Data Mining and Machine Learning Lab. Lezione 4 Master in Data Science for Economics, Business and Finance 2018

20.04.18

Marco Frasca

Università degli Studi di Milano

Efficienza dei loop in R

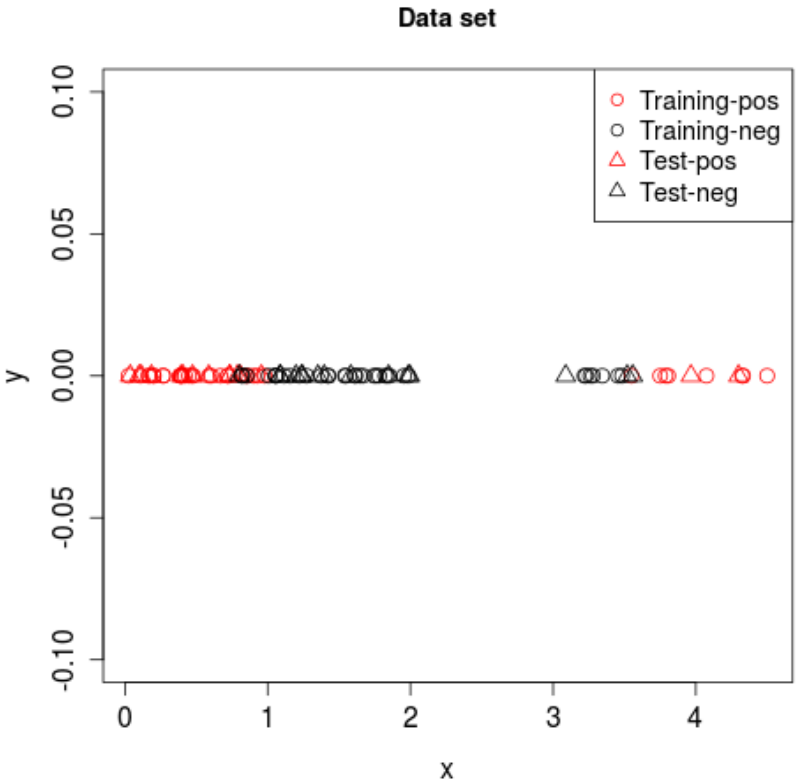
Compariamo le due soluzioni `d.Eucl.mat` e `d.Eucl.mat2` dell'esercizio 2 della lezione scorsa. La prima soluzione implementa il ciclo esterno mediante istruzione `for`, quello interno mediante la funzione `apply`. La seconda funzione implementa il doppio ciclo richiamando codice precompilato scritto in linguaggio `C`, come si può verificare digitando `dist` dal terminale **R**.

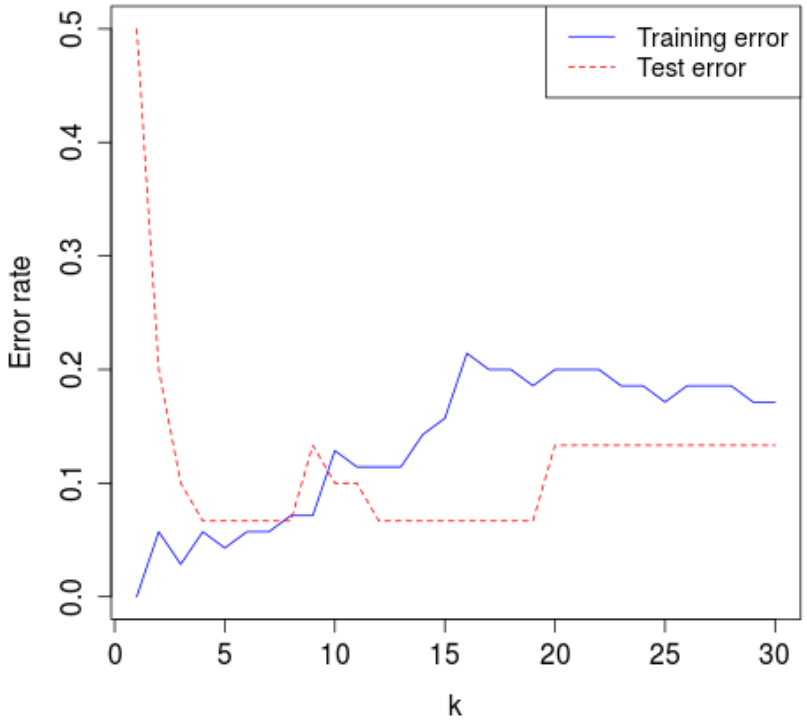
```
source("Sol_Es_Lez3.R")
a<-matrix(runif(100000), nrow=500); Init <- proc.time()
D <- d.Eucl.mat2(a); End <- proc.time();
time <- (End[3]-Init[3]);  print(time)
elapsed
  0.209
Init <- proc.time();  D <- d.Eucl.mat(a);  End <- proc.time();
time <- (End[3]-Init[3]);  print(time)
elapsed
  0.773
a<-matrix(runif(100000), nrow=1000); Init <- proc.time()
D <- d.Eucl.mat2(a); End <- proc.time();
time <- (End[3]-Init[3]);  print(time)
elapsed
    0.376
Init <- proc.time();  D <- d.Eucl.mat(a);  End <- proc.time();
time <- (End[3]-Init[3]);  print(time)
elapsed   1.886
```

Test del classificatore kNN su data set artificiale

Generiamo un campione casuale a una dimensione ($d = 1$) che sia relativamente "facile" da classificare.

```
pos.points <- c(runif(40), runif(10, min=3.5, max=4.5))
neg.points <- c(runif(40, min=0.8, max=2), runif(10, min=3, max=3.6))
y<-c(rep(1, length(pos.points)), rep(-1, length(neg.points)))
M<- as.matrix(c(pos.points, neg.points), ncol=1);       n <- nrow(M)
train <- sample(1:n, ceiling(0.8*n));       test <- setdiff(1:n, train)
k.best <- train.kNN.test(M, y, train, test)  #k.best  5
```

