

# UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELLE  
TECNOLOGIE DELL'INFORMAZIONE

CORSO DI INTELLIGENZA ARTIFICIALE

Elaborato Esercitativo

## Work Projects

*Milo Saverio*  
*Previtera Gabriele*  
*Pommella Michele*

Prof. Neri Filippo

Anno 2016-2017

# Work Project 1

Questo primo lavoro di gruppo si incentra sulla comprensione empirica dei **Decision Trees**, esempio di strumento ampiamente diffuso nel **Machine Learning**, e sul **Problem Solving**, metodo scientifico applicato dall'I.A. per la risoluzione dei problemi.

## 1.1 Esercizio 1: Decision Trees

### 1.1.1 Esperimento: creazione Decision Tree

Un albero di decisione è un modello di rappresentazione delle decisioni e delle loro possibili conseguenze, costruito al fine di supportare l'azione decisionale. Costituisce un importante strumento nel contesto dell'**Inductive Learning**, in cui ricopre il ruolo di modello predittivo su cui si basa il comportamento dell'agente. La sua struttura discende da un insieme di esempi dati e determina le regole di condizione-azione atte alla classificazione di esempi futuri. Dunque l'agente è in grado di apprendere da una serie di dati il comportamento da assumere in situazioni non specificate.

#### Iris

Come primo esperimento abbiamo adoperato un insieme di dati facenti riferimento a varie specie di *Iris*, le quali vengono caratterizzate da quattro attributi: lunghezza del sepalo, larghezza del sepalo, lunghezza del petalo, larghezza del petalo. Queste grandezze sono dimensionalmente espresse tutte in *cm*. Il *target* è, appunto, la tipologia di *Iris*.

La dimensione complessiva del dataset è di 150 elementi. Per effettuare la prova è stato necessario formattare l'insieme dei dati originario, rendendolo consistente con le esigenze algoritmiche correlate al linguaggio utilizzato, *Python* nel nostro caso.

Gli esempi del dataset si presentano nella seguente forma:

```
5.1,3.5,1.4,0.2,Iris-setosa
```

Abbiamo, dunque, determinato i valori degli attributi per ciascun esempio in base alle virgole delimitatorie, riconosciuto i valori numerici precedentemente visti come stringhe (si è realizzato ciò mediante una funzione adibita allo scopo, sfruttando il casting) ed eliminato eventualmente il carattere di *new line* al fine di evitare valori spuri.

Di seguito la funzione implementata per caricare gli esempi dal file nella lista utilizzata nel programma:

---

```
1 def aprifile(fil="nomefile.txt"):
2     data=[]
3     for line in file(fil):
4         srt=line.split(',')
5         for count in range(0,len(srt)):
6             if(isfloat(srt[count])):
7                 srt[count]=float(srt[count])
8             else :
9                 srt[count]=srt[count].strip('\n')
10        data=data+[srt];
11    return data
```

---

Un campione di esempio formattato si presenterà, quindi, nella forma:

```
[5.1 , 3.5 , 1.4 , 0.2 , 'Iris-setosa ']
```

