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
Pommella Michele
Previtera Gabriele

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, sfruttante 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
```

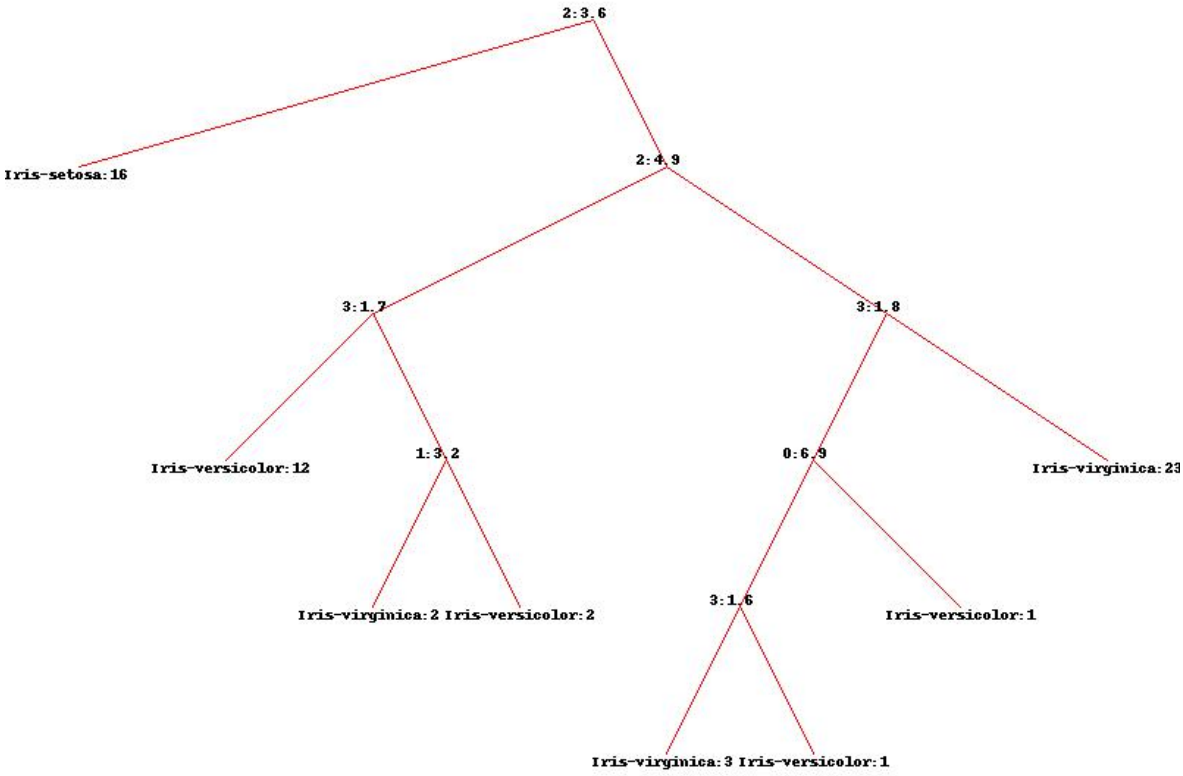
Un campione di esempio formattato si presenterà, quindi, nella forma:

```
[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 dataset *Mushroom*, relativo ai funghi. Il fine sarà quello di determinarne se un fungo sia commestibile o avvelenato. 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, classifica tutti gli esempi distintamente senza utilizzare tutti gli attributi forniti dal dataset. Esso si presenta, quindi, in una forma compatta e generalizzante.
Ipotizziamo, allora, che l'albero sia stato costruito facendo uso di attributi rappresentativi con alto guadagno informativo, in grado di effettuare un chiaro split degli esempi sulla base dei valori da essi assunti.

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 corrispondete al 10% del dataset, si procede con un ciclo di 5 iterazioni che implementa l'intera logica. Ad ogni ciclo, `createdataset` drena il 10% del test set nel training set (memorizzato iterativamente nella lista `t`). Segue la creazione dell'albero,