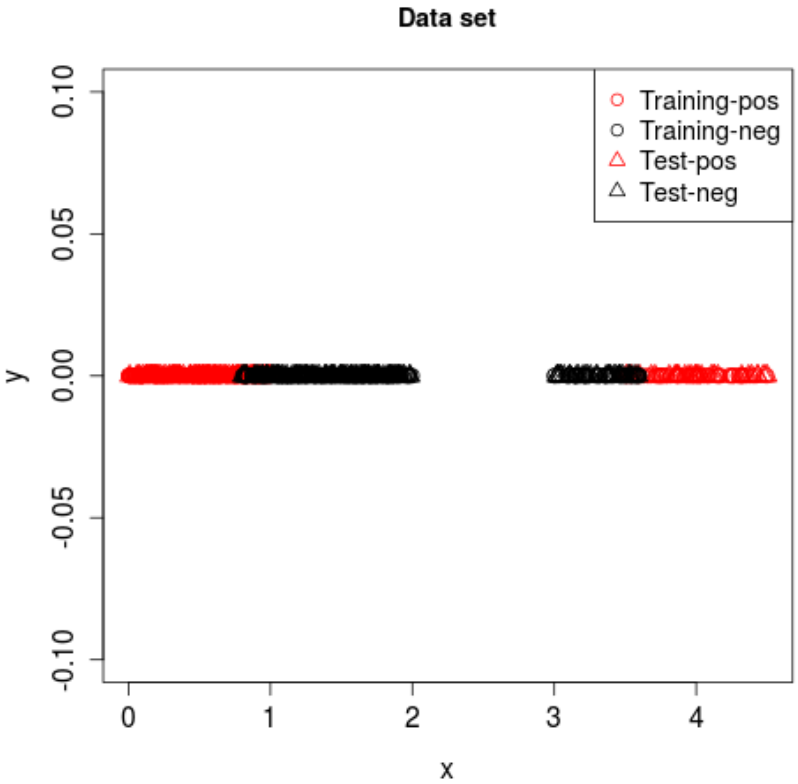


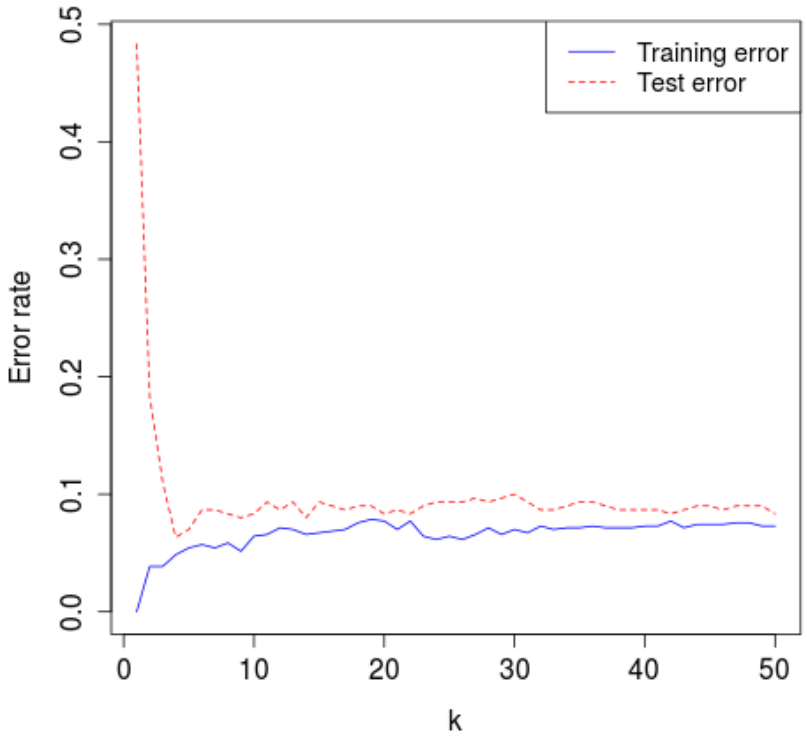


Test del classificatore kNN su data set artificiale - 2

Campione casuale a una dimensione ($d = 1$) e taglia maggiore ($n = 1000$). Notate l'andamento degli errori più 'regolare'

```
pos.points <- c(runif(400), runif(100, min=3.5, max=4.5))
neg.points <- c(runif(400, min=0.8, max=2), runif(100, min=3, max=3.6))
y<-c(rep(1, length(pos.points)), rep(-1, length(neg.points)))
M<- as.matrix(c(pos.points, neg.points), ncol=1);      n <- nrow(M)
train <- sample(1:n, ceiling(0.8*n));      test <- setdiff(1:n, train)
k.best <- train.kNN.test(M, y, train, test)  #k.best  4
```