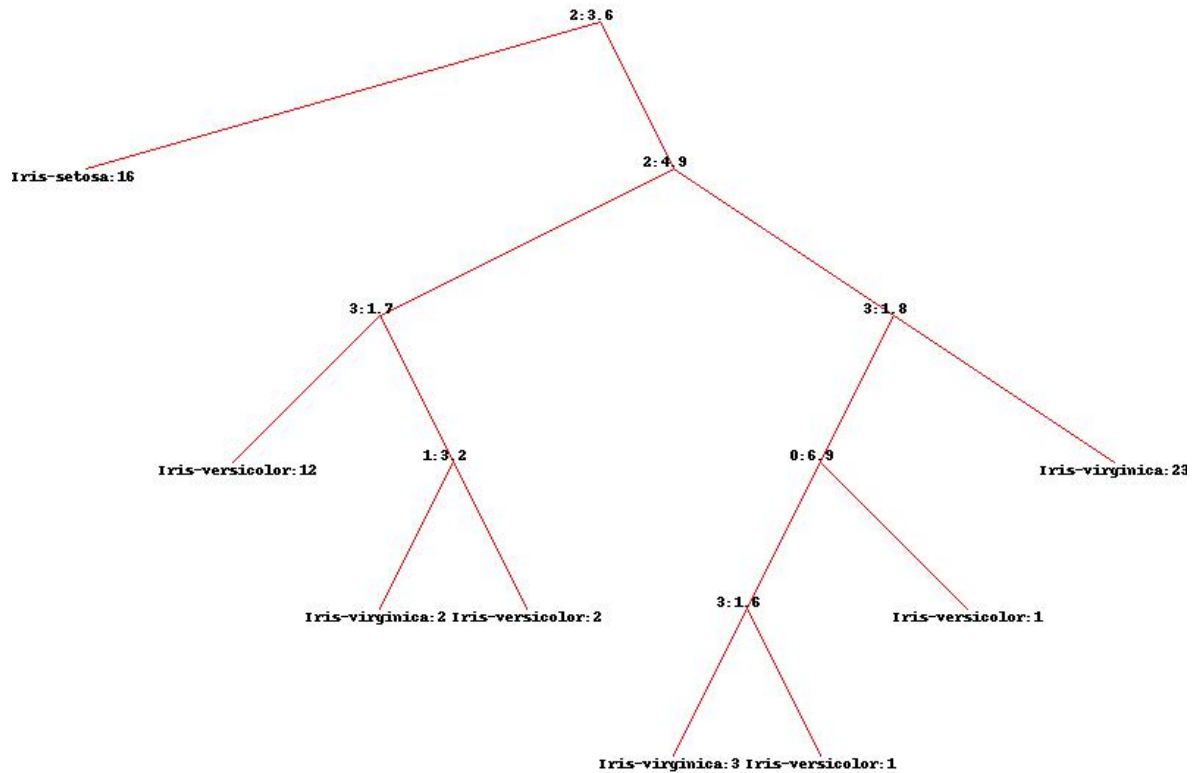
Decidiamo di usare il 40% dei dati forniti dal dataset come training set, quindi per effettuare l'apprendimento, ed il restante 60 % per il test set. Per la determinazione di training set e test set si è utilizzata una funzione che seleziona un insieme casuale di esempi dal dataset di partenza, con cardinalità determinata dai parametri di ingresso in base alla percentuale desiderata (calcolo effettuato prima della chiamata).

---

```
1 def createdataset(data,numdati):
2     tr=[]
3     te=[]
4     t=[]
5     for i in range(0,numdati):
6         t=random.choice(data);
7         tr=tr+[t]
8         num=data.index(t)
9         del data[num]
10    te=data
11    return(tr,te)
```

---

Costituito il training set, vengono sfruttati gli algoritmi forniti per la costruzione dell'albero (`buildtree`) e la relativa rappresentazione (`drawtree`).



Otteniamo, infine, l'agognato albero di decisione!

Come si evince dalla figura, il nodo scelto come radice è relativo alla terza colonna (colonna numero 2 da programma, poiché la numerazione rimarca quella delle liste in python) ed il valore che ne minimizza la funzione d'entropia è 3.6. Notiamo che questa scelta ci permette già la classificazione di 16 esempi. L'albero, attraverso le informazioni del training set, classifica tutti gli esempi, anche se, evidentemente, il *guadagno informativo* degli attributi non sarà alto. Osserviamo, infatti, che un attributo non riesce a caratterizzare nettamente un gruppo di esempi. Conseguentemente, ciascuno di essi viene richiamato più volte per lo *split* dei dati, aumentando inevitabilmente la profondità dell'albero.