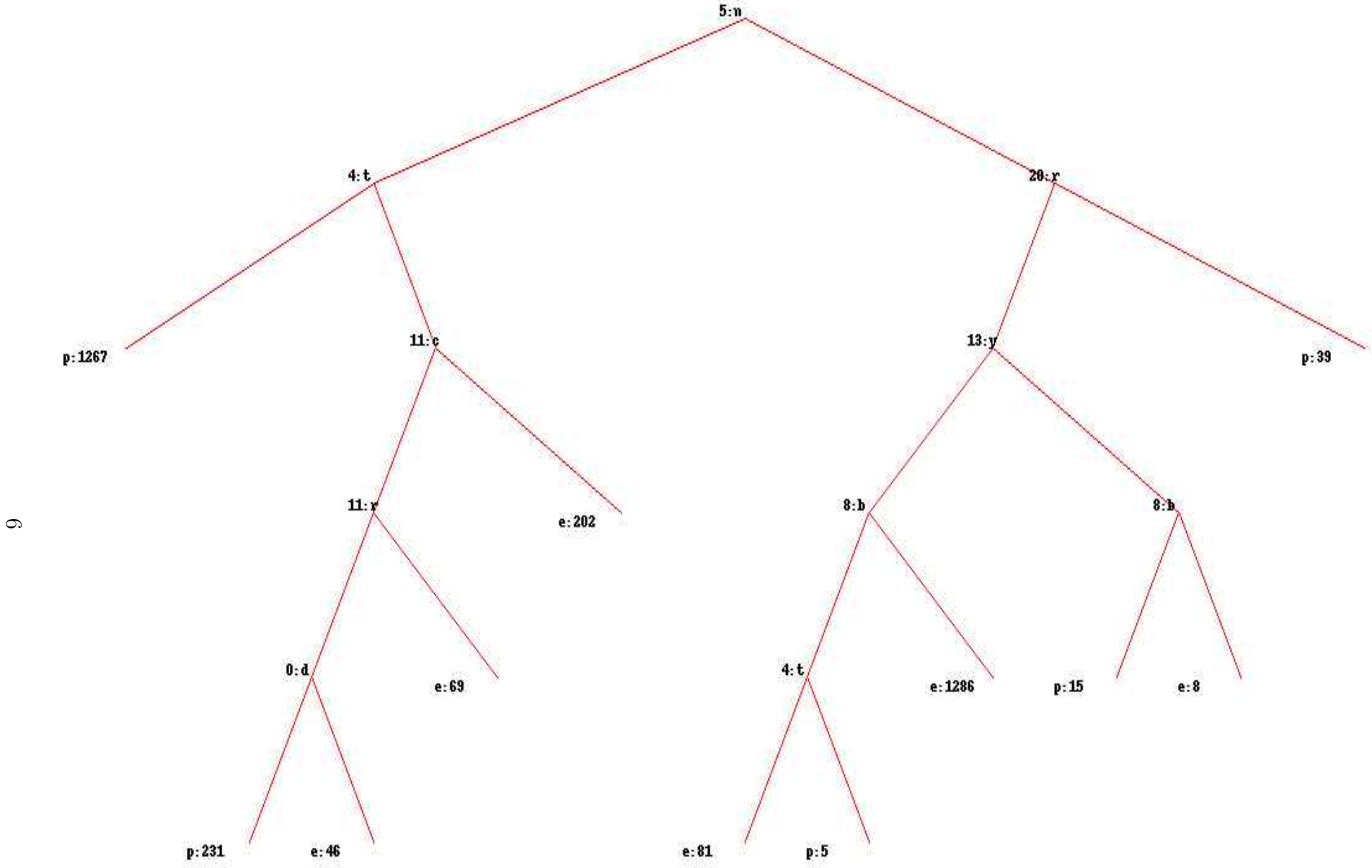


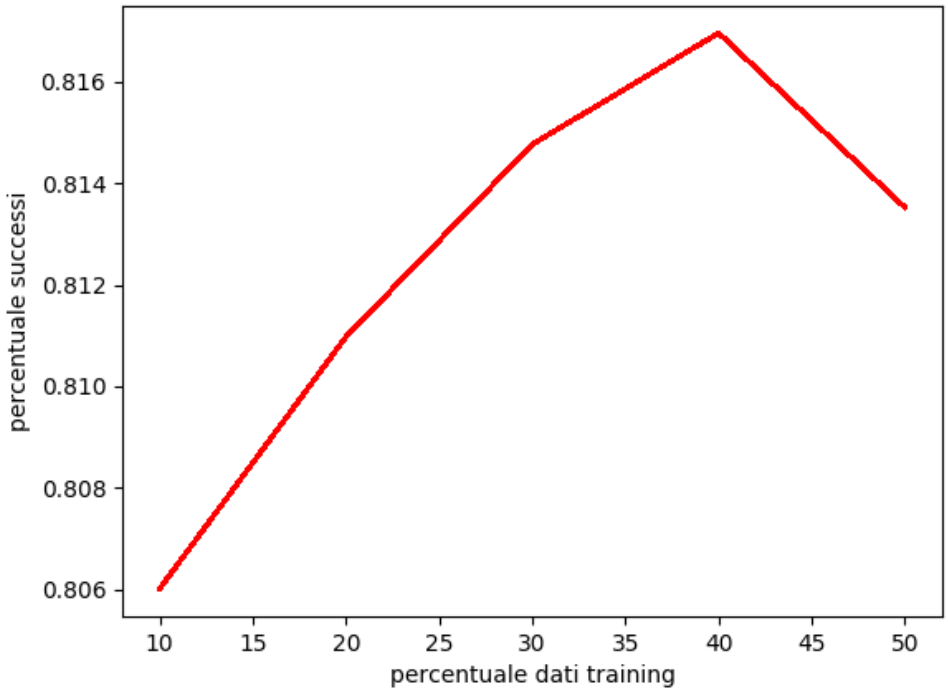
Figura 1.1: Mushrooms Decision Tree

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, poiché, **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.

Work Project 2

2.1 Esercizio 1: Evolutionary Optimization

2.1.1 Esperimento 1: Ottimizzazione

Attraverso gli algoritmi forniti siamo stati in grado di sperimentare le differenti tecniche di *ricerca locale* su diversi problemi. Esse si basano sul principio dei miglioramenti successivi: si cerca in primo luogo una soluzione, anche non ottima, ed in seguito ci si concentra sull'ottimizzazione. Va considerato che spesso si riesce a determinare solo un massimo locale, ottenendo risultati più o meno soddisfacenti a seconda dell'applicazione.

Random Searching

Questo algoritmo prevede semplicemente la determinazione casuale di un insieme di soluzioni e la comparazione dei relativi costi per l'identificazione di quella a costo minimo.
Abbiamo effettuato differenti esperimenti variando il numero di soluzioni estratte casualmente: ad un suo eccessivo aumento non si ottengono necessariamente risultati nettamente migliori da giustificare l'evidente incremento della complessità computazionale in termini temporali; d'altro canto, un numero esiguo di soluzioni produce risultati dalla qualità incostante.

Hill Climbing

L'`Hill Climbing` inizialmente determina una soluzione ed i suoi "vicini", se uno di essi presenta un costo migliore della soluzione corrente, verrà scelto questo come soluzione corrente alla prossima iterazione, attuando in tal modo il processo dei miglioramenti successivi; se la soluzione corrente ha costo minore dei suoi vicini, termina la serie di miglioramenti iterativi poiché si è in presenza di un massimo locale (minimo locale rispetto al costo a seconda della formulazione del problema).

Vari esperimenti ci hanno condotto a risultati decisamente migliori del `Random Searching`, che sottolineano la maggior efficienza dell'`Hill Climbing`. La variabilità nella qualità della soluzione, al seguito di svariate esecuzioni, è dipendente dai massimi locali del problema specifico: partendo da situazioni iniziali diverse, arriviamo a risultati diversi.

Simulation annealing

Versione rivisitata dell' L'`Hill Climbing`, questo metodo di ottimizzazione è ispirato dal riscaldamento termico di una lega e al suo successivo raffreddamento. Viene comparata la soluzione corrente ad una determinata casualmente (nel nostro caso tramite un cambiamento di quella corrente): l'algoritmo tende a spostarsi sempre verso una soluzione successiva migliore, solo con una certa probabilità verso quelle peggiori, che tenderà a diminuire col tempo. Questa probabilità dipende dalla qualità della soluzione successiva: mosse pessime avranno minore probabilità. Viene, inoltre, introdotto un concetto di "temperatura": essa determina la probabilità di spostarsi verso soluzioni non ottime. Inizialmente questo valore sarà alto (temperatura elevata), in seguito essa tenderà a calare (raffreddamento), consentendo, in ultima battuta, la scelta della soluzione ottimizzata. Tale probabilità è data dalla seguente espressione:

$$e^{\frac{-(costosuccessivo+costoattuale)}{temperatura}}$$

Osserviamo che con l'abbassamento della temperatura, ad ogni iterazione dell' algoritmo, tale probabilità decresce. Le ragioni alla base di questa logica risiedono nella maggiore possibilità di trovare un massimo globale, evitando che ci si possa assestare su un punto di massimo locale come una *spalla* (shoulder).
Dalle prove sperimentali abbiamo constatato risultati migliori o uguali all'`Hill Climbing`, appurando la maggiore capacità di ottenere soluzioni con costo inferiore.

Genetic Algorithms

Ulteriore algoritmo che tenta di superare i limiti dell'`Hill Climbing` è quello genetico. Esso è ispirato dalla natura, proponendo un'evoluzione delle soluzioni tipicamente darwiniana. L'algoritmo prende le mosse selezionando un insieme casuale di soluzioni, detto *popolazione*. Gli esemplari di questa vengono poi valutati ed ordinati attraverso una *funzione di fitness* (rappresentata nel nostro caso dal costo minimo). Nell'implementazione utilizzata, si procede alla creazione della successiva *generazione* a partire da

un prestabilito numero di migliori soluzioni. Attraverso il processo di *elitarismo* quelle ottime sopravviveranno nella nuova generazione; infatti questa sarà costituita attraverso la *mutazione* (cambiamento casuale ad una soluzione) ed il *crossover* (combinazione di due soluzioni) dell'elite.
Vari esperimenti di hanno portato a riscontrare risultati simili al `Simulated Annealing`, se non migliori in taluni casi, che presentano, però, una minore variabilità con misure ripetute. Riusciamo, in definitiva, ad ottenere una buona ottimizzazione, di qualità superiore all'`Hill Climbing` ed al `Random Searching`.