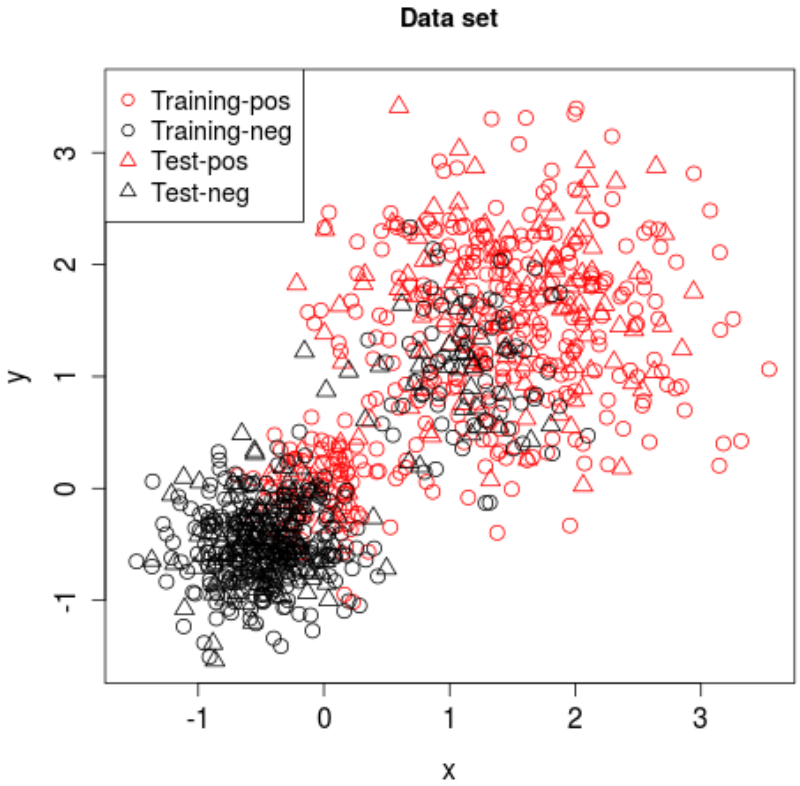


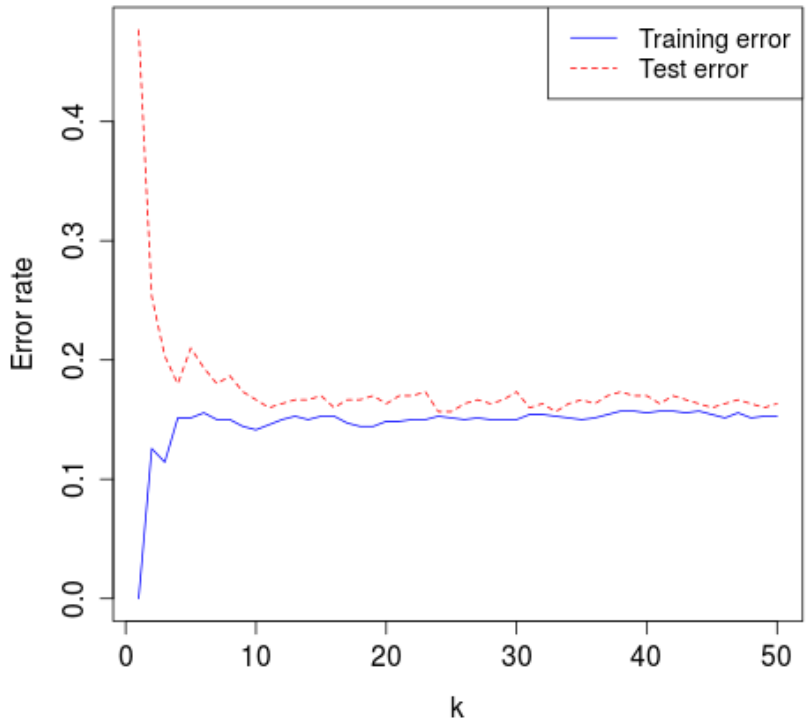


Test del classificatore kNN su data set artificiale - 3

Generiamo un campione casuale a due dimensioni ($d = 2$) e taglia $n = 1000$.

```
pos.points.x <- c(rnorm(400, mean=1.5, sd=0.7), rnorm(100, mean=0, sd=0.3))
pos.points.y <- c(rnorm(400, mean=1.5, sd=0.7), rnorm(100, mean=0, sd=0.3))
neg.points.x <- c(rnorm(400, mean=-0.5, sd=0.35), rnorm(100, mean=1, sd=0.45))
neg.points.y <- c(rnorm(400, mean=-0.5, sd=0.35), rnorm(100, mean=1, sd=0.45))
y<-c(rep(1, length(pos.points.x)), rep(-1, length(neg.points.x)))
M<- cbind(c(pos.points.x, neg.points.x), c(pos.points.y, neg.points.y));  n <- nrow(M)
train <- sample(1:n, ceiling(0.8*n));      test <- setdiff(1:n, train)
k.best <- train.kNN.test(M, y, train, test)  #k.best  25
```



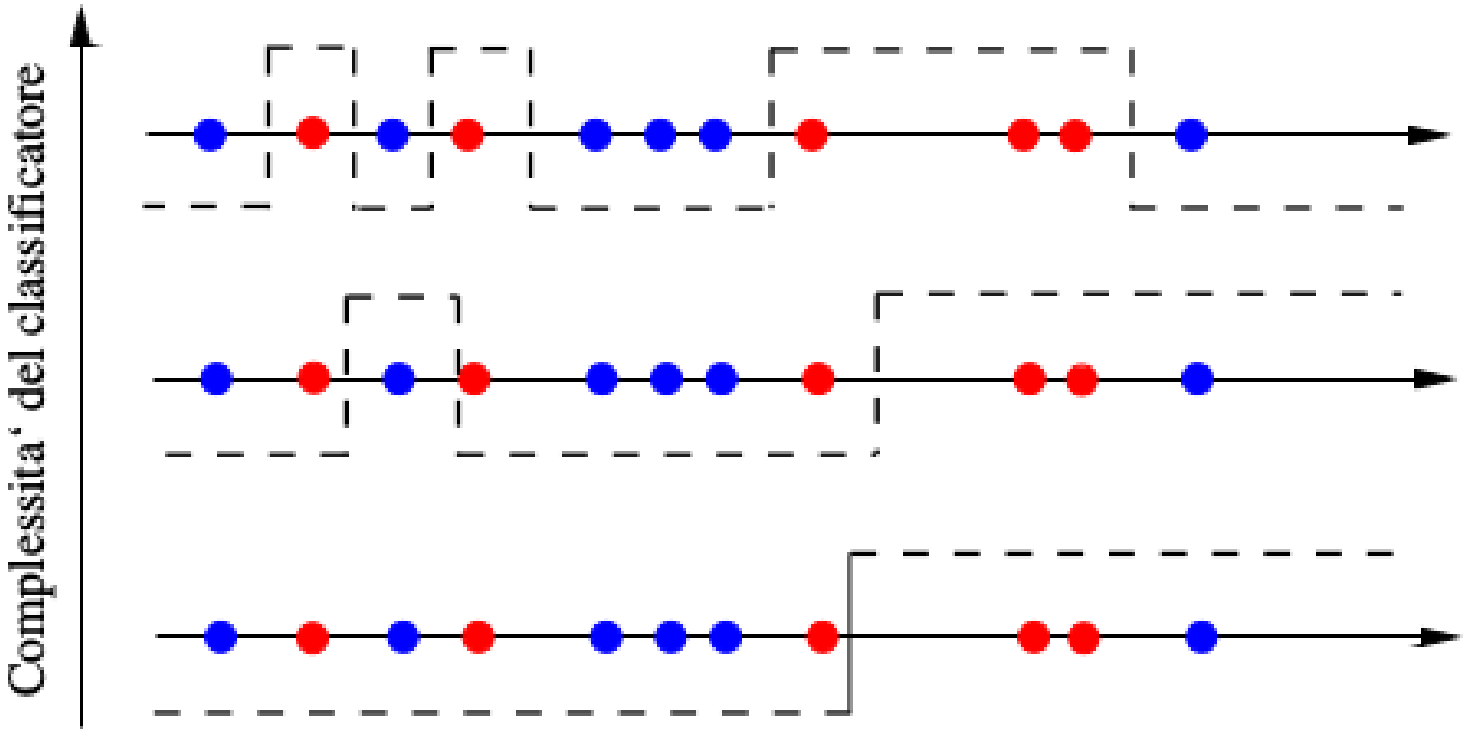


Complessità del classificatore kNN

La complessità della funzione f_{kNN} cresce o descresce con k?

