

# **DIABETES**

*Tesina sull'apprendimento - Intelligenza Artificiale*  
*Cosimo Cecchi, Tommaso Visconti*

# Indice

Cap. 1 - Descrizione del problema.....	<i>pag. 3</i>
Cap. 2 - Preprocessing dei dati.....	<i>pag. 4</i>
Cap. 3 - Suddivisione dei dati.....	<i>pag. 5</i>
Cap. 4 - Addestramento della rete.....	<i>pag. 5</i>
Cap. 5 - Selezione dell'architettura.....	<i>pag. 6</i>
Cap. 6 - Test della rete.....	<i>pag. 11</i>
Cap. 7 - Valutazione tramite 3-fold cross validation.....	<i>pag. 12</i>
Cap. 8 - Confronto con alberi di decisione.....	<i>pag. 13</i>
Cap. 9 - Conclusioni.....	<i>pag. 15</i>
Cap. 10 - Appendice.....	<i>pag. 16</i>
Cap. 10.1 - Randomizer.....	<i>pag. 16</i>
Cap. 10.2 - Result parser.....	<i>pag. 16</i>
Cap. 10.3 - Normalizer.....	<i>pag. 19</i>

# 1 Descrizione del problema

Il dataset *diabetes*, costruito dal *National Institute of Diabetes and Digestive and Kidney Diseases* nel 1990, raccoglie informazioni sull'incidenza di vari parametri medici nel manifestarsi del diabete. I casi analizzati nel dataset riguardano donne di almeno 21 anni di etnia Pima, una particolare etnia degli indiani nativi d'America, presenti in una zona che all'incirca corrisponde all'odierno Arizona.

Il dataset è composto da 768 esempi, tutti riguardanti soggetti di sesso femminile; gli attributi presi in considerazione sono 8, mentre le classi di uscita sono due, e rappresentano la presenza o meno di diabete nel soggetto in questione. Gli attributi sono tutti di tipo continuo, e non vi è alcun attributo mancante negli esempi.

Nelle due tabelle sottostanti sono riportate rispettivamente la percentuale di esempi per ciascuna classe ed una breve descrizione degli 8 attributi.

Valore di uscita	N. esempi	Percentuale
1 (test positivo)	500	65.1%
2 (test negativo)	268	34.9%

Tabella 1: suddivisione degli esempi in classi di uscita.

Attributo	Media	Deviazione Std
Numero di gravidanze	3.8	3.4
Concentrazione del glucosio nel plasma	120.9	32.0
Pressione distolica (mm Hg)	69.1	19.4
Spessore della pelle sui tricipiti (mm)	20.5	16.0
Insulina ( $\mu$ U/ml)	79.8	115.2
Indice di massa corporea	32.0	7.9
Variabile di ereditarietà diabetica	0.5	0.3
Età	33.2	11.8

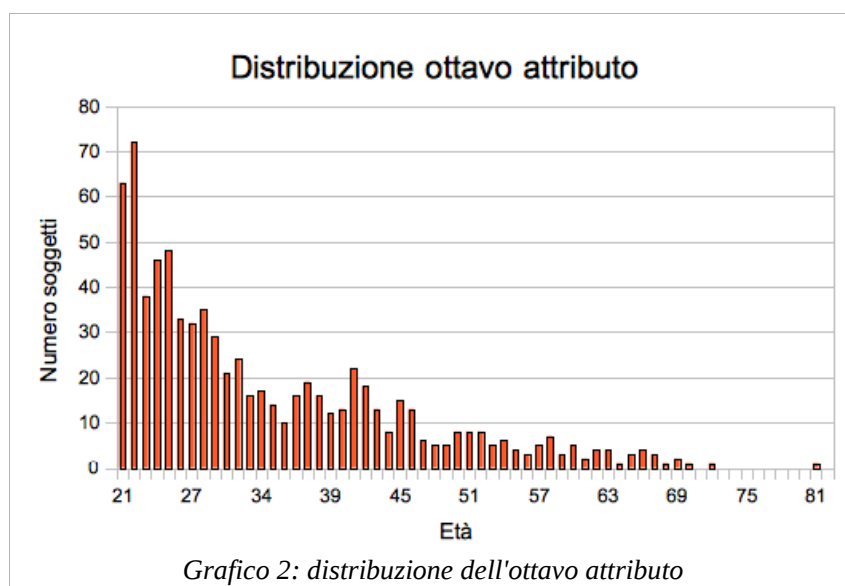
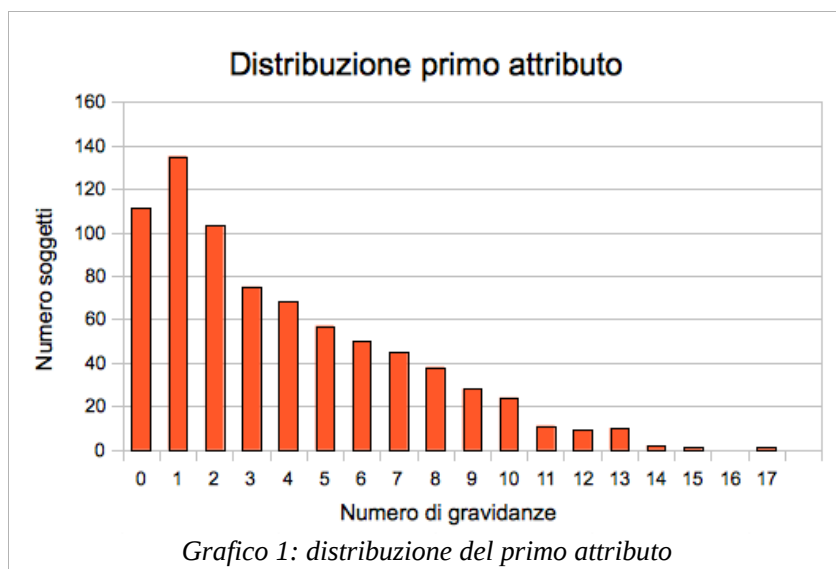
Tabella 2: attributi del dataset e dati statistici

