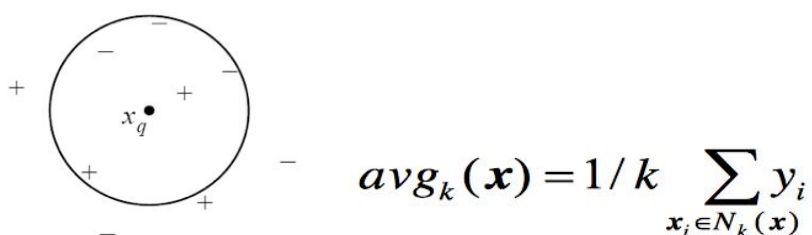
# Quattordicesima lezione

## KNN: K nearest neighbors

KNN è un metodo di supervised learning in cui abbiamo a disposizione dei dati di training  $\langle x, y \rangle$ . Quello che si fa è andare a ricordare questi dati di training, nel momento in cui ho un nuovo dato da classificare vado a fare una ricerca all'interno dello spazio dei dati che ho a disposizione. Posso trovare un dato simile a quello da classificare o un dato identico. Se trovo un dato identico restituisco la  $y$  corrispondente, se invece non ne ho uno identico utilizzo la formula della distanza euclidea per trovare un dato che sia abbastanza simile a quello su cui voglio fare la previsione, quando trovo il più simile allora mando in output quella  $y_i$ . Per fare la previsione, oltre alla distanza posso utilizzare anche una metrica che mi dice a chi sono simile.

Un modello di questo tipo è molto flessibile, forse anche troppo, infatti i confini del “decision boundary” risultano abbastanza irregolari cosa che comporta un possibile overfitting perchè sui dati di training l'errore è 0 ma sui dati di test poi posso avere degli errori.

Un'alternativa può essere quella di non guardare solamente un vicino per capire il valore in uscita per quell'input, guardiamo infatti vari vicini e cerco di mandare in output il valore che compare maggiormente facendo la media. Se sto facendo la regressione allora posso restituire direttamente questo valore, altrimenti se faccio la classificazione considero sempre la media però restituisco 1 o 0 a seconda del valore restituito, se è maggiore di 0,5 restituisco 1 altrimenti 0.


$$avg_k(x) = 1/k \sum_{x_i \in N_k(x)} y_i$$

Quindi aumentare il numero dei vicini di cui mi fido mi permette di avere un modello che è sempre flessibile ma che allo stesso tempo permette di evitare di avere l'overfitting. Anche in questo caso avrei un problema non lineare e quindi dei confini non troppo regolari.

Numero dei vicini di cui mi fido basso → modello troppo flessibile → overfitting.

Numero dei vicini di cui mi fido alto → modello poco flessibile → underfitting.

Nel modello KNN abbiamo quindi l'obbligo di portarci dietro i dati, non posso buttarli via.

Il fatto di portarsi dietro i dati e confrontarli ogni volta con i dati su cui voglio fare la predizione comporta un aumento di costo nella fase di predizione perchè devo andare a calcolare ogni volta delle distanze.

Allo stesso tempo la fase di training avrà un costo uguale a 0.

Un altro problema è legato alla curse of dimension, in pratica se abbiamo un insieme di dati che devono essere rappresentati in più dimensioni, il volume cresce e quindi cresce anche la distanza tra i dati. Quando andiamo ad inserire un nuovo dato su cui vogliamo fare una previsione abbiamo il problema che le distanze degli altri dati da questo saranno così alte che non avrà un senso andare a calcolare la distanza per fare la media. I dati sarebbero troppo sparsi per poter trovare delle similitudini.

L'inductive bias del modello riguarda il tipo di metrica che scelgo per decidere se due dati sono simili tra loro.

Una metrica può essere la distanza tra le feature dei dati, di solito dipende anche dal dominio perchè a seconda del problema ho una metrica differente per capire se due cose sono simili o no.

## K-Means

In questo caso va gestito un task che non è supervisionato, vuol dire che abbiamo un training set in cui abbiamo solamente la  $\langle X \rangle$  e non il valore di output associato a quel dato.

Quello che possiamo fare è andare a raggruppare questi dati in modo che si organizzino in cluster ovvero in raggruppamenti di dati simili tra loro (basandomi sulla simmetricità dei dati) con un elemento centrale che possa “rappresentare” tutto il cluster, questo elemento è detto centroide.

Quello che si fa è una analisi esplorativa di questi dati per trovare i pattern comuni, questo lo facciamo perchè è meno costoso andare a trovare dei pattern comuni e quindi organizzare i dati in cluster piuttosto che etichettare i dati a mano, specialmente se i dati da etichettare sono tanti.

La loss in questo caso si misura considerando il cluster in cui viene inserito il dato e la  $x_i$  da classificare, questo viene chiamato errore di distorsione o ricostruzione:

$$L(h(\mathbf{x}_i)) = \|\mathbf{x}_i - c(\mathbf{x}_i)\|^2$$

### Algoritmo K-Means

L'obiettivo di questo algoritmo è calcolare l'errore di distorsione e poi utilizzarlo per trovare il centroide del cluster. E' un algoritmo semplice da usare ma ha alcune limitazioni.

Algoritmo:

- Abbiamo  $k$  cluster differenti, scegliamo  $k$  centri di cluster che coincidono con  $k$  punti scelti random all'interno della zona dove stanno i cluster
- Ad ogni centro vado ad assegnare un certo pattern

- Ricalcolo il centro del cluster utilizzando gli attuali punti che sono all'interno del cluster.

Una volta spostato il centroide verso l'effettivo centro del cluster devo ricalcolare le distanze tra i punti e il centroide in modo che tutti i punti finiscano nel cluster corretto.

Limitazioni di K-Means:

- Il numero dei cluster deve essere scelto a priori
- Possiamo avere un problema con i minimi locali e quindi può capitare di dover svolgere l'algoritmo più volte
- Funziona bene se i cluster hanno una forma compatta e sferica ma non troppo bene se ho forme strane
- Non abbiamo possibilità di visualizzare i dati in spazi di dimensione inferiore.

Come faccio a valutare se K-Means si è comportato nel modo giusto?

Devo andare a confrontare l'output di due esecuzioni differenti tra loro, la K deve però essere la stessa in entrambi i casi perchè se la k è minore in un caso allora vuol dire che l'errore è diminuito.

Possiamo ad esempio capire quale divisione in cluster ha funzionato meglio se andiamo a considerare l'errore di quantizzazione.

Un'altra possibilità consiste nel prendere i dati nel formato  $\langle x, y \rangle$ , quindi etichettati, non consideriamo però l'etichettatura e facciamo k-Means andando a classificare i dati in cluster differenti, poi confrontiamo la suddivisione in cluster con l'effettiva etichettatura dei dati in modo da capire se la divisione in cluster è stata effettuata nel modo giusto oppure no.

Nel learning non supervisionato può essere utile svolgere la dimensionality reduction ovvero una riduzione dei parametri in input, in questo modo tramite la PCA andiamo a cercare gli assi di maggiore varianza all'interno del grafico originale e poi li portiamo in un grafico che è più semplice da visualizzare rispetto a quello originale.