2.1.2 Esperimento 2: Caso d'uso

L'ottimizzazione sulla quale ci vogliamo concentrare è quella di una funzione $F(x)$, così definita:

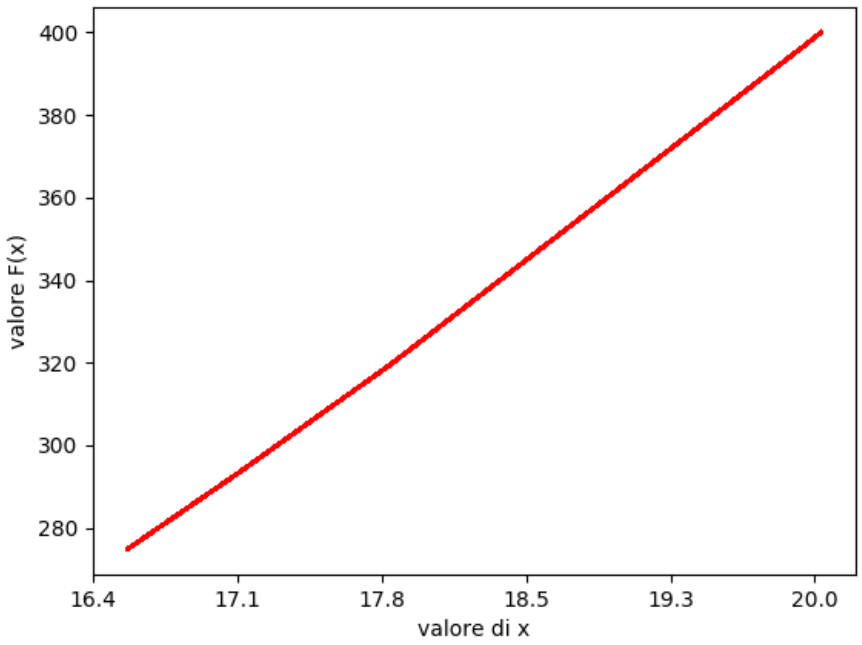
$$F(x) = \begin{cases} 10 & \text{se } x < 5.2 \\ x^2 & \text{se } 5.2 \leq x \leq 20 \\ x - 1 & \text{se } x > 20 \end{cases} ; x \in [-100, 100]$$

dove la x può assumere valori tra $[-100, 100]$. La ricerca di tale valore verrà effettuata tramite algoritmo genetico. Prima di tutto, però, dobbiamo definire la funzione di costo, così da poter determinare quando la nostra soluzione tende a migliorare o, viceversa, a peggiorare:

```
1 def costmax(sol):
2         if sol[0]>100 or sol[0]<-100:
3                 cost=0
4         else:
5                 cost=F(sol)
6 return cost
```

La nostra funzione di costo è definita nel seguente modo: se il valore di cui si calcola il costo non è accettabile, poiché nel passaggio tra una generazione ad un'altra si è avuto un esemplare della popolazione non voluto, cioè non accettabile per la soluzione (valore al di fuori dell' intervallo $[-100, 100]$), gli si dà un costo pari a 0, in modo tale che per la creazione della prossima generazione non verrà preso in considerazione; altrimenti il costo è dato dall'applicazione della F sul valore in ingresso.
L' algoritmo prima di tutto genera una popolazione iniziale scegliendo a caso valori presi nello spazio delle soluzioni, ne calcola i vari score e sceglie una

elite di individui (quelli che in questo caso massimizzano lo score), dopodiché, per un numero di generazioni volute, possiamo far mutare un elemento o fare il crossover tra due di essi in base ad una probabilità di mutazione da noi fornita.
Abbiamo però riscontrato un problema nella rappresentazione numerica: i dati vengono passati all' algoritmo sotto forma di float, non permettono di definire un' operazione di mutazione o crossover semplice. Per ovviare a questo inconveniente i numeri vengono convertiti nella loro forma binaria (in questo caso a 32 bit), definendo quindi la mutazione come il cambiamento di un bit da 1 a 0, o viceversa.
Il crossover è eseguito scegliendo un numero di bit casuale da una soluzione e il numero restante di bit da un'altra, affinché si riottenga un codice di 32 bit. Possiamo vedere il crossover come un'operazione di ricombinazione genetica in cui parte del materiale genetico di un individuo si ricombina con un altro. Così facendo, ad ogni cambio generazionale, si genera una progenie come combinazione di geni dei genitori migliori della generazione precedente, attuando in tal modo una selezione naturale.
La mutazione, invece, è un fattore casuale che permette di introdurre nuovi tratti negli individui, di variarne i geni, donando in tal modo alla progenie caratteristiche distintive non presenti nei genitori e consentendo alla specie di attuare un processo evolutivo che favorisca la sopravvivenza.



Come si può vedere, il valore che ottimizza la F(x) tende a x=20. Il risultato si riesce a trovare anche partendo da una popolazione iniziale ristretta a soli 100 elementi. Per un numero di iterazioni discreto (100 in questo caso) la soluzione non solo viene trovata, ma viene confermata più volte, dimostrando che, anche se si provano nuove combinazioni, l'esemplare che massimizza la fitness è stato già trovato precedentemente e le iterazioni successive non fanno altro che confermarlo.

2.2 Esercizio 2: MDS: Visually Exploring US Senator Similarity

Clustering

Il Clustering, o analisi di raggruppamento, è una tecnica di intelligenza artificiale volta alla selezione e raggruppamento di elementi omogenei in un insieme di dati. Tutte le tecniche di clustering si basano sul concetto di distanza tra due elementi, che ci permette di definire il concetto di somiglianza tra gli elementi. Infatti l'appartenenza o meno ad un insieme dipende da quanto l'elemento preso in esame sia distante dall'insieme. La bontà delle analisi ottenute dagli algoritmi di clustering dipende dalla metrica: metriche diverse porteranno quasi certamente a cluster differenti.

MDS: MultiDimensional Scaling

Il MultiDimensional Scaling è una tecnica di analisi statistica usata per mostrare graficamente le differenze o somiglianze tra elementi di un insieme. È una generalizzazione del concetto di ordinamento: partendo da un matrice contenente le informazioni che vogliamo analizzare e moltiplicandola per la sua trasposta, otterremo una matrice quadrata reppresentante la "somiglianza" di ogni elemento con un altro, dopodiché l'algoritmo di scaling multidimensionale assegnerà a ogni soggetto una posizione in uno spazio N-dimensionale, con N stabilito a priori tramite una metrica.

Riflessioni

L'esempio affrontato descrive l'orientamento politico dei membri dei Congressi americani, dal 101-esimo al 111-esimo, sull' analisi dei voti. Sono utilizzati i dataset in formato *STATA*, programma commerciale di calcolo statistico diffuso tra gli accademici nell'ambito delle scienze politiche. Essi presentano i voti delle differenti sedute dei 10 Congressi presi in esame. La metrica utilizzata per la distanza è quella euclidea. Essendo i dati troppo

complessi e strutturati per i nostri scopi, per facilitarne l' analisi viene eseguita una semplificazione aggregando dei tipi di voto: tra tutte le possibili classi di votazione (nove in totale) se ne ricavano solo tre.
Viene dapprima analizzato il caso del 110-esimo Senato. Notiamo che gli schieramenti dei voti sono netti: Repubblicani a destra e Democratici a sinistra. Estendendo l'analisi a tutti i Congressi, constatiamo che le scelte di voto sono consistenti con il partito di appartenenza. Alcuni grafici possono sembrare meno polarizzati, caratterizzati da punti più centrali, come se i voti siano stati meno radicali, ma tale risultato si ottiene perché non si ha una scala univoca che definisca la distanza.