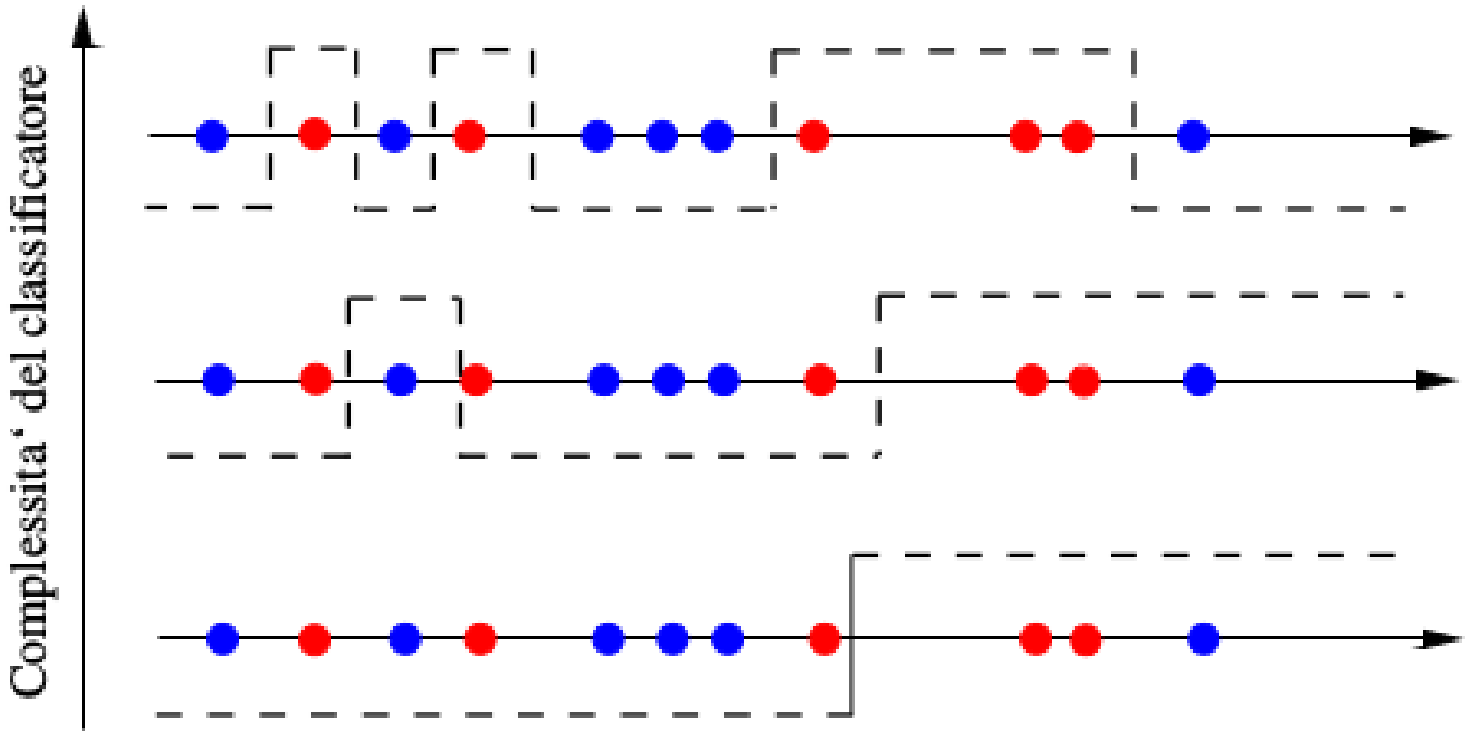
Complessità del classificatore kNN

Forma della funzione f_{kNN} per i valori $k = 1$ (alto), 3, 5 (basso) su un training set unidimensionale (*rosso* negativo, *blue* positivo).



Complessità del classificatore kNN

Forma della funzione f_{kNN} per i valori $k = 1$ (alto), 3, 5 (basso) su un training set unidimensionale.



Quando $k = n$, kNN diventa il classificatore costante che predice ogni punto $\boldsymbol{x}$ con la maggioranza di tutte le etichette del training set

Esercizio

Ripetere i tre esperimenti nelle slide precedenti aggiungendo, per ciascun data set, anche l'apprendimento di k mediante la funzione `train.kNN.CV(M,y,nfolds)`, provando per esempio $nfolds = 5$. Confrontare le stime del parametro k di kNN ottenute mediante i due metodi (metodo del test e cross validation) sugli stessi dati artificiali.

Classificatori ad albero: CART (Classification and Regression Trees)

- Possiamo realizzare alberi di decisione attraverso il package **R** *rpart*
- La funzione `rpart` permette di apprendere un albero di decisione (ma anche di regressione)

```
rpart(formula, data, weights, subset, na.action = na.rpart, method,
      model = FALSE, x = FALSE, y = TRUE, parms, control, cost, ...)
```

- Diversi argomenti, i principali sono: `data`, data.frame contenente le osservazioni; `formula`, oggetto di classe 'formula' che descrive i predittori e la variabile dipendente; `method`, per indicare il tipo di albero (decisione o regressione)
- I puntini sospensivi `...` indicano argomenti opzionali. Es. quelli per la funzione `rpart.control`, tra cui: `minsplit` (def. 20), minimo numero di istanze che devono esistere in un nodo affinché ne si possa tentare lo split; `cp` (Complexity Parameter, def. 0.01), parametro che determina gli split 'significativi', quelli in cui la complessità dell'albero è ancora maggiore di *cp*. Riduce il tempo di computazione tagliando gli split che ridurrebbero poco l'errore; `xval`, numero di fold nella crossa validazione interna; `maxdepth` (def. 30) massima profondità di una foglia (profondità radice 0)

Package rpart: data set *Kyphosis*

- Il package *rpart* fornisce anche data set di prova *khyphosis*, che raccoglie informazioni sulla presenza o meno di deformazione (*Kyphosis*) dopo un operazione alla colonna (cifosi, o comunemente detta 'gobba'), e su altri dati del paziente, come età in mesi (*Age*), numero di vertebre coinvolte (*Number*) e la più alta vertebra operata (*Start*)
- Contiene 81 osservazioni, di cui 64 senza e 17 con deformazione

```
library(rpart)
data(kyphosis) # carichiamo il data set
attr <- colnames(kyphosis);   print(attr)
  [1] "Kyphosis" "Age"      "Number"   "Start"
dim(kyphosis)
  [1] 81  4
str(kyphosis)
'data.frame': 81 obs. of  4 variables:
 $ Kyphosis: Factor w/ 2 levels "absent","present": 1 1 2 1 1 1 1 1 1 2 ..
 $ Age     : int  71 158 128 2 1 1 61 37 113 59 ...
 $ Number  : int  3 3 4 5 4 2 2 3 2 6 ...
 $ Start   : int  5 14 5 1 15 16 17 16 16 12 ...
table(kyphosis$Kyphosis)
 absent present
     64      17
```

Funzione `rpart`

```
library(rpart)
data(kyphosis)
fit <- rpart(Kyphosis ~ Age + Number + Start,
   method="class", data=kyphosis)
```

- Il primo argomento è la formula per definire la variable dipendente e come essa dipende dai predittori
- 'class' è il metodo per costruire un albero di decisione (classificazione)
- `data` invece receve il data.frame che contiene esattamente le colonne indicate nel primo argomento