Alla luce di queste osservazioni possiamo giungere alla seguente considerazione: molto probabilmente questa classificazione non sarà abbastanza generalizzante ed, ipotizzando di utilizzare un test set

sufficientemente vario, non ci offrirà ottimi risultati in termini di *performance*. Si potrebbe, però, ricercare una maggiore generalizzazione tramite l'applicazione di tecniche di *pruning*, che diminuirebbero la profondità dell'albero.

## Mushrooms

Analizziamo ora il data set *Mushroom*, relativo ai funghi. Il fine sarà di determinarne l' habitat. Il dataset è caratterizzato da ben 21 attributi e 8124 istanze. Utilizziamo sempre le medesime percentuali di dati per il training set ed il test set (40%-60%).

L' albero, come si può osservare dalla figura 1.1 nella pagina seguente, non è in grado di classificare gli esempi distintamente. In questo stato l'albero può portare a decisioni inconcludenti. Risultato analogo si è ottenuto al seguito di differenti esperimenti sul medesimo data set, rivelandosi tutt'altro che un *unicum*. Ciò ci ha portato ad interpretare questo esito come conseguenza del fatto che gli attributi utilizzati non siano abbastanza rappresentativi, non riuscendo a classificare pienamente i vari esempi.

Notiamo, in conclusione, che un grande numero di attributi non necessariamente porta a spiegare tutti gli esempi.

### 1.1.2 Esperimento: Learning Curve

Per valutare l'approssimazione dell'ipotesi alla funzione ideale (**Problem of Induction**) si utilizzano delle misure di *performance*, senza le quali il solo albero delle decisioni risulta un modello incompleto. La misura di performance da noi adoperata fa uso di un test set per stimare l'accuratezza del modello di apprendimento. Essa può essere descritta mediante la **Learning Curve**, che rappresenta la percentuale di correttezza sul test set rispetto alle dimensioni del training set.

In questo esperimento ci porremo, quindi, come obiettivo la ricerca di una curva di apprendimento. Si è scelto, al fine di preservare un test set consistente, di attribuire al training set dal 10% fino al 50% dell'intero data set. Il data set impiegato è inerente ad un censimento atto ad identificare gli individui con reddito superiore ai cinquanta mila dollari annui.

La curva è stata prodotta facendo uso della funzione `fperformance`, che si avvale, a sua volta, di una funzione `performance` per il calcolo dell'*accuracy*, e della libreria `matplotlib` per il tracciamento del grafico in questione. Al seguito di una fase iniziale, prevedente l'inizializzazione delle variabili (test set massimo) e la determinazione del numero di esempi corrispondenti al 10% del dataset, si procede con un ciclo di 5 iterazioni che implementa l'intera



**Figure 1.1:** Mushrooms Decision Tree

logica. Ad ogni ciclo, `createdataset` drena il 10% del test set nel training set (memorizzato iterativamente nella lista `t`). Segue la creazione dell'albero, il calcolo della sua accuratezza, di cui la lista `p` memorizza i differenti valori da porre sull'asse delle ordinate, e l'aggiornamento della lista `perc`, che memorizza i valori dell'asse delle ascisse. Al termine del ciclo ci si avvale delle funzioni offerte dalla libreria per la rappresentazione grafica della curva.