2.3 Esercizio 3: Constraint Satisfaction Problems

Il problema di soddisfacimento dei vincoli che ci ritroviamo ad affrontare è il seguente: "Un agente intelligente deve colorare le province della regione Campania evitando che le confinanti abbiano lo stesso colore". Questo è un tipico esempio di domanda a cui si può rispondere con algoritmi che tengono conto del soddisfacimento dei vincoli. Come prima cosa formalizziamo il problema :

Variabili: Caserta, Benevento, Napoli, Avellino, Salerno
Vincoli: province confinanti devono avere colori diversi

Possiamo, dunque, ricavare il grafo dei vincoli, i cui nodi sono le variabili ed i cui archi connettono coppie di attributi che partecipano ad un vincolo. In questo caso i vincoli sono binari, cioè sono coinvolti soltanto due nodi per ogni vincolo.

Analizziamo il caso in cui l'agente disponga di solo 2 colori.

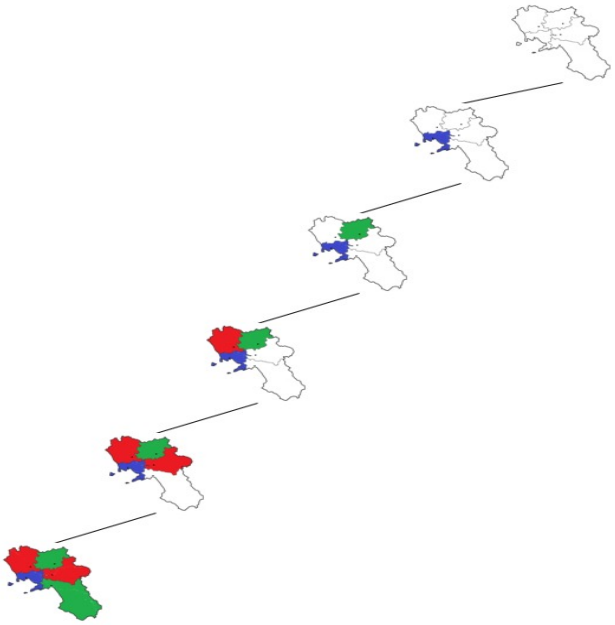
$$\text{Dominio} = \{\text{Colore1}, \text{Colore2}\}$$

Questo significa che le variabili possono assumere solo due colori. Da uno sguardo attento del grafo possiamo formalizzare un ulteriore vincolo, non binario: le variabili Napoli, Benevento e Avellino devono essere necessariamente tutte diverse (non sono l'unica tripla soggetta a questo vincolo). Un dominio formato da solo due colori non ci permetterà mai di poter soddisfare questo vincolo, pertanto il problema non è risolvibile. Ma, oltre questa formulazione informale, abbiamo provato anche l'algoritmo di `Backtracking Search`, attraverso il quale si perviene presto alla conclusione dell'impossibilità di risolvere il problema vincolato. Ad esempio, applicando le euristiche per migliorare l'efficienza della ricerca, assegniamo a Napoli (`Degree Heuristic`: variabile con più vincoli sulle rimanenti variabili) il valore Colore1; se procediamo assegnando Colore2 a Caserta (o qualsiasi altra variabile dato che ora presentano tutte un solo valore ammissibile), si ottiene un fallimento, poiché tramite il `Forward Checking` si osserva che Benevento non può assumere nessun valore ammissibile. Questa risultato è comune anche agli altri path dell'albero di ricerca e tutto ciò è intuitivamente comprensibile poiché con soli 2 colori non siamo in grado di soddisfare i loop di 3 nodi presenti nel grafo dei vincoli.

Analizziamo il caso in cui l'agente disponga di 3 colori.

$$\text{Dominio} = \{\text{Colore1}, \text{Colore2}, \text{Colore3}\}$$

Proviamo ora l'algoritmo con il dominio di dimensione maggiore. Partiamo da Napoli assegnandole il Colore1 e notiamo che questa scelta riduce i valori ammissibili per le altre province confinanti a due. Coloriamo, ora, Benevento con il Colore2; di conseguenza Caserta sarà la nostra prossima scelta poiché le si può assegnare solo un colore, il Colore3. Tale scelta è dettata dall'euristica che abbiamo applicato, il `Minimum Remaining Values`, che prevede la scelta della variabile con numero minimo di valori possibili. Ad Avellino, non confinante con Caserta, le possiamo assegnare il Colore3 senza produrre nessun conflitto. A Salerno, infine, assegniamo il Colore2. Osserviamo che ogni provincia ha un colore diverso dalle confinanti, ma queste ultime possono essere colorate allo stesso modo nel caso in cui non siano a loro volta adiacenti. Possiamo, in conclusione, affermare che è possibile soddisfare i vincoli dato questo dominio.



Anche il caso con quattro colori sicuramente permetterà di soddisfare i vincoli, perché si è aumentato il numero di valori ammissibili lasciando invariate le informazioni iniziali. Verifichiamo che sia così.

$$\text{Dominio} = \{\text{Colore1}, \text{Colore2}, \text{Colore3}, \text{Colore4}\}$$

La prima provincia che coloriamo è Napoli, assegnandole il Colore1: così facendo riduciamo tutti i colori possibili per le altre province, avendo Napoli il maggior numero di vincoli. Avellino sarà la nostra prossima scelta: assegnandole il Colore2 diminuirà anche il numero di valori per Salerno e Benevento. Passiamo a Caserta: essa può assumere tutti i valori tranne Colore1. Osserviamo, però, che l'assegnazione dei valori Colore3 e Colore4 produrrebbe una diminuzione del dominio degli attributi che possiamo assegnare a Benevento. Applichiamo il `Least Constraining Value`, euristica che predilige il valore che lascia più libertà alle variabili adiacenti, e scegliamo il Colore2 per Caserta. Segue Benevento con il Colore3 e Salerno con il Colore3. Si noti che, per le ultime due assegnazioni, si può effettuare una scelta completamente arbitraria tra Colore3 e Colore4.
Come ci si poteva aspettare, l'aggiunta di un nuovo elemento nel Dominio, senza l'alterazione dei vincoli, permette ugualmente la risoluzione del problema, perfino evitando di utilizzare il nuovo valore. Possiamo, allora, dedurre che, una volta trovata la soluzione con un determinato numero di valori, essa esisterà anche per un Dominio più esteso.
In quest'ultimo caso, però, si può parlare di vincoli più "elastici", più rilassati, grazie ai maggiori gradi di libertà per la determinazione della

soluzione. Ad esempio, con un Dominio di tre elementi il numero di soluzioni possibili sicuramente sarà molto più limitato rispetto ad uno con quattro elementi, che permette più opportunità di scelta. Possiamo, quindi, affermare che le soluzioni relative al Dominio esteso includano quelle del compatto.

2.4 Esercizio 4

Ricerchiamo un possibile approccio per trovare la soluzione con l' ausilio di algoritmi genetici al problema vincolato con $card(Dominio) = 4$. Prima di utilizzare gli algoritmi genetici per risolvere il problema, dobbiamo definire una funzione di costo. Tra le tante possibili scelte, abbiamo optato per la seguente:

$$\text{costo} \;=\; \text{numero di conflitti}$$

dove il conflitto è inteso come l'evento in cui due provincie confinanti hanno lo stesso colore. Una soluzione sarà quindi valutata rispetto ad una funzione di fitness che prende in considerazione il numero di conflitti: quanto più una soluzione presenterà un costo minore, tanto più assumerà un valore di fitness maggiore. Ovviamente ottimizzare la soluzione significa minimizzare il valore assunto dalla funzione di costo. Poiché essa può assumere solo valori positivi, la migliore soluzione riscontrabile avrà costo nullo.
Particolarizziamo l'algoritmo genetico per il problema in esame. La popolazione iniziale sarà costituita da un insieme di mappe che presentano una colorazione casuale delle province. Definiamo, poi, una probabilità di mutazione che consente di variare casualmente colore ad una provincia, tra i tre rimanenti (escludendo il colore attualmente assegnato). Ora non ci resta che definire l'operazione di crossover, che consiste nel prendere in modo casuale dall'élite due elementi (mappe colorate) e scegliere probabilisticamente il numero di province di uno, da unire con le restanti dell'altro.
Questi processi consentono la creazione della nuova generazione, dopodiché il tutto si ripeterà ciclicamente fino a che non si arrivi alla soluzione ottimale, o ad una sub-ottima nel caso il problema sia troppo complesso. Potrebbe, infatti, capitare che tutti gli elementi della popolazione convergano verso una soluzione che rappresenti solo un punto di massimo locale, e non globale, della funzione di fitness (soluzione a costo minimo relativo), quindi il risultato che potremmo ottenere utilizzando questo algoritmo non è detto sia ottimo e, conseguentemente, potrebbe non rappresentare una soluzione per il problema vincolato. Ciò non dovrebbe necessariamente sorprenderci considerando che il problema vincolato potrebbe anche non avere del tutto soluzione (come nel caso del problema della mappa con 2 colori).

