ASML: Esempio Laboratorio 8

Analisi del fenomeno di churn per una banca

N. Torelli

2024

Contents

Il problema e i dati

Il problema e i dati sono descritti in Kaggle https://www.kaggle.com/datasets/shrutimechlearn/churn-modelling

Si tratta di dati relativi ai clienti di una banca e la variabile di interesse è una variabile dicotomica che registra il fatto che il cliente abbia lasciato la banca (chiuso il suo conto) o continui a essere un cliente. L'obiettivo è quindi quello di costruire un algoritmo che consenta di prevedere se un attuale cliente è a rischio di lasciare la banca o meno. Tale informazione potrebbe essere utile a modulare campagne promozionali volte a trattenere gli attuali clienti che sono però classificati fra quelli che lascerebbero l'istituto.

Si tratta quindi di costruire un modello di classificazione.

Per svolgere l'esercizio sarà opportuno installare alcuni packages che torneranno utili (se non già installati in precedenza). Si può procedere con l'interfaccia utente di Rstudio (cliccando, ad esempio, su install nel menù package nel pannello in basso a destra) oppure con il comando `install.package()`

Lettura e preparazione dei dati

Dalla pagina Kaggle sopra citata sarà possibile scaricare un file compresso, una volta decompresso avremo un file con estensione .csv che è agevole leggere con `R` (ricordare di mettere il file nella directory di lavoro "working directory")

```
# i dati sono letti e caricati nel dataframe chiamato churn

churn1<-read.csv("Churn_Modelling.csv", header=T)

# Il file contiene 14 variabili (per la descrizione delle
# variabili si veda la pagina kaggle
# Le prime 3 variabili sono di scarso interesse per l'analisi
```

```
# selezioniamo solo le variabili di interesse
churn<-churn1[,4:14]

# conviene ora definire le variabili che sono fattori
# qualitatitivi e assegnare i livelli ad alcune

churn$Geography<-factor(churn$Geography)
churn$Gender<-factor(churn$Gender)
churn$HasCrCard<-factor(churn$HasCrCard)
churn$IsActiveMember<-factor(churn$IsActiveMember)
churn$Exited<-factor(churn$Exited)
levels(churn$Exited)<-c("no","yes") # Questa è la variabile target
levels(churn$IsActiveMember)<-c("no","yes")
levels(churn$HasCrCard)<-c("no","yes")
```

Creare la partizione (con caret)

Abbiamo visto che vi sono diversi modi per ottenere una partizione del file di dati in sottoinsiemi, scelti a caso, per i diversi scopi: allenare o valutare l'algoritmo. Il package `caret` è molto ampio e contiene diverse funzioni che possono essere usate per costruire regole di previsione o classificazione, per valutarle, o anche per preparare i dati (si veda, per approfondimento, https://topepo.github.io/caret/).

Ad esempio, contiene una funzione `CreateDataPartition` molto generale per generare partizioni.

```
library(caret)
set.seed(1234) # per ottenere sempre i medesimi insiemi di training e di test
trainID<-createDataPartition(churn$Exited,p=0.75, list=F) # p è la proporzione
                                                          # di dati nel training set
testID<-setdiff(1:length(churn$Exited), trainID)
train<-churn[trainID,]
test<-churn[testID,]
```

Creare un albero di classificazione (con rpart)

Il pacchetto `rpart` è uno dei principali strumenti che consentono di costruire un albero di regressione o di classificazione. Il suo funzionamento base è intuitivo tuttavia il pacchetto contiene numerose funzioni per le quali si rinvia a https://cran.r-project.org/web/packages/rpart/vignettes/longintro.pdf.

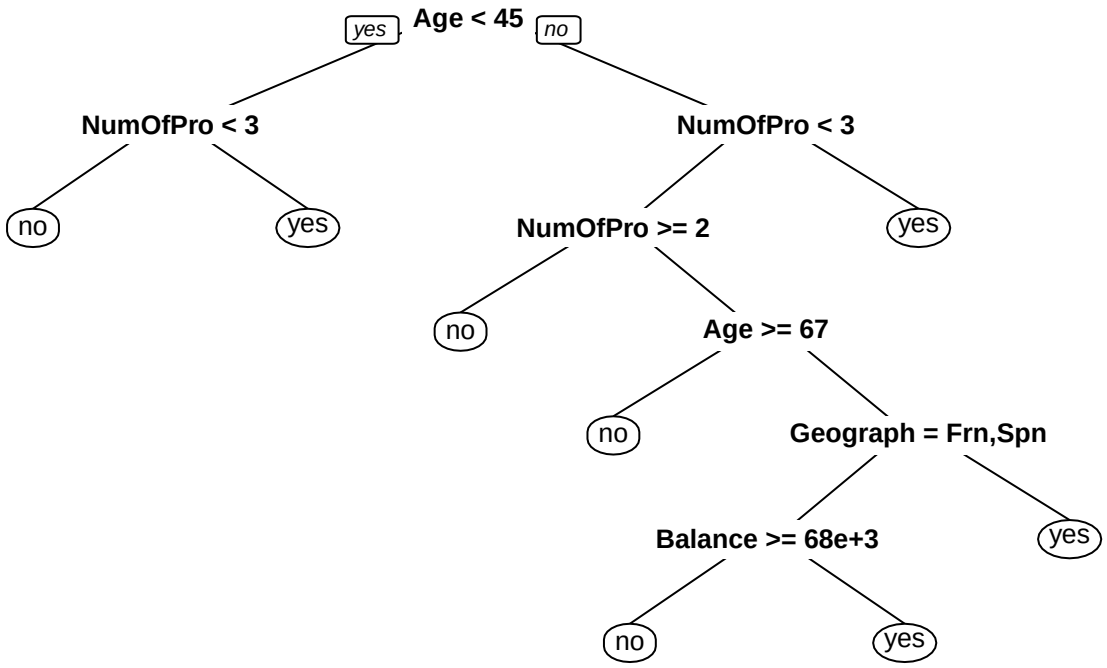
Proviamo a creare un albero (usando i default fissati nel pacchetto) e a disegnarlo (servirà il pacchetto `rpart.plot`).

```
library(rpart)
alb<-rpart(Exited~Geography+Gender+
             HasCrCard+Age+Tenure+Balance+NumOfProducts+
             EstimatedSalary+CreditScore, data=train, method="class")
# la variabile risposta è Exited e abbiamo inserito tutte le variabili come potenziali
# predittori. Si noti l'indicazione method=class che indica che si tratta di un albero di
# classificazione (a rigore sarebbe non necessaria avendo definite Exited come fattore)