Vediamo un modo alternativo di impostare la formula, anche in casi in cui i predittori siano tanti, dove quindi risulterebbe impossibile elencarli tutti

```
 library(rpart); data(kyphosis)
predictors <- colnames(kyphosis);
outcome.col <- which(predictors == "Kyphosis")
fmla <- as.formula(paste("Kyphosis ~ ", paste(predictors[-outcome.col],
collapse= "+")));
print(fmla)
   Kyphosis ~ Age + Number + Start
```

Funzione `rpart` - 2

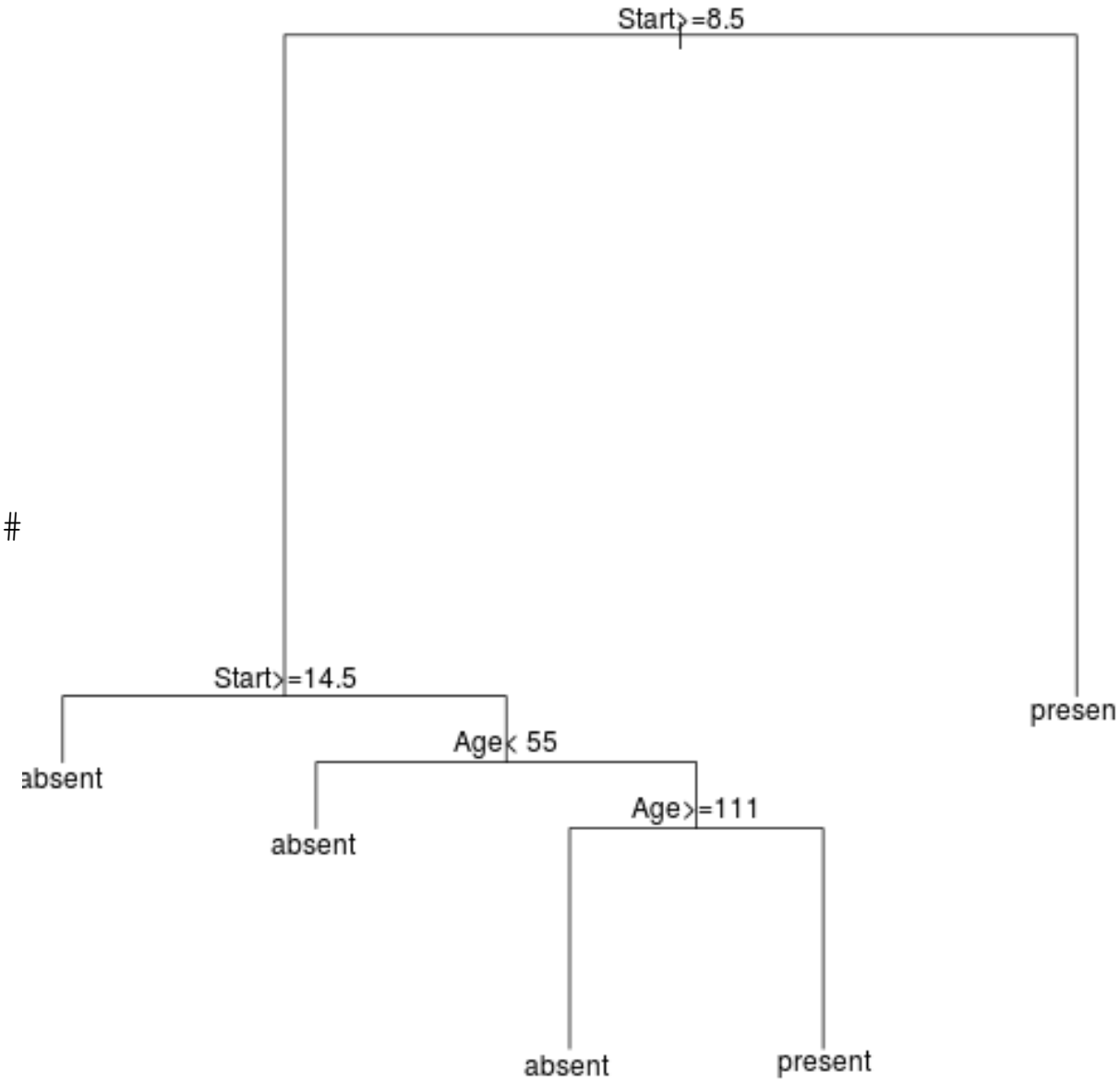
```
library(rpart); data(kyphosis)
fit <- rpart(Kyphosis ~ Age + Number + Start, method="class",
        data=kyphosis) # oppure
fit <- rpart(fmla,  method="class", data=kyphosis)
print(fit)
n= 81

node), split, n, loss, yval, (yprob)
      * denotes terminal node

 1) root 81 17 absent (0.79012346 0.20987654)
   2) Start>=8.5 62  6 absent (0.90322581 0.09677419)
     4) Start>=14.5 29  0 absent (1.00000000 0.00000000) *
     5) Start< 14.5 33  6 absent (0.81818182 0.18181818)
      10) Age< 55 12  0 absent (1.00000000 0.00000000) *
      11) Age>=55 21  6 absent (0.71428571 0.28571429)
        22) Age>=111 14  2 absent (0.85714286 0.14285714) *
        23) Age< 111 7  3 present (0.42857143 0.57142857) *
   3) Start< 8.5 19  8 present (0.42105263 0.57894737) *
```

Grafichiamo l'albero

```
par(mar = rep(0.1, 4))#
plot(fit)
text(fit)
```



Funzione `rpart` - Funzione di errore (impurità)

- Di default `rpart` utilizza la funzione di *Gini* per definire il nodo da espandere, corrispondente alla funzione $f(p) = p(1-p)$
- si può anche scegliere come indice 'information', corrispondente alla funzione $f(p) = -plog(p)$