## 2 Preprocessing dei dati

Il dataset ci è stato fornito in formato compatibile con *C4.5*, diverso da quello del simulatore di reti neurali *JavaNNS* che abbiamo in seguito usato. E' stato quindi necessario trattare i dati prima di procedere all'analisi e all'implementazione della rete neurale.

Inizialmente abbiamo esaminato la distribuzione dei valori dei vari attributi, per determinare la giusta funzione da utilizzare per la normalizzazione del dataset. Com'è possibile vedere dai due grafici seguenti, i dati non seguono una distribuzione di tipo gaussiano. Pertanto, essendo gli attributi tutti continui, ci siamo limitati a normalizzare i loro valori nell'intervallo [0,1], con la formula:

$$\bar{v} = \frac{v - v_{min}}{v_{max} - v_{min}}$$



Per quanto riguarda le due classi di uscita, il dataset *C4.5* associava i valori di positività e negatività del test del diabete ai valori *1* e *2*, mentre *JavaNNS* utilizza *0* e *1*. Abbiamo quindi effettuato una traslazione delle classi di output verso questi due valori. Nella rete neurale, il nostro layer di output sarà composto soltanto da un neurone.

### 3 Suddivisione dei dati

---

Dopo aver reso i dati utilizzabili da *JavaNNS* è necessario dividerli in tre subset (in formato PAT) per l'addestramento della rete (le percentuali tra parentesi si considerano in riferimento al numero totale degli esempi)

- training set (50%) – insieme di esempi utilizzati per l'apprendimento
- validation set (25%) – insieme per effettuare un pre-test sulla rete
- test set (25%) – utilizzato successivamente alla scelta della miglior architettura per la rete neurale, per verificarne l'accuratezza

Ogni set deve contenere esempi positivi e negativi nella stessa proporzione presente nel dataset originale. Nel nostro caso, come abbiamo ricordato in Tabella 1, vi sono il 65.1% di esempi positivi; il training set è quindi composto da 250 esempi positivi e 134 esempi negativi, mentre gli altri due set avranno entrambi 125 esempi positivi e 67 negativi, mantenendo la stessa proporzione.

Per effettuare questo procedimento, successivamente alla normalizzazione abbiamo effettuato i seguenti passaggi:

- separazione degli esempi nelle due classi di uscita
- randomizzazione degli esempi all'interno dei due insiemi ottenuti
- costruzione dei tre set, prelevando le giuste percentuali di esempi dai due insiemi randomizzati
- aggiunta dell'header PAT e salvataggio finale

Per velocizzare queste operazioni meccaniche, abbiamo scritto una piccola collezione di script in linguaggio Ruby, che è possibile trovare in appendice.

### 4 Addestramento della rete

---

Addestrare la rete è la parte più importante dell'esercitazione, e consiste nel trovare la configurazione neuronale ottimale. Tale ricerca avviene in maniera empirica, modificando durante i test i vari parametri della configurazione. Tali parametri includono:

- *numero di layer nascosti e numero di neuroni* per ognuno di essi
- *algoritmo di apprendimento*
- *learning rate*

- *inizializzazione dei pesi dei rami*
- *variabili specifiche dell'algoritmo scelto (e.g. Momentum)*

*JavaNSS* facilita la ricerca della rete ottimale, mettendo a disposizione un'interfaccia grafica dalla quale, dopo aver costruito l'architettura base della rete (scelta dei *layer* e del numero di neuroni) è possibile variare tutti gli altri parametri di apprendimento in modo immediato.