Conoscendo il tipo di problema potremmo anche tentare di adattare l'algoritmo alle esigenze ad esso relative: modificandolo potremmo fare in modo che esso converga in tempo finito ad una soluzione significativa. Una prima modifica potrebbe consistere nell'impedire all'algoritmo di terminare fin quando non si sia giunti alla soluzione con costo pari a zero (soluzione del problema vincolato). Ovviamente tale modifica potrebbe fare in modo che il nostro algoritmo non termini mai! Per ovviare a questo inconveniente, potremmo invece pensare di introdurre una modifica che faccia ripartire l'algoritmo se non si sia giunti alla soluzione dopo un certo numero di iterazioni, o che lo arresti nel caso in cui la soluzione ci aggrada.

Work Project 3

3.1 Esercizio 1: Logica Proposizionale

La logica proposizionale è un linguaggio formale utilizzato dagli agenti logici per la rappresentazione della conoscenza. Un agente intelligente che sfrutta tale approccio si ispira al processo umano di deduzione a partire da avvenimenti noti che ha precedentemente appreso. Quindi l'agente avrà bisogno di una base di conoscenza (`KB: knowledge base`), in cui archiviare le informazioni utilizzate nella deduzione, e di un meccanismo di inferenza (`inference engine`) per effettuare le deduzioni.
L'agente basato sulla conoscenza, ogni volta che innesca il processo deduttivo, enuncia alla base di conoscenza la percezione corrente, dopodiché chiede ad essa l'azione da eseguire; la scelta avverrà in base alle informazioni archiviate. Comunica infine di aver eseguito l'operazione, così da garantire che l'evento scatenante il processo di ragionamento possa ampliare la propria base di conoscenza. Inizialmente l'agente può non essere a conoscenza di tutto ciò di cui necessita per completare la sua operazione, quindi sarà necessario una fase di raccolta delle informazioni iniziali. Possiamo fornire questa conoscenza al nostro agente in due modi: mediante un approccio dichiarativo vi è proprio una fase di raccolta di informazioni da parte dell'agente; nel caso di un approccio procedurale possiamo fornire noi una conoscenza iniziale all'agente. Solitamente i due approcci vengono usati insieme in modo tale da avere una buona base di conoscenza.
Le informazioni non possono essere salvate nella base di conoscenza dell'agente in un linguaggio naturale, o comunque in un qualsiasi modo da cui non sia possibile trarne delle conclusioni. Per tale motivo le frasi, o formule, che costituiscono il KB, sono espresse in un linguaggio formale di rappresentazione della conoscenza. Esso determina per le formule una sintassi per esprimerle in maniera corretta ed una semantica che ne definisce la verità rispetto ad ogni mondo possibile.
I mondi possibili sono gli ambienti, le condizioni in cui l'agente potrebbe venirsi a trovare. Mondi strutturati formalmente, rispetto ai quali è possibile

valutare la verità delle frasi, sono più propriamente detti *modelli*. Diciamo che m è un modello di una frase α se α è vera in m. Indichiamo, inoltre, come $M(\alpha)$ l'insieme di tutti i modelli di α.
La stessa identica frase in linguaggio naturale può essere tradotta in modi differenti in base alla diversa sintassi della logica utilizzata. I concetti fondamentali del ragionamento logico sono, però, indipendenti dalla forma particolare di logica. Determinata la base di conoscenza, da essa si possono riuscire a dedurre ulteriori informazioni attraverso la relazione di *conseguenza logica* (`entailment`):

$$KB \vDash \alpha \iff \alpha \text{ è vero in tutti i mondi in cui } KB \text{ è vero} \tag{3.1}$$

ovvero

$$KB \vDash \alpha \iff M(KB) \subseteq M(\alpha) \tag{3.2}$$

La conseguenza logica può essere applicata per derivare conclusioni, ovvero per eseguire *inferenze logiche*. Con la notazione $KB \vdash_i \alpha$ si intende che α è derivato da KB attraverso l'algoritmo di inferenza i. Una procedura di inferenza si dice corretta (`soundness`) se deriva solo formule che sono conseguenze logiche ($KB \vdash_i \alpha \implies KB \vDash \alpha$), completa (`completeness`) se può derivare ogni formula che è conseguenza logica ($KB \vDash \alpha \implies KB \vdash_i \alpha$).
La logica proposizionale è una logica semplice in cui la sintassi è composta da frasi atomiche, o simboli proposizionali, legate tra loro tramite connettivi logici per ottenere formule più complesse. Un simbolo proposizionale può assumere il valore vero o falso e le regole per determinare il valore di verità di una formula, rispetto ad un particolare modello, sono definite dalla semantica:

- $\neg P$ è vero se e solo se P è falso
- $P \wedge Q$ è vero se e solo se P e Q sono entrambi veri
- $P \vee Q$ è vero se e solo se P o Q è vero
- $P \Rightarrow Q$ è falso se e solo se P è vero e Q è falso
- $P \Leftrightarrow Q$ è vero se e solo se P e Q sono entrambi veri o entrambi falsi

Queste regole possono anche essere espresse con le *tabelle di verità*, che esprimono i valori di verità di una frase complessa per ogni possibile configurazione dei suoi simboli proposizionali.
Come procedura di inferenza possiamo ricorrere ad un approccio per enumerazione (`model checking`): enunciamo tutti i possibili modelli, tramite la tabella di verità, e verifichiamo che α sia vera in ogni modello in

cui KB è vera. I modelli sono rappresentati da un assegnamento di valori vero o falso ad ogni simbolo proposizionale. Questo algoritmo è corretto e completo, ma è molto oneroso in quanto i possibili mondi sono tanti (crescono esponenzialmente) ed enumerarli tutti comporterebbe una notevole complessità spaziale e temporale.
Per tale ragione si preferiscono altri approcci basati sull'applicazione delle regole inferenziali. Esaminiamo l'algoritmo di risoluzione, strettamente legato al concetto di *soddisfacibilità*. Una frase è soddisfacibile se è vera in qualche modello, viceversa insoddisfacibile se non è vera in nessun modello. La soddisfacibilità è connessa all'inferenza dalla:

$$KB \models \alpha \iff KB \wedge \neg\alpha \text{ è insoddisfacibile} \tag{3.3}$$