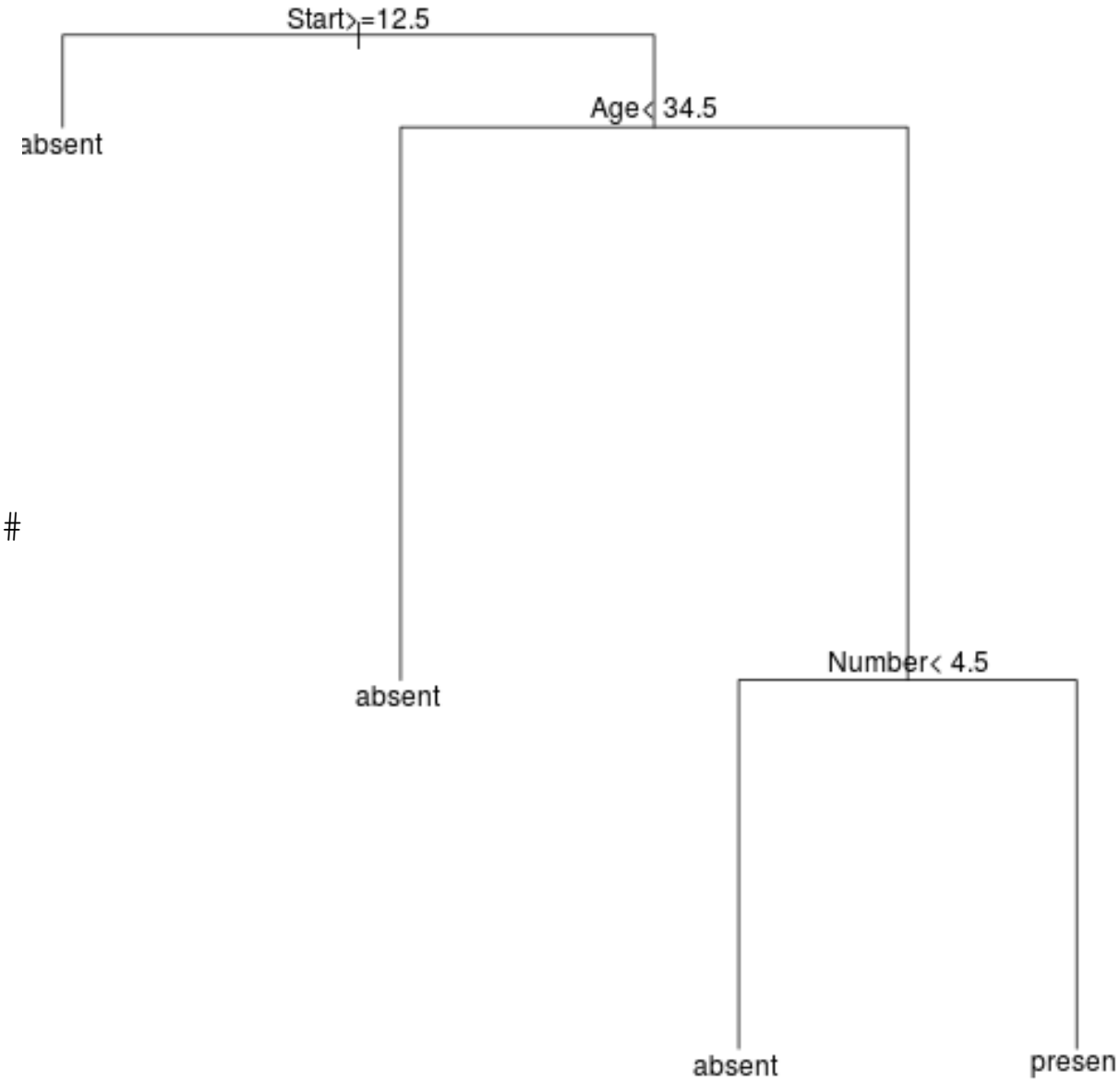
```
library(rpart); data(kyphosis)
fitI <- rpart(fmla, data=kyphosis, parms = list(split = "information"))
print(fitI)
n= 81

node), split, n, loss, yval, (yprob)
      * denotes terminal node

 1) root 81 17 absent (0.79012346 0.20987654)
   2) Start>=12.5 46  2 absent (0.95652174 0.04347826) *
   3) Start< 12.5 35 15 absent (0.57142857 0.42857143)
     6) Age< 34.5 10  1 absent (0.90000000 0.10000000) *
     7) Age>=34.5 25 11 present (0.44000000 0.56000000)
      14) Number< 4.5 12  5 absent (0.58333333 0.41666667) *
      15) Number>=4.5 13  4 present (0.30769231 0.69230769) *
```

Grafichiamo in nuovo albero

```
par(mar = rep(0.1, 4))#
plot(fitI)
text(fitI)
```



Valutiano gli errori: funzione `printcp`

```
printcp(fit)
Classification tree:
Variables actually used in tree construction:
 [1] Age   Start
Root node error: 17/81 = 0.20988
n= 81
        CP nsplit rel error xerror    xstd
1 0.176471      0   1.00000 1.0000 0.21559
2 0.019608      1   0.82353 1.1765 0.22829
3 0.010000      4   0.76471 1.1765 0.22829

printcp(fit)
Variables actually used in tree construction:
[1] Age    Number Start
Root node error: 17/81 = 0.20988
n= 81
        CP nsplit rel error xerror    xstd
1 0.088235      0   1.00000 1.0000 0.21559
2 0.010000      3   0.70588 1.2353 0.23200
```

- *CP* il indica la complessità dell'albero. Quando diventa minore di 0.01 (default) l'algoritmo si arresta
- *CP* è l'errore commesso dall'albero attuale + il suo numero di nodi moltiplicato per l'errore commesso dalla radice. Serve a penalizzare alberi con troppi nodi e basse riduzioni di errore
- Le colonne errore vengono scalate in modo da avere `rel error` 1 nella radice (che altrimenti sarebbe 0.20988);
- `xerror` è l'errore di cross validazione (sul test), `error` è l'errore sul training.
- Queste infomazioni sono in genere utili per decidere dove fare pruning per evitare overfitting