Il primo passo nella ricerca della migliore configurazione avviene lanciando l'algoritmo di apprendimento sui dati di training e validation, e osservando l'andamento del grafico degli errori, calcolato secondo la formula:

$$E = \frac{1}{2n} \sum_{k \in \text{Output}} \sum_{d \in D} (t_{kd} - o_{kd})^2$$

È importante fermare l'apprendimento non appena si verifica un fenomeno di *overfitting*: questo accade quando la rete inizia a perdere di generalità e tende ad uniformarsi ai dati del *training set*. Proprio per questo motivo l'apprendimento include anche il *validation set*, il cui scopo è quello di testare preventivamente la risposta della rete su dati non direttamente usati per l'apprendimento. Infatti, quando la curva dell'errore del *validation set* raggiunge un minimo per poi aumentare, siamo all'inizio dell'*overfitting*, ed è necessario interrompere l'apprendimento. Questa tecnica prende il nome di *Early stopping*.

## 5 Selezione dell'architettura

Per studiare la risposta della rete al variare dei parametri di configurazione, abbiamo deciso di seguire uno schema prefissato, formato da tre serie di test, nelle quali varia il numero di neuroni hidden contenuti nella rete (0, 8 e 15).

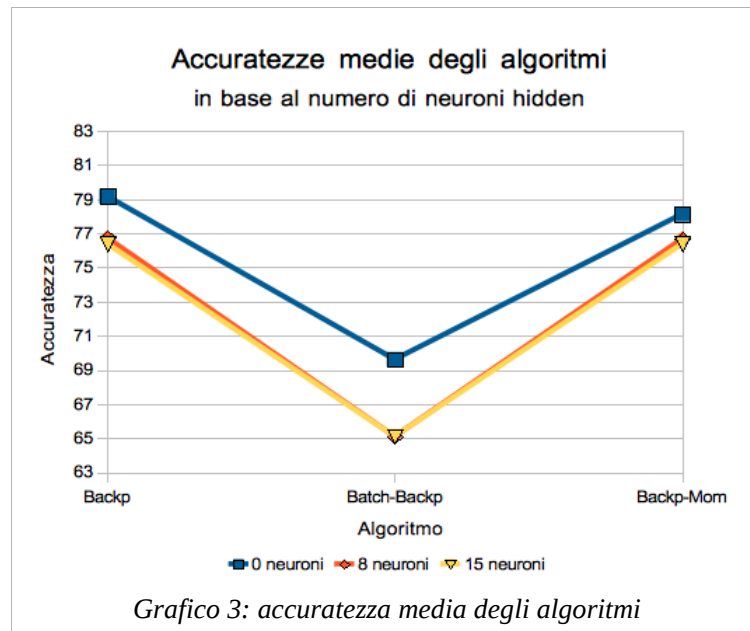
In ciascuna serie, abbiamo testato i tre algoritmi di apprendimento (*backpropagation*, *batch-backpropagation* e *backpropagation-momentum*) variando per ciascuno i parametri principali all'interno di un range che abbiamo scelto dopo alcuni test preliminari, nei quali abbiamo osservato che la rete apprende in maniera migliore, per valori di  $\eta < 0.05$  (lo stesso abbiamo fatto per  $\mu$  nel caso di algoritmo *backpropagation-momentum*).

#	Algoritmo	$\eta$	$\mu$	Hidden	Accuratezza %
1	backprop	0.05	-	0	78,65
2	backprop	0.01	-	0	78,65
<b>3</b>	<b>backprop</b>	<b>0.001</b>	-	<b>0</b>	<b>80,21</b>
4	batch-backp	0.05	-	0	77,60
5	batch-backp	0.01	-	0	66,15
6	batch-backp	0.001	-	0	65,10
7	backp-mom	0.05	0.1	0	73,96
8	backp-mom	0.05	0.01	0	73,44
9	backp-mom	0.05	0.001	0	79,17