La procedura inferenziale tenterà, dunque, di provare α attraverso una *reductio ad absurdum*, ovvero dimostrando che $KB \wedge \neg\alpha$ è falsa in ogni modello. Prima di tutto dobbiamo trasformare le frasi complesse nella forma *CNF* (`Conjunctive Normal Form`), cioè come congiunzioni di clausole (disgiunzioni di letterali), attraverso le regole di equivalenza logica. Si cerca, poi, di dimostrare che la congiunzione della base di conoscenza con il negato della frase da dedurre sia insoddisfacibile, cioè non esista nessun modello in cui KB non implichi α. Se ciò fosse provato, significherebbe che in tutti i mondi in cui è vera la base di conoscenza, sarebbe vera anche la frase. Si considerano, allora, coppie di clausole della frase complessa $KB \wedge \neg\alpha$ alle quali si applica la regola di risoluzione inferenziale: si produce una nuova clausola che contiene tutti i loro letterali tranne quelli complementari (l e $\neg l$). Si provvede ad applicare questo ragionamento ricorsivamente fino a che non si riescono più a generare nuove clausole, ottenendo esito negativo, oppure si arriva alla clausola vuota, rappresentante la deducibilità della frase dalla base di conoscenza. Anche questo algoritmo è corretto e completo per la logica proposizionale.
In ogni caso l'agente non conosce il significato della frase, che le può essere attribuito, invece, dall'essere umano: l'agente è solo capace di applicare le regole inferenziali.

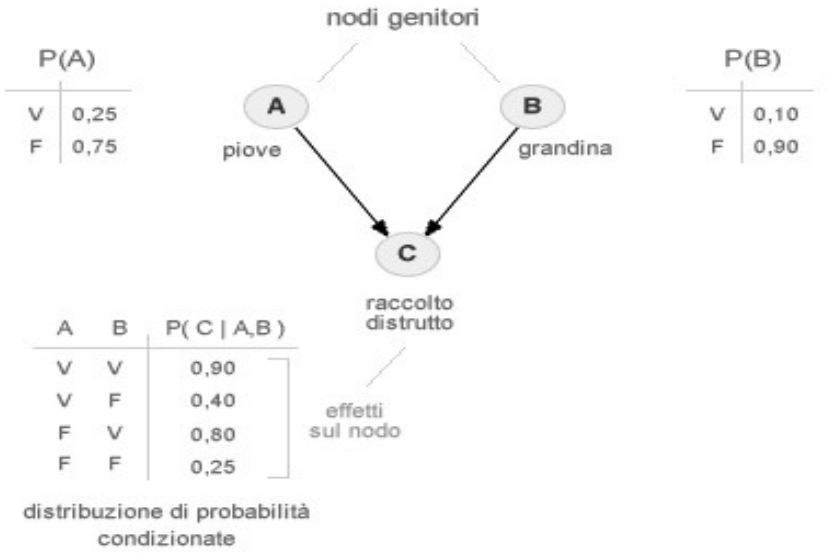
3.2 Esercizio 2: Reti Bayesiane

L'approccio probabilistico, rispetto alle altre tipologie, tiene conto del grado di incertezza. Infatti, a differenza di un approccio logico in cui un fatto può essere vero o falso, nell'approccio probabilistico si considera la probabilità dell'avvenimento. Tale livello d'incertezza non ci assicura a priori la verità di un'affermazione: affermando che oggi pioverà con una probabilità di 0.99,

potrebbe capitare che non piova, proprio perché la probabilità definisce il livello di incertezza rispetto ad una conoscenza da noi acquisita che varia da persona a persona (agente ad agente), dipendente da esperienze diverse che determinano valori di incertezza diversi. Le reti Bayesiane si rifanno al concetto di probabilità, più propriamente alla regola di Bayes così definita:

$$P(causa|effetto) = \frac{P(effetto|causa) * P(causa)}{P(effetto)} \tag{3.4}$$

Una rete Bayesiana di solito è fatta in questo modo:



Come possiamo osservare, ci sono dei nodi che determinano la variabile aleatoria da tenere in considerazione, a cui si assegnano anche i relativi valori di probabilità a priori, e dei collegamenti che indicano la relazione di causalità tra i nodi, rappresentati dagli archi orientati.
Per realizzare tale rete dobbiamo tenere conto delle relazioni di causalità delle nostre variabili, così facendo otteniamo una rete che riesce ad esplicare bene le nostre conoscenze. Una peculiarità di questa rete è la compattezza, infatti qualsiasi altro modo di realizzare la rete ne aumenterebbe la complessità. Una rete così realizzata permette di effettuare calcolo delle probabilità in modo semplice, perché in una rete siffatta vale l' indipendenza condizionata: una volta definito un valore per una variabile padre, le variabili figlie sono indipendenti tra di loro. Per esempio, se la variabile A è padre di B e C, la probabilità $P(B \wedge C|A) = P(B|A) * P(C|A)$; se non avessimo l'indipendenza, non potremmo scrivere i due domini separati e dovremmo tener conto anche della probabilità di B dato C. Per queste reti si definisce anche il concetto di

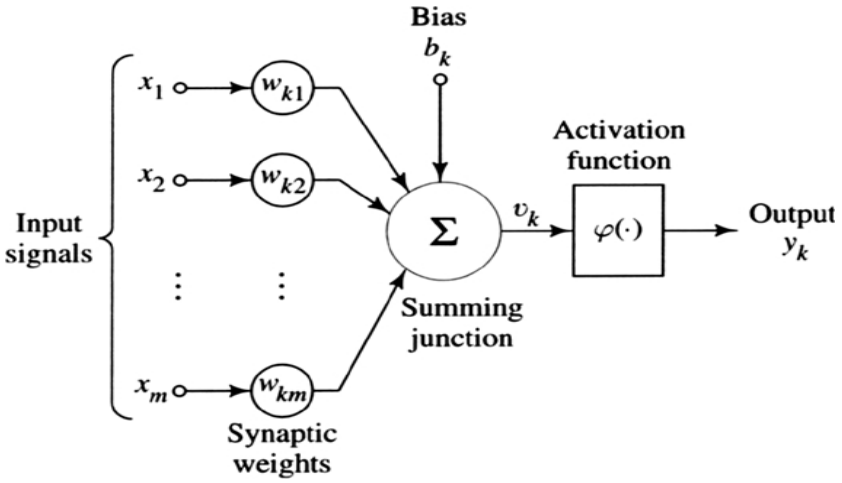
semantica locale, cioè ogni nodo è indipendente dai suoi non discendenti dato il nodo padre, e della coperta di Markov, ovvero un nodo è indipendente da tutti gli altri dato il nodo padre, i figli e i padri dei figli.
Per ricavare dati da queste reti, date le probabilità a priori, possiamo semplicemente calcolare tutti i valori di probabilità a posteriori che ci occorrono, metodo complesso nel caso in cui la rete sia grande e il numero di possibili valori delle variabili sia elevato. Di solito si può ricorrere all' approccio frequentista, che consiste nel generare un token che avrà una probabilità di ottenere un valore del nodo pari proprio all' incertezza legata ad esso, dopodiché si inizia a campionare la rete facendo scorrere il token su di essa, fissandone i valori con quelli risultanti dal campionamento. Le configurazioni ricavate, una volta raggiunta la quantità di risultati voluti, vengono divise per il numero dei campioni valutati, tenendo conto della loro molteplicità. Tale metodo per un alto numero di eventi determinati deve tendere all' approccio di calcolare le probabilità a posteriori. Le reti bayesiane così costruite sono statiche perché non tengono conto di un riferimento temporale. Le reti bayesiane dinamiche tengono conto anche di un valore temporale, suddiviso in istanti. In questo caso la variabile casuale non avrà solo relazioni con altre variabili, ma anche con se stessa valutata in istanti diversi di tempo. I valori di probabilità che di solito vengono ricavati su tali reti sono: il `filtering`, cioè la probabilità che avvenga un determinato evento all' istante t dati gli eventi precedenti e l' evidenza; lo `smoothing`, la probabilità di un evento in un istante t-k, con k che va da 0 a t, data l'evidenza fino all'istante t, utilizzato per correggere la probabilità relativa ad un fatto da un istante ad un altro; la `previsione`, la probabilità di un evento all' istante t+k, con k>1, dati gli eventi e l'evidenza fino a t; la `spiegazione migliore`, la sequenza di stati che più probabilmente ha generato l'evidenza data.
Il `filtering` può essere facilmente calcolato ricorsivamente. Conoscendo il fatto iniziale ed essendo i processi stazionari, possiamo determinare il risultato del filtraggio all'istante precedente e calcolare il risultato grazie alla nuova evidenza. Questo è possibile per la stazionarietà dei processi, cioè i cambiamenti sono regolati da leggi immutabili nel tempo, ovvero la tabella di probabilità di un fatto non cambia rendendo tale relazione identica ad ogni avanzamento. Esiste anche una tecnica detta di `particle filtering`, che prevede un campionamento sul fatto iniziale, la propagazione di tali esempi sul campione successivo, la pesatura in base all'evidenza ed il ricampionamento per lo stato successivo, tenendo conto del nuovo peso dei campioni. Ovviamente per ottenere il valore di probabilità di una determinata configurazione dobbiamo dividere il numero di casi uguali per il totale di campionamenti effettuati, e per un alto valore di quest'ultimi la