---

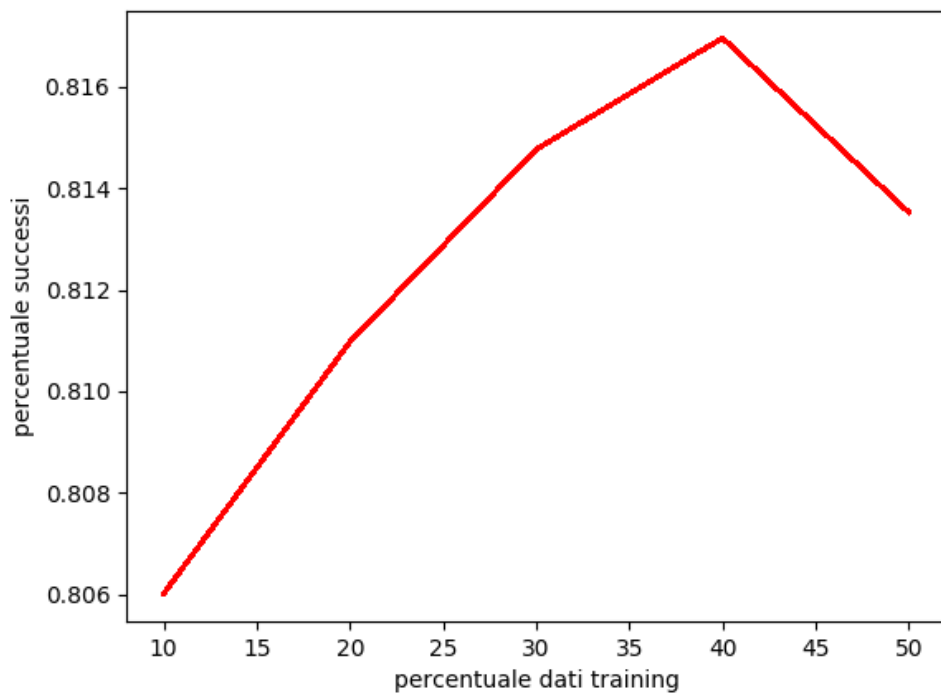
```
1 def fperformance(data):
2     testc=data
3     percent=10
4     p=[]
5     perc=[]
6     t=[]
7     numdati=(int)((float)(len(testc))/100*percent);
8     for i in range(0,5):
9         (train,testc)=createdataset(testc,numdati)
10        t=t+train;
11        tree=buildtree(t)
12        p=p+[performance(tree,testc)]
13        perc=perc+[percent]
14        percent=percent+10;
15    line,=plt.plot(perc,p,'r-')
16    plt.xlabel('percentuale dati training')
17    plt.ylabel('percentuale successi')
18    line.set_antialiased(False)
19    plt.show()
20
21
22 def performance(tree,test):
23     t=0
24     for row in test:
25         results=classify(row,tree)
26         for r in results:
27             if r==row[len(row)-1]:
28                 t=t+1
29     percent=float(t)/len(test)
30     return percent
```

---

Abbiamo fatto uso della funzione `performance` per il calcolo dell'accuratezza. Essa effettua, tramite l'albero, la classificazione per ogni esempio del test set

(esito in una variabile dizionario **results**) ed effettua un conteggio nel caso in cui il risultato ottenuto sia coerente con il relativo *target*. Si ottiene, infine, la percentuale di correttezza sul test set tramite il rapporto tra gli esempi correttamente classificati e quelli totali.

Visualizziamo graficamente il prodotto di questo algoritmo sul dataset di riferimento.



Come si evince dal grafico, la nostra accuratezza migliora fino al 40%, dopodiché si percepisce un degrado dell'apprendimento all'aumentare dei dati forniti al training set. Abbiamo interpretato questo andamento supponendo che, con l'incremento di quest'ultimo, si venga a costituire un albero più specifico (*overfitting*), ovvero con meno capacità di classificare gli esempi. Osservando la curva di apprendimento, si è ipotizzato che questa rappresentazione dell'ambiente descritto dal dataset presenti un'espressività ridondante. Ciò è stato dedotto dall'analisi della crescita, lenta nell'intervallo di riferimento: al crescere del training set dal 10% al 50% si rileva un aumento massimo di circa 1%. Per migliorare la rapidità della curva si potrebbe considerare un insieme ridotto di attributi, ricercando una rappresentazione, mediante albero di decisione, più compatta. Nella determinazione degli attributi si dovrebbe tentare di evitare la ridondanza, eventualmente focalizzandosi su quelli più rappresentativi e con un alto



guadagno informativo.