#	Algoritmo	$\eta$	$\mu$	Hidden	Accuratezza %
10	backp-mom	0.01	0.1	0	78,65
11	backp-mom	0.01	0.01	0	78,65
12	backp-mom	0.01	0.001	0	78,65
13	<b>backp-mom</b>	<b>0.001</b>	<b>0.1</b>	<b>0</b>	<b>80,21</b>
14	<b>backp-mom</b>	<b>0.001</b>	<b>0.01</b>	<b>0</b>	<b>80,21</b>
15	<b>backp-mom</b>	<b>0.001</b>	<b>0.001</b>	<b>0</b>	<b>80,21</b>
16	backprop	0.05	-	8	76,56
17	backprop	0.01	-	8	75,00
18	backprop	0.001	-	8	78,65
19	batch-backp	0.05	-	8	65,10
20	batch-backp	0.01	-	8	65,10
21	batch-backp	0.001	-	8	65,10
22	backp-mom	0.05	0.1	8	77,08
23	backp-mom	0.05	0.01	8	72,40
24	backp-mom	0.05	0.001	8	75,52
25	backp-mom	0.01	0.1	8	76,04
26	backp-mom	0.01	0.01	8	76,56
27	backp-mom	0.01	0.001	8	76,56
28	backp-mom	0.001	0.1	8	78,65
29	backp-mom	0.001	0.01	8	78,65
30	backp-mom	0.001	0.001	8	78,65
31	backprop	0.05	-	15	75,00
32	backprop	0.01	-	15	75,52
33	backprop	0.001	-	15	78,65
34	batch-backp	0.05	-	15	65,10
35	batch-backp	0.01	-	15	65,10
36	batch-backp	0.001	-	15	65,10
37	backp-mom	0.05	0.1	15	76,04
38	backp-mom	0.05	0.01	15	74,48
39	backp-mom	0.05	0.001	15	75,52
40	backp-mom	0.01	0.1	15	76,04
41	backp-mom	0.01	0.01	15	75,00
42	backp-mom	0.01	0.001	15	75,00
43	backp-mom	0.001	0.1	15	78,65
44	backp-mom	0.001	0.01	15	78,65
45	backp-mom	0.001	0.001	15	78,12

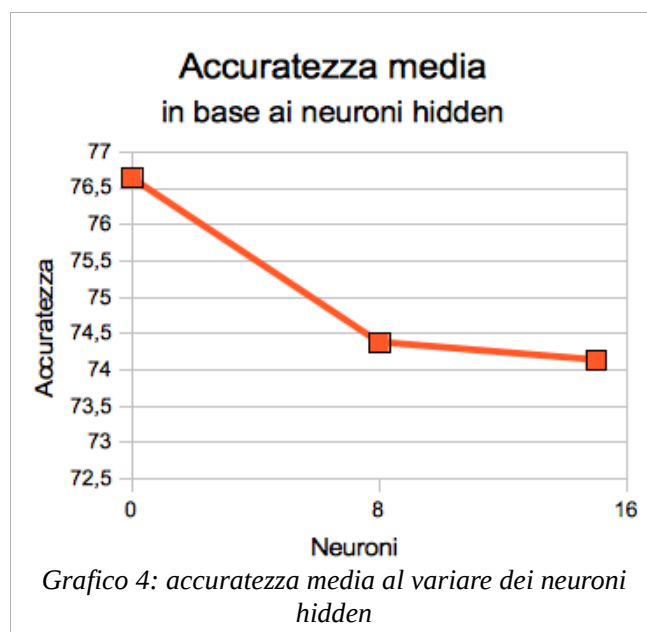
Tabella 3: Risultati dell'apprendimento

Prima di analizzare le architetture di rete che hanno ottenuto la migliore accuratezza, vogliamo mettere in evidenza alcuni dati che emergono dai risultati complessivi dei nostri test.



Dal grafico 3 si nota che l'accuratezza dell'apprendimento della rete, al variare del numero di neuroni hidden, varia proporzionalmente con l'algoritmo utilizzato (questo si evince dal fatto che le tre linee in grafico sono quasi parallele). Ciò sta a significare che, nonostante l'accuratezza media cambi al variare dei neuroni, essa è consistente con l'algoritmo utilizzato.

Analizziamo adesso l'impatto che il numero di neuroni hidden ha complessivamente sull'accuratezza della rete.