probabilità tenderà a quella reale che avremmo ottenuto con il filtering. Tali reti possono essere usate anche nel machine learning. Dato un insieme di ipotesi, possiamo vedere qual è la probabilità associata ad esse data un'evidenza, cioè calcolare la probabilità di $P(hi|d)$ dove hi è l'ipotesi e d l'evidenza. Siamo interessanti alla probabilità di ogni ipotesi (sommatoria). Ad ogni nuova evidenza potrebbe variare la hi che massimizza la probabilità. Non è molto agevole portare avanti i calcoli per tutte le ipotesi, per tale ragione si sceglie quella che massimizza il valore a posteriori (`MAP`), cioè l'ipotesi che massimizza $\alpha * P(d|hi) * P(hi)$, dove α è la costante di normalizzazione. Nel caso in cui le ipotesi abbiano probabilità a priori uguali (stessa complessità), si può usare la `maximum-likelihood (ML)`, che consiste nel massimizzare $P(d|hi)$. Potremmo pensare all'uso dei logaritmi nel caso in cui compaiano esponenziali rendendo la massimizzazione della derivata molto più semplice.

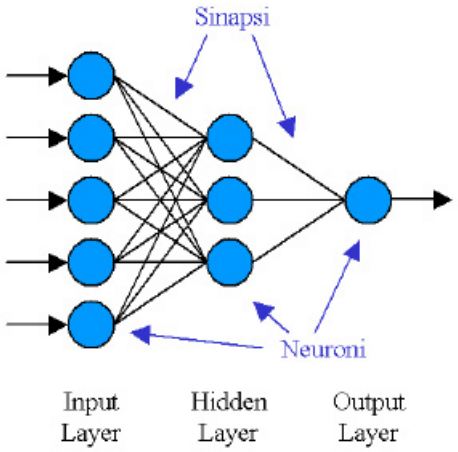
3.3 Esercizio 3: Reti Neurali

Le reti neurali sono un meccanismo computazionale che si ispira al funzionamento del cervello umano. Sono rappresentabili tramite un grafo orientato, i cui nodi costituiscono un modello matematico del neurone, e agli archi orientati è associato un peso. Ogni nodo i calcola la sua uscita a_i applicando una funzione di attivazione g, chiamata `activation function`, alla somma pesata degli ingressi.

$$a_i = g(in_i) = g(\sum_{j=0}^{n} W_{j,i} a_j) \tag{3.5}$$



La funzione di attivazione storicamente utilizzata era quella di *soglia*; in seguito si è diffusa la funzione *sigmoide*, continua e derivabile rispetto alla precedente.
Possiamo classificare strutturalmente le reti neurali distinguendo le `Recurrent Networks` e le `Feed-Forward Networks`. Le ultime non si fondano sul concetto di stato interno, a differenza delle prime. Inoltre possono essere suddivise in più livelli, o *layers*. Le reti neurali a singolo layer riescono a rappresentare solo operazioni *linearmente separabili*, a due layer tutte le funzioni continue, a tre layer anche le funzioni discontinue.



Una difficoltà non trascurabile risiede nella costruzione della struttura della rete neurale. Esse, infatti, sono sistemi `black-box`, il cui funzionamento interno è di difficile interpretazione logica, anche se è possibile averne una matematica che non aiuta nella sua costruzione.
Per determinare il peso degli archi si utilizza l'algoritmo di `Back-Propagation`. Inizialmente si costituisce una rete con pesi casuali o uniformi per abolire l'onere computazionale. Consideriamo poi una coppia ingressi-uscita del data-set. L'errore tra l'uscita ottenuta e quella desiderata verrà propagato all'indietro per ogni livello, così da modificare i pesi ed ottenere una migliore approssimazione della funzione. Tale propagazione fa uso del `learning rate` per far si che un singolo esempio non influisca interamente sui valori dei pesi, altrimenti questi tenderebbero ad oscillare di molto nel caso in cui ogni esempio fornisca un errore relativamente grande. La procedura verrà iterata per ogni esempio, ciascuno dei quali consentirà una correzione dei pesi, delineando una struttura della rete coerente con la funzione da approssimare. I problemi principali che riscontriamo in questo

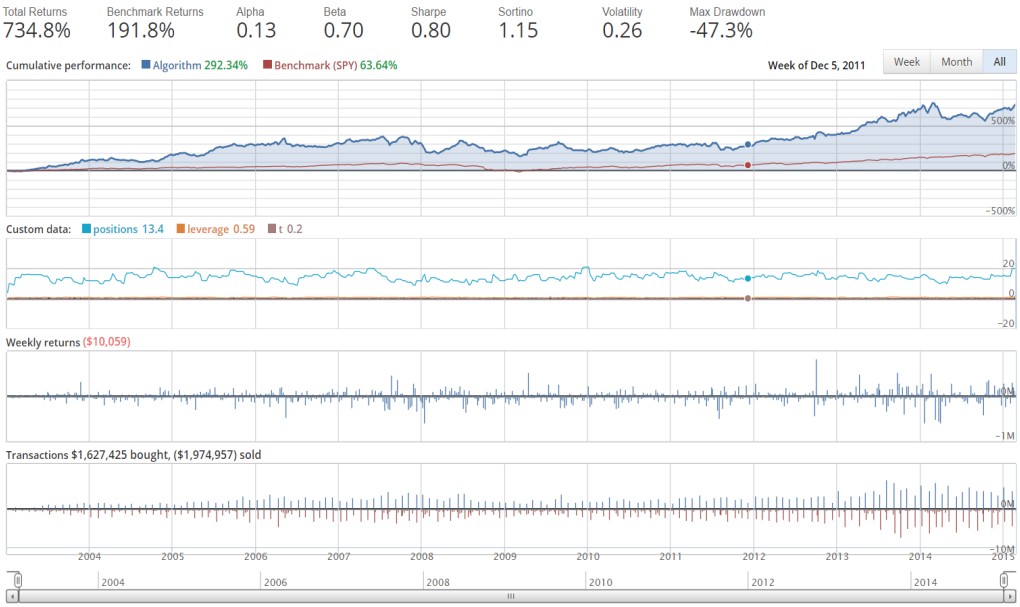
algoritmo sono l'eccessiva complessità temporale e la possibilità di arresto su di un minimo locale.
Per quanto riguarda, invece,la determinazione del numero di neuroni, si procede tipicamente con un approccio `trial and error`, perché non è chiaro precisamente il significato della rete. Si parte, quindi, da reti semplici: se queste dopo la `Back-Propagation` funzionano bene, le si usa, altrimenti si iniziano a inserire nuovi neuroni o layer di neuroni e si continua a testare la validità delle nuove reti create.
In definitiva, le reti neurali si sono rivelate empiricamente un buon approssimatore di funzioni ed oggi il loro utilizzo ha ripreso vigore nella forma di `Deep Learning`. Tali tecniche ottengono un basso *error rate* nei problemi relativi al riconoscimento di caratteri e sono largamente diffuse nei sistemi multimediali per quanto concerne il riconoscimento di immagini, audio o nella `Computer Vision` in generale. Seppur presentando buoni risultati, bisogna ricordare che *non esiste un algoritmo di apprendimento migliore in assoluto, ma bisogna trovare quello più adatto al problema dato.*