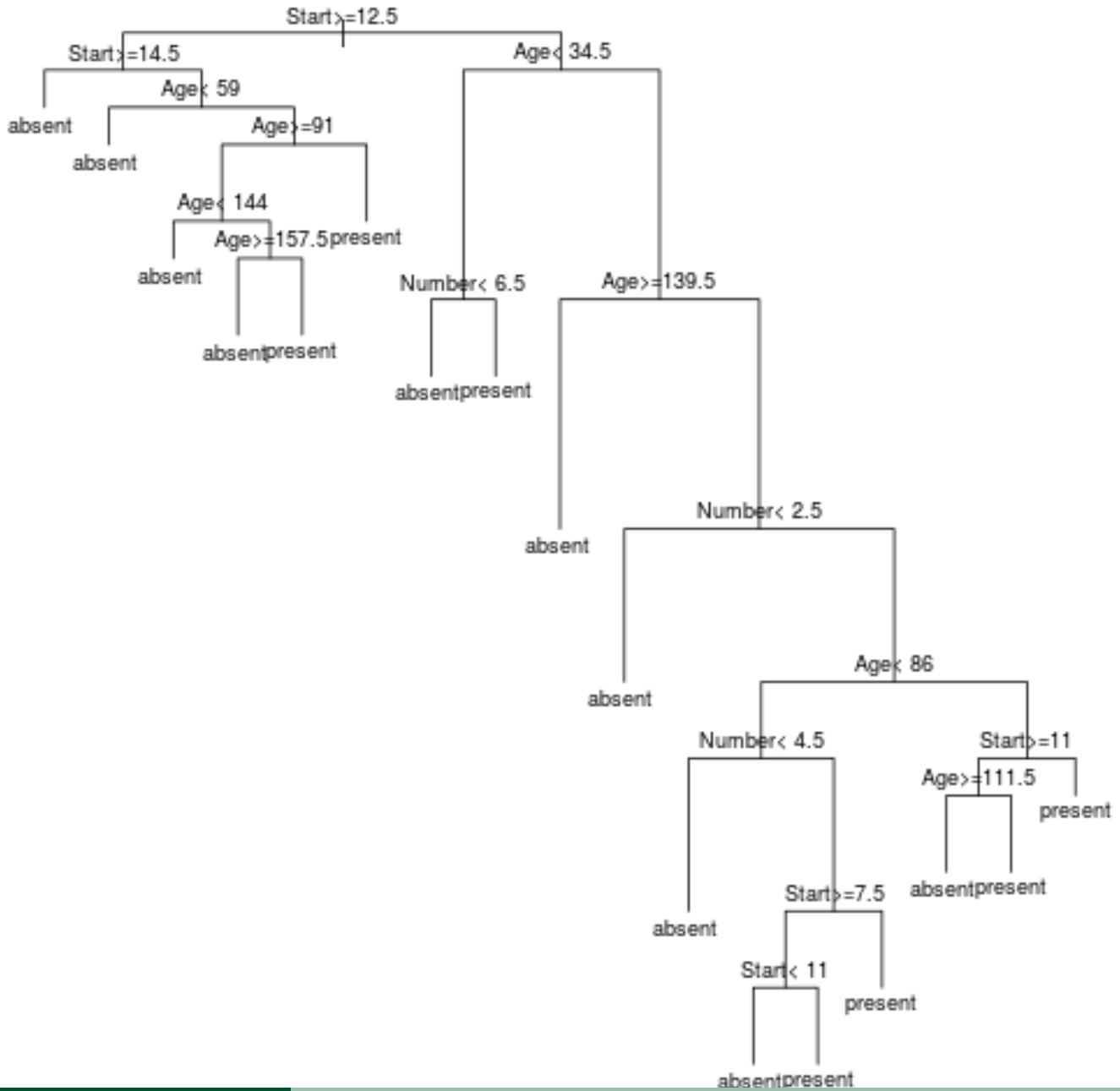
Gestire la complessità dell'albero

Riducendo il numero di osservazioni minimo per nodo per poter avere uno split (argomento `minslipt`, default 20) possiamo aumentare la complessità del classificatore fino eventualmente ad avere training error 0

```
fit2 <- rpart(fmla, data=kyphosis, parms = list(split = "information"),
              control = rpart.control(minsplit = 2) )
printcp(fit2)

Classification tree:
rpart(formula = fmla, data = kyphosis, parms = list(split = "information")
    control = rpart.control(minsplit = 1))
Variables actually used in tree construction:
[1] Age    Number Start
Root node error: 17/81 = 0.20988
n= 81
        CP nsplit rel error  xerror    xstd
1 0.117647      0   1.00000 1.00000 0.21559
2 0.088235      4   0.52941 1.23529 0.23200
3 0.058824      6   0.35294 1.17647 0.22829
4 0.029412      9   0.17647 0.82353 0.20018
5 0.019608     11   0.11765 0.82353 0.20018
6 0.010000     16   0.00000 0.82353 0.20018
```

Albero corrispondente



Significatività dei predittori (variabili)

- L'oggetto restituito da `rpart` possiede un campo `varaible.importance` che è il vettore degli indici di informatività di ogni variabile indipendente
- Considera l'indice di errore di ogni split in cui la variabile è variabile primaria

```
print(fit$variable.importance)
   Start      Age   Number
8.198442 3.101801   1.521863

print(fitI$variable.importance)
   Start      Age   Number
9.813182 3.908452   3.142603

print(fit2$variable.importance)
     Age    Start   Number
19.73162 19.12002   11.28640
```

Notate per il primo albero la variabile *number* ha una rilevanza bassa, infatti essa è esclusa dall'albero

Predire nuove istanze

- L'albero di decisione restituito dalla funzione `rpart` può essere utilizzato per predire nuove istanze
- Dividiamo il data set in training e test set in maniera stratificata. Usiamo il 20% dei dati come test

```
n <- nrow(kyphosis)
pos <- which(kyphosis$Kyphosis=="present");   n.pos <- length(pos)
neg <- setdiff(1:n, pos);  n.neg <- length(neg)
test <- c(sample(pos, floor(0.2*n.pos)), sample(neg, floor(0.2*n.neg)))
train <- setdiff(1:n, test)
n.test <- length(test)
n.test
[1] 15
length(train)
[1] 66
length(intersect(train,test))
[1] 0
n.pos.t <- length(intersect(pos, test))
[1] 3
n.neg.t <- length(test)-n.pos.t
[1] 12
```

La funzione `predict`

Addestriamo l'albero sul training set, poi prediciamo il valore delle istanze del test

```
outcome.col <- which(predictors == "Kyphosis")
fit.tr <- rpart(fmla, method="class", data=kyphosis[train,])
test.data <- kyphosis[test, -outcome.col]
pred <- predict(fit.tr, newdata=test.data, type="class")
pred
     11      62      53      34      76      39      43      63      47
 absent present present  absent  absent  absent present present  absent  a
     79      28      72       1      19
 absent  absent present present  absent
Levels: absent present
```

L'output di `predict` per il tipo "class" è un 'fattore', tipo di dato con cui **R** rappresenta dati discreti. `newdata` è il data set da predire.

Errore assoluto ed errore relativo

Calcoliamo errore assoluto ed errore relativo (somma degli errori assoluti sulle singole classi)

```
abs.err <- sum(pred!=kyphosis[test,"Kyphosis"])/n.test
[1] 0.2

p.err<-sum(pred!="present"&kyphosis[test,"Kyphosis"]=="present")/n.pos.t

n.err<-sum(pred=="present"&kyphosis[test,"Kyphosis"]!="present")/n.neg.t

rel.err <- 0.5*p.err + 0.5*n.err
[1] 0.375
```

Valutiamo un modello più complesso sugli stessi dati

Cambiamo il parametro di controllo `minsplit` per addestrare un albero in cui un nodo può essere espanso quando contiene almeno 2 osservazioni

```
fit.tr2 <- rpart(fmla, method="class", data=kyphosis[train,],
control=rpart.control(minsplit = 2) )
pred <- predict(fit.tr2, newdata=test.data, type="class")
p.err<-sum(pred!="present"&kyphosis[test,"Kyphosis"]=="present")/n.pos.t
n.err<-sum(pred=="present"&kyphosis[test,"Kyphosis"]!="present")/n.neg.t
abs.err <- sum(pred!=kyphosis[test,"Kyphosis"])/n.test
print(abs.err)
[1] 0.2666667
rel.err <- 0.5*p.err + 0.5*n.err
print(rel.err)
[1] 0.4166667
```

C'è overfitting !