### 1.1.3 Domanda 1

Un agente in grado di apprendere mediante alberi di decisione fonda questo suo processo su principi di apprendimento induttivo (**inductive learning**).

L'apprendimento induttivo è una forma di apprendimento basata sull'induzione a partire da esempi dati. Esso, data una collezione di esempi (**training set**) della funzione **target**  $f$  che si vorrebbe imparare, mira a restituire una funzione  $h$  (**hypothesis**) che approssimi al meglio la  $f$ .

Concettualmente, il criterio nella determinazione di  $h$  tra le differenti funzioni dello spazio delle ipotesi dovrebbe essere legato, più che alla consistenza nello spiegare i dati, alla bontà dell'approssimazione e quindi alla capacità di generalizzazione per predire esempi non ancora incontrati. In questo senso, l'agente agisce in modo razionale poichè cerca di **decidere come comportarsi in situazioni a lui sconosciute basandosi su quelle già note**. Possiamo individuare proprio in questo aspetto una **forma di intelligenza**, determinata dall'agire razionalmente.

### 1.1.4 Domanda 2

Le procedure di **Decision Tree Learning** consentono la costruzione di un albero di decisione "piccolo", consistente con gli esempi forniti in input per la determinazione di tale struttura. Ogni nodo interno all'albero corrisponde ad una condizione sul valore di un attributo, gli archi verso i nodi figli ai possibili valori per quell'attributo, le foglie alla classificazione. Si ottiene così, attraverso i **path** dell'albero, una rappresentazione compatta delle regole di condizione-azione. L'albero di decisione prende in input una situazione descritta da un insieme di attributi e restituisce una decisione, ovvero il valore predetto di uscita per tale input, sulla base del cammino percorso. In questo senso possiamo parlare di apprendimento, poichè, **alla luce di un dato insieme di esempi, si viene a costituire un albero di decisione dalla ben determinata topologia e legge condizione-azione, utilizzabile per la classificazione di esempi futuri**.

## 1.2 Esercizio 2: Problem Solving

Il problema descritto è del tipo *non deterministico e parzialmente osservabile*, quindi classificabile come **Contingency Problem**. Essendo il pianeta

caratterizzato da una proprietà di elasticità geografica, possiamo immaginare che forze esterne, ad ogni cambio di città del robot, facciano variare le distanze tra le differenti città, incrementandole o diminuendole. Per tale motivo l'algoritmo di navigazione proposto fa uso di un albero di ricerca con strategia di analisi in profondità (**depth-first search**), la quale prevede di espandere primariamente il nodo più profondo non espanso.

L'alterazione non deterministica delle distanze ci ha spinto a sottolineare l'importanza di **minimizzare il numero di città percorse** lungo il tragitto verso l'obiettivo. Un maggior numero di città attraversate aumenterebbe probabilisticamente la distanza percorsa. Sarebbe, quindi, auspicabile evitare, o minimizzare, il numero di processi di risalita dell'albero, che prevederebbero il ritorno alle stesse città più volte, e continuare nella navigazione in profondità, sfruttando, nel migliore dei casi, la possibilità di arrivare al **goal** senza attuare un **backtracking**.

Un'ulteriore scelta, dettata dalla natura del problema, riguarda l'utilizzo della struttura dati lista semplice. Ciò dipende dal fatto che, in questo esempio, le distanze tra le città non sono costanti ed anzi considerate varianti in maniera non predicibile e non deterministica. Quindi, la struttura dati lista, ordinata in base alla distanza, non si adatta a tale tipologia di problema (a differenza di quello trattato in classe).

Inoltre, per evitare l'insorgere di cicli, si è previsto di tenere traccia dell'insieme dei nodi già esplorati mediante l'utilizzo di una lista **closed**, ottenendo in definitiva un algoritmo del tipo **Graph Search**.