3.4 Esercizio 4: Cloud e Crowdsourcing

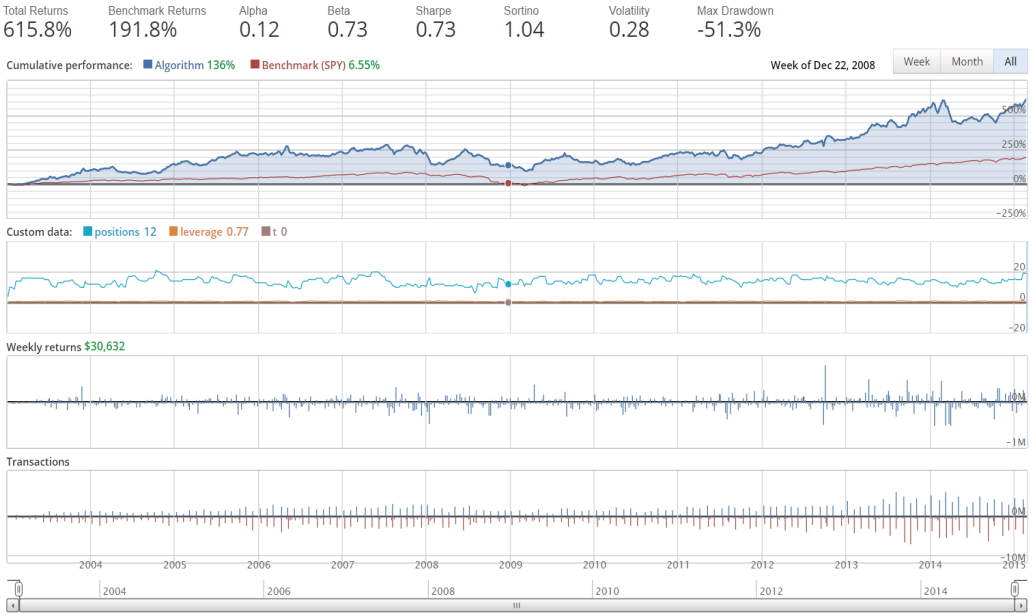
Il `Cloud Computing` ed il `Crowdsourcing` sono paradigmi affermatisi nel nuovo millennio, figli della dirompente rivoluzione della rete non solo al livello culturale, ma anche economico, fornendo nuovi modelli per sviluppare ed offrire servizi al cliente.
Il cloud prevede l'erogazione di risorse informatiche *on demand* attraverso Internet. Tipicamente, in linea del tutto generale, un `Cloud Provider` fornisce un *pool* condiviso di risorse; esse possono essere configurate da un cliente amministratore che le carica di valore aggiunto; infine l'utilizzatore finale usufruisce delle risorse in tal modo configurate per poi rilasciarle al termine del suo utilizzo.
Il crowdsourcing è lo sviluppo collettivo di un progetto da parte di volontari esterni al suo ideatore. Questo fenomeno deriva dalla nascita delle prime *open community* di condivisione, legate profondamente al concetto di *sharing*. Le *community open source* sono state le prime a sfruttare i benefici del crowdsourcing. Questo paradigma si è diffuso anche in ambito aziendale con il modello di business di *open enterprise*.
Quantopian applica questi concetti fornendo i servizi per scrivere algoritmi di trading e rendendoli disponibili all'intera community, nella quale si potranno condividere idee, codice e dati. Esso, inoltre, investe sugli algoritmi con le migliori performance, condividendone i profitti con l'autore. Rappresenta, dunque, un modello di business all'avanguardia, che sfrutta pienamente le

potenzialità della community, guidandola tramite i propri strumenti e ricavando introiti dall'impulso creativo a cui il crowdsourcing porta.
Tale piattaforma offre molti dataset direttamente, evitando l'esigenza di importarli da altre fonti. Nel nostro esempio vediamo operazioni di trading su azioni americane di una nostra possibile società, che ha un certo numero di azioni iniziali e decide quando vendere o comprare tali titoli per aumentare i suoi introiti partendo da un capitale iniziale ed osservando cosa accade in un determinato lasso di tempo.
Il codice fornisce tale output:

La linea rossa definisce il reale valore che si avrebbe avuto confrontandosi con il mercato, quella blu determina la previsione stimata dall'algoritmo, il valore position ci dice il numero di azioni attive, cioè non ancora chiuse, leverage ci dice quanto siamo disposti ad investire in base agli introiti, il guadagno settimanale è la transazione monetaria che c'è stata. Come si può vedere dall' esempio, le nostre scelte di trading secondo il nostro algoritmo sono molto rosee rispetto a ciò che è successo realmente, anche se a grandi linee, comunque, la previsione rispetta l'andamento avuto, dandoci però un guadagno falsato. La modifica dell'algoritmo, permessa a qualsiasi utente tramite cloud, viene eseguita da un *server farm*, sicuramente più performante di un pc qualunque. Difatti, in figura sono rappresentate le valutazioni del mercato americano di circa 12 anni, un onere di computazione non banale.
Una possibile modifica dell'algoritmo è puntare solo sulle azioni che ci danno un valore di utile maggiore:



Come vediamo, l'algoritmo si è comportato meglio, perché sono state prese solo le 3000 azioni che davano un'utilità sperata maggiore, quindi abbiamo fatto trading su azioni che ci permettono di massimizzare i guadagni. Questa è un operazione che di solito premia a lungo termine. Non è sempre detto, però, che un'azione abbia il comportamento sperato, perchè può capitare che la sua efficienza sia minore a causa di fattori non ancora accaduti, che potrebbero in effetti portare ad un'utilità diversa. Un altro esperimento è stato quello di aumentare il profitto che si voleva fare con una determinata azione, tendendo quindi a cercare di sfruttare quanto più quell'azione per massimizzare i guadagni. Il risultato è peggiorato:



Possiamo vedere che il numero di transazioni è minore rispetto ai due casi precedenti, appunto perché una volta che si è iniziato a fare un trade su quell'azione, fino a che non ci porta al guadagno voluto o scende al di sotto di un valore minimo, non la vendiamo. In tal modo abbiamo cercato di attuare una politica più conservativa sulla compravendita, che ci ha portato a dei ricavi minori, ma ad un atteggiamento più consono a quello che in realtà è successo.