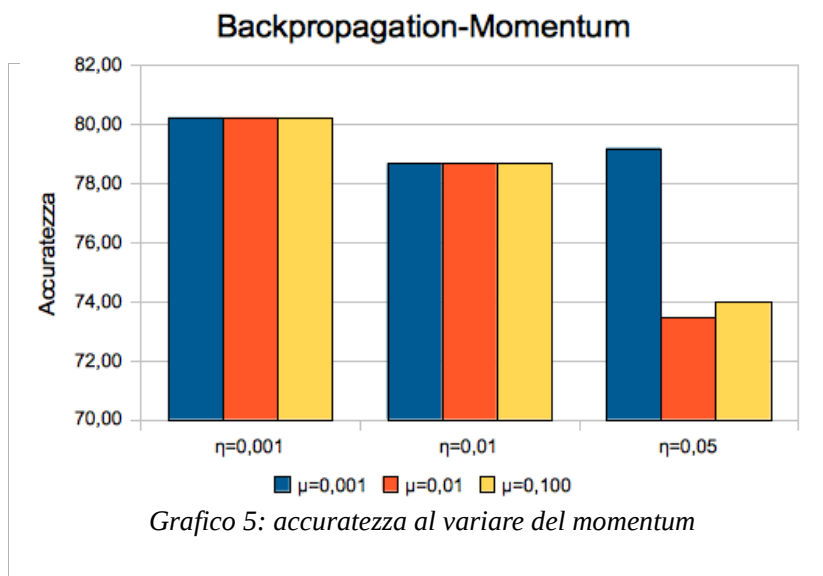


Il grafico 4 mette in evidenza il fatto che i risultati migliori di accuratezza si hanno, in media, quando non vi sono neuroni hidden. Questo, unito al dato fornitoci dal grafico precedente, giustifica l'esclusione dall'analisi delle reti con 8 e 15 neuroni, senza perdere di generalità.

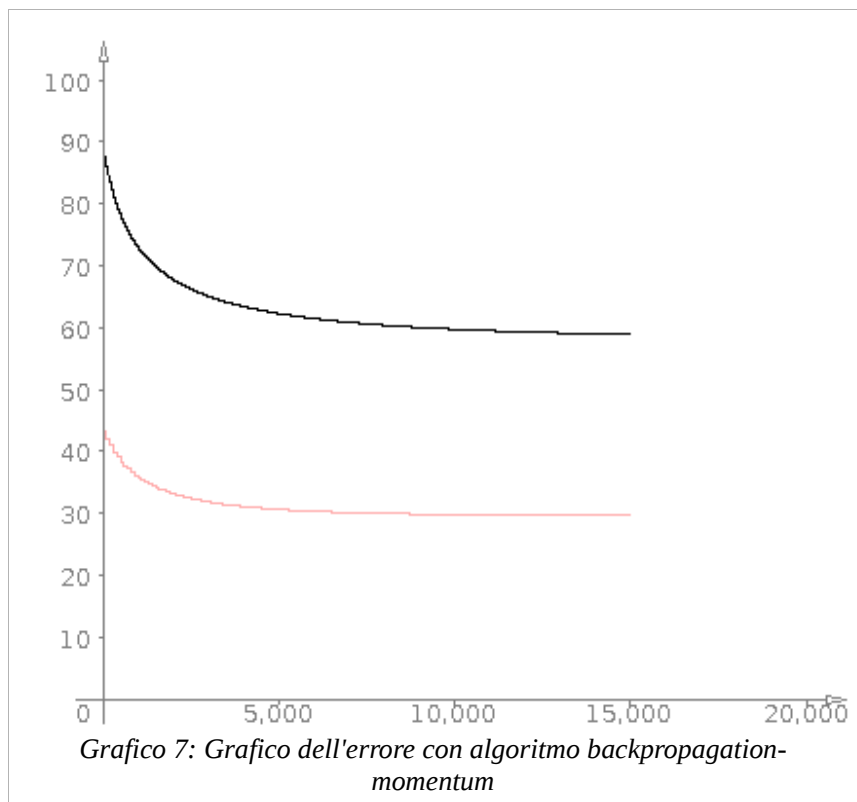
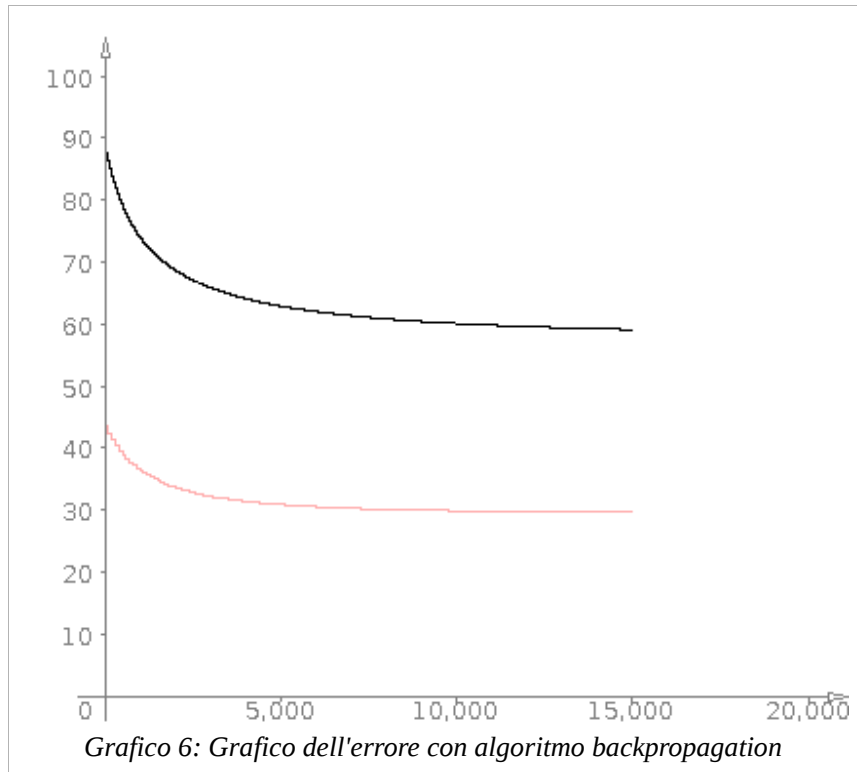
I migliori risultati di accuratezza (80,21%) sono stati ottenuti nelle seguenti configurazioni, tutte con zero neuroni hidden:

- *backpropagation*
  - $\eta = 0,001$
- *backpropagation-momentum*
  - $\eta = 0,001$ 
    - $\mu = 0,1$
    - $\mu = 0,01$
    - $\mu = 0,001$

Questi risultati ci hanno indotto a pensare che nella nostra configurazione, il *momentum* non apporti alcune modifiche significative all'apprendimento della rete. Approfondendo l'analisi, come è possibile osservare dal grafico 5, vediamo infatti che il variare del *momentum* all'interno della stessa configurazione è significativo soltanto per  $\eta$  relativamente grandi.



Ecco quindi i grafici dell'errore ottenuti con JavaNNS utilizzando le configurazioni scelte.



---

## 6 Test della rete

---

Scelte quindi le migliori architetture, siamo passati a testare le reti con il dataset di *test*, che non era mai stato visto prima dalle stesse.

Questi sono i risultati ottenuti:

- *backpropagation*
  - accuratezza: 75,00 %
  - esempi corretti: 144
  - esempi errati: 48
- *backpropagation-momentum* ( $\mu = 0.1$ )
  - accuratezza: 75,00 %
  - esempi corretti: 144
  - esempi errati: 48
- *backpropagation-momentum* ( $\mu = 0.01$ )
  - accuratezza: 75,52 %
  - esempi corretti: 145
  - esempi errati: 47
- *backpropagation-momentum* ( $\mu = 0.001$ )
  - accuratezza: 75,00 %
  - esempi corretti: 144
  - esempi errati: 48

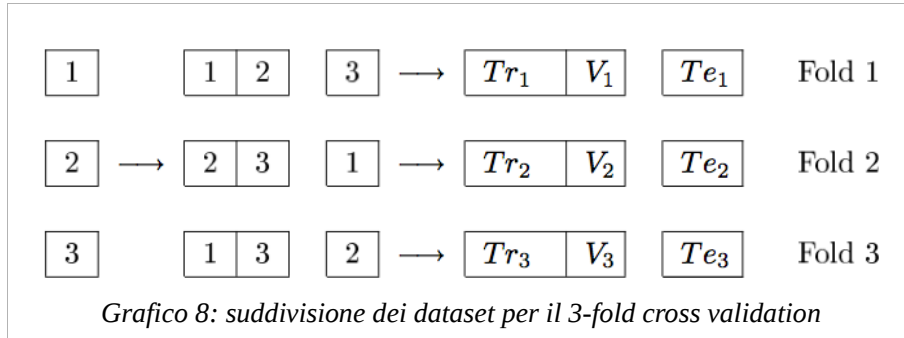
In virtù della maggiore accuratezza sul dataset di *test*, abbiamo scelto quindi la rete *backpropagation-momentum* ( $\mu = 0.01$ ) come migliore architettura per questo problema.

---

## 7 Valutazione tramite 3-fold cross validation

---

Per concludere l'analisi della rete neurale ottenuta, abbiamo effettuato la validazione dei risultati con il metodo del *3-fold cross validation*, ovvero abbiamo inizialmente diviso gli esempi normalizzati in tre set contenenti ciascuno lo stesso numero di esempi per classe. A questo punto i tre set, uniti a coppie formano i dati del *validation set* e del *training set*, mentre il terzo insieme rimanente forma, da solo, il *test set*.



In tabella 4 sono riassunti i risultati di accuratezza sul test set di ognuno dei tre fold.