Valutiamo un modello più complesso sugli stessi dati - 2

```
print(fit.tr2)
n= 66
node), split, n, loss, yval, (yprob)
      * denotes terminal node
  1) root 66 14 absent (0.78787879 0.21212121)
    2) Start>=12.5 39  1 absent (0.97435897 0.02564103)
      4) Age< 154 31  0 absent (1.00000000 0.00000000) *
      5) Age>=154 8  1 absent (0.87500000 0.12500000)
       10) Age>=157.5 7  0 absent (1.00000000 0.00000000) *
       11) Age< 157.5 1  0 present (0.00000000 1.00000000) *
    3) Start< 12.5 27 13 absent (0.51851852 0.48148148)
      6) Age< 34.5 8  1 absent (0.87500000 0.12500000)
       12) Number< 6 7  0 absent (1.00000000 0.00000000) *
       13) Number>=6 1  0 present (0.00000000 1.00000000) *
      7) Age>=34.5 19  7 present (0.36842105 0.63157895)
       14) Age>=139.5 2  0 absent (1.00000000 0.00000000) *
       15) Age< 139.5 17  5 present (0.29411765 0.70588235)
         30) Number< 2.5 2  0 absent (1.00000000 0.00000000) *
         31) Number>=2.5 15  3 present (0.20000000 0.80000000)
           62) Age< 70.5 6  3 absent (0.50000000 0.50000000)
            124) Age>=60 2  0 absent (1.00000000 0.00000000) *
            125) Age< 60 4  1 present (0.25000000 0.75000000)
              250) Age< 51.5 2  1 absent (0.50000000 0.50000000)
                500) Age>=46.5 1  0 absent (1.00000000 0.00000000) *
                501) Age< 46.5 1  0 present (0.00000000 1.00000000) *
              251) Age>=51.5 2  0 present (0.00000000 1.00000000) *
           63) Age>=70.5 9  0 present (0.00000000 1.00000000) *
```

Pruning dell'albero per evitare overfitting

- La funzione `prune` permette di rimuovere alcuni nodi dell'albero
- Prende come primo algomento il modello di cui fare pruning, il secondo `cp` è il paramtero di complessità dove l'albero sarà troncato
- Possiamo capire che valore dare a tale argomento valutando l'output della funzione `printcp`

```
printcp(fit.tr.pr)
Variables actually used in tree construction:
[1] Age    Number Start

Root node error: 14/66 = 0.21212

n= 66
   CP nsplit rel error xerror    xstd
1 0.214286      0   1.00000 1.0000 0.23723
2 0.142857      1   0.78571 1.2143 0.25376
3 0.071429      3   0.50000 1.1429 0.24868
4 0.047619      5   0.35714 1.2143 0.25376
5 0.035714      8   0.21429 1.2857 0.25844
6 0.010000     14   0.00000 1.4286 0.26668
```

- Dal passo 3 al passo 4 comincia a verificarsi overfitting: il training error (*rel error*) scende e l'errore tramite cross validation (*xerror*) cresce.
- Una regola empirica, quando l'overfitting non emerge direttamente, è quella di scegliere il primo *cp* per cui *rel error* + *xstd* < *xerror*

Pruning dell'albero per evitare overfitting

Eseguiamo il pruning con $cp = 0.071429$

```
fit.tr2.pr <- prune(fit.tr2, cp=0.071429)
pred <- predict(fit.tr2.pr, newdata=test.data, type="class")
p.err<-sum(pred!="present"&kyphosis[test,"Kyphosis"]=="present")/n.pos.t
n.err<-sum(pred=="present"&kyphosis[test,"Kyphosis"]!="present")/n.neg.t
abs.err <- sum(pred!=kyphosis[test,"Kyphosis"])/n.test
print(abs.err)
[1] 0.2
rel.err <- 0.5*p.err + 0.5*n.err
print(rel.err)
[1] 0.3333333
```

- Si noti come entrambi gli errori siano sensibilmente diminuiti
- L'errore relativo è diminuito anche rispetto al primo modello appreso
- Tuttavia, questi risultati possono essere affetti dalla specificità del test set. Come visto, per avere una milgiore stima dell'errore occorre eseguire la cross validation

Esercizi

1. Implementare la funzione **R** `train.minsplit.DT.test(data,fmla train,test)`, dove: `data` è un named `data.frame` di dimensione $n \times m$, dove n sono le osservazioni, e m le colonne, contenute nell'argomento di tipo formula `fmla`, che esprime le dipendenze tra variabile dipendente e quelle indipendenti (predittori). `train` è un vettore di interi contenente le righe di `data` da considerare come training set, mentre `test` è un vettore di interi contenente le righe di `data` che costituiscono il test set. La funzione deve:
   - Stimare il valore ottimale del parametro di controllo `minsplit` dell'albero di decisione appreso dai dati di training. Per farlo, valutare tutti i valori di `minsplit` da 2 a $\lfloor n/2 \rfloor$.
   - Tracciare il grafico del training e del test error ottenuti.
2. Ripetere l'esercizio precedente ottimizzando l'albero di decisione non sul parametro `minsplit` ma sul parametro di controllo `maxdepth`. Testare valori di `maxdepth` da 1 a 10.

Esercizi

3. Implementare la funzione **R** `train.minslipt.DT.CV(data,fmla,nfolds)`, dove `data` dove: `data` è un named `data.frame` di dimensione $n \times m$, dove n sono le osservazioni, e m le colonne, contenute nell'argomento di tipo formula `fmla`, che esprime le dipendenze tra variabile dipendente e quelle indipendenti (predittori). La funzione deve stimare, mediante cross-validation con `nfolds` sottoinsiemi, il valore ottimale del parametro `minsplit` del classificatore ad albero applicato ai dati di training, con $minsplit \in \{2, 3, \ldots, \lfloor n/2 \rfloor\}$. Suggerimenti:
   - Utilizzare la funzione `partition.bin` per una CV stratificata
   - Usare la funzione `train.minsplit.DT.test(data,fmla train,test)` ad ogni iterazione della CV, sopprimendo la stampa dei grafici di errore
   - Restuire la media delle stime di `minsplit` ottenute su ciascun sottoinsieme
4. Ripetere l'esercizio precedente sul parametro `maxdepth`.

Esercizi

5. Scrivere la funzione **R** `DT.minsplit.eval(data,fmla,nfolds)` che restituisca l'errore medio ottenuto dal dall'albero di decisione sui dati `data` mediante una cross validation esterna con `nfolds` sottoinsiemi. Ad ogni iterazione della CV esterna, usare la funzione `train.minslipt.DT.CV` sul training set per ottenere il valore del parametro `minsplit` (quindi mediante CV interna) da usare per predire le istanze del test set.