Fold	Esempi totali	Esempi corretti	Accuratezza
1	255	192	75,29 %
2	255	194	76,07 %
3	255	188	73,72 %

Tabella 4: accuratezza del 3-Fold validation

Abbiamo infine calcolato l'accuratezza media e la deviazione standard dei dati così ottenuti.

- **Accuratezza media:**  $\bar{acc} = \frac{acc_1 + acc_2 + acc_3}{3} = 75,03$
- **Deviazione standard:**  $\delta_{acc} = \sqrt{\frac{acc_1^2 + acc_2^2 + acc_3^2}{3} - \bar{acc}^2} = 0,67$

Come possiamo vedere, i dati sono in accordo con l'accuratezza sul test set precedentemente calcolata e la deviazione standard risulta essere piuttosto limitata.

## 8 Confronto con alberi di decisione

Per concludere la nostra analisi applichiamo, a titolo di confronto, la stessa metodologia *3-fold cross-validation* utilizzando però gli alberi di decisione invece dell'approccio sub-simbolico delle reti neurali.

Negli alberi di decisione il fenomeno di overfitting viene limitato con la potatura, che trasforma alcuni sottoalberi in foglie, generalizzando in maniera ottimale la classificazione degli esempi non visti.

Ci siamo avvalsi dello strumento di analisi *C4.5*, che accetta in input i dati nello stesso formato da noi

ricevuto. L'unico pre-processing che abbiamo quindi eseguito, oltre alla divisione nei 3 *fold* con ugual numero di esempi per classe, è stato quello di randomizzare il contenuto di ciascun *fold* prima di darlo in ingresso al programma.

Di seguito è fornito l'output dell'elaborazione sui tre *fold*.

*Fold 1:*

```

Evaluation on training data (512 items):

      Before Pruning      After Pruning
      -----
Size      Errors  Size      Errors  Estimate
      55  67(13.1%)  49  69(13.5%)  (20.7%)  <<

Evaluation on test data (256 items):

      Before Pruning      After Pruning
      -----
Size      Errors  Size      Errors  Estimate
      55  81(31.6%)  49  80(31.2%)  (20.7%)  <<
    
```

*Fold 2:*

```

Evaluation on training data (512 items):

      Before Pruning      After Pruning
      -----
Size      Errors  Size      Errors  Estimate
      43  76(14.8%)  39  77(15.0%)  (21.6%)  <<

Evaluation on test data (256 items):

      Before Pruning      After Pruning
      -----
Size      Errors  Size      Errors  Estimate
      43  64(25.0%)  39  63(24.6%)  (21.6%)  <<
    
```

Fold 3:

Evaluation on training data (512 items):					
Before Pruning			After Pruning		
Size	Errors		Size	Errors	Estimate
29	110(21.5%)		23	112(21.9%)	(26.4%) <<
Evaluation on test data (256 items):					
Before Pruning			After Pruning		
Size	Errors		Size	Errors	Estimate
29	66(25.8%)		23	64(25.0%)	(26.4%) <<

Infine, la Tabella 5 riassume i risultati statistici dell'analisi effettuata.

Fold	C4.5 before pruning	C4.5 after pruning
1	68,4 %	68,8 %
2	75,0 %	75,4 %
3	74,2 %	75,0 %
<b>Media</b>	72,53 %	73,07 %
<b>Deviaz. Std.</b>	3,02	2,94

Tabella 5: accuratezza alberi di decisione C4.5

## 9 Conclusioni

Mettendo a confronto i risultati di accuratezza dell'analisi mediante reti neurali, con quelli forniti dagli alberi di decisione, si nota, come è possibile osservare in Tabella 6, un miglior comportamento delle prime che, oltre ad avere un'accuratezza media superiore, hanno anche una dispersione dei dati notevolmente minore, il che suggerisce una migliore capacità delle reti neurali ad adattarsi a variazioni sui dati di ingresso.

	<b>Media</b>	<b>Deviaz. Std.</b>
<b>JavaNNS</b>	75,03 %	0,67
<b>C4.5 b.p.</b>	72,53 %	3,02
<b>C4.5 a.p.</b>	73,07 %	2,94

Tabella 6: confronto tra i risultati di JavaNNS e C4.5

D'altro canto, l'analisi con C4.5 ha l'innegabile vantaggio di essere facilmente automatizzabile e di fornire risultati attendibili in tempi molto più brevi e senza l'analisi preliminare richiesta dalla progettazione di una rete neurale vera e propria.

## 10 Appendice

### 10.1 Randomizer

```
#!/usr/bin/ruby
#
# randomizza le linee all'interno di un file
fileclass1 = File.new(ARGV[0], 'r')
fileclass1_random = File.new(ARGV[0] + 'rand', 'w')
file1_hash = []
fileclass1.each do |line|
  file1_hash << line
end
i = file1_hash.size
i.downto(1) do |x|
  index = rand(x - 1)
  tmp = file1_hash[index]
  file1_hash[index] = file1_hash.last
  file1_hash[file1_hash.size - 1] = tmp
  fileclass1_random.puts(file1_hash.pop)
end
fileclass1.close
```

### 10.2 Result parser

```
#!/usr/bin/ruby
#
# effettua il parsing di un set di risultati di JavaNNS.
class Parser
  def initialize(file)
    @file = file
    @results = { }
```

```
@sorted = nil
end
def order
  if @sorted.nil?
    @sorted = @results.sort { |a, b| a[1] <=> b[1] }.reverse!
  end
end
def best
  self.order
  puts '=====  
Migliore risultato: ====='  
puts 'File: ' + @sorted.first[0].to_s  
puts 'Accuratezza: ' + @sorted.first[1].to_s  
puts
end

def halloffame
  self.order
  puts '=====  
Hall of Fame: ====='  
puts
  1.upto(9) do |i|
    puts "#{(i + 1)}o miglior risultato:"  
puts 'File: ' + @sorted[i][0].to_s  
puts 'Risultato: ' + @sorted[i][1].to_s  
puts
  end
end

def parse
  if File.directory?(@file.path)
    files = Dir.entries(@file.path)
    files.each do |path|
      unless File.directory?(@file.path + "/" + path)
        puts '=====  
File: ' + path + ' ====='  
puts  
parseFile(File.new(@file.path + "/" + path))  
puts
      end
    end
  else
    parseFile(@file)
  end
end

def parseName (basename)
  strv = basename.split('_')
  case strv[1]
  when 'backp'
    parseBackp(strv)
  end
end
```

```
when 'backp-mom'
  parseBackpMomentum(strv)
when 'batch-backp'
  parseBackp(strv)
end
end

def parseBackp (elements)
  puts 'n: ' + elements[2].to_s.insert(1, '.')
  puts 'dmax: ' + elements[3].to_s.insert(1, '.')
  puts 'neuroni: ' + elements[4].to_s
  puts
end

def parseBackpMomentum (elements)
  puts 'n: ' + elements[2].to_s.insert(1, '.')
  puts 'u: ' + elements[3].to_s.insert(1, '.')
  puts 'c: ' + elements[4].to_s.insert(1, '.')
  puts 'dmax: ' + elements[5].to_s.insert(1, '.')
  puts 'neuroni: ' + elements[6].to_s
  puts
end

def parseFile (file)
  line0=false
  line1=false
  val = nil
  response = 0
  right_vals = 0
  wrong_vals = 0
  parseName(File.basename(file.path, '.res'))
  file.each do |line|
    if line1 == false and line0 == false and
      line[0,1] == '#'
      line0 = true
      next
    end
    if line0 == true
      val = line
      line0 = false
      line1 = true
      next
    end
    if line1 == true
      if line.to_f > 0.5
        response = 1
      else
        response = 0
      end
    end
  end
end
```

```

        end

        if response == val.to_i
            right_vals += 1
        else
            wrong_vals += 1
        end
        line1 = false
    end
end

val = ((right_vals.to_f / (right_vals.to_f +
    wrong_vals.to_f)) * 100)

puts 'Giuste ' + right_vals.to_s
puts 'Sbagliate ' + wrong_vals.to_s
puts 'Totale ' + (right_vals + wrong_vals).to_s
puts 'Accuratezza ' + val.to_s

@results[File.basename(file.path)] = val.to_f
end
end

file = File.new(ARGV[0], 'r')
parser = Parser.new(file)
parser.parse
parser.best
parser.halloffame

```

### 10.3 Normalizer

```

#!/usr/bin/ruby
#
# normalizza i dati per l'importazione in JavaNNS

file = File.new('diabetes.all', 'r')
max = Array.new(8)
min = Array.new(8)
file.each do |line|
    entry = line.gsub(' ', '').split(',')
    entry.pop
    i = 0
    entry.each do |c|
        max[i] = c.to_f if max[i].nil? or c.to_f > max[i]
        min[i] = c.to_f if min[i].nil? or c.to_f < min[i]
        i += 1
    end
end

file.rewind
file2write = File.new('parsed.txt', 'w')

```

```
file.each do |line|
  entry = line.gsub(' ', '').split(',')
  n = Array.new(9)
  n[8] = entry.pop
  i = 0

  entry.each do |c|
    n[i] = (c.to_f-min[i])/(max[i]-min[i])
    i += 1
  end

  file2write.puts(n.join(", "))
end
file2write.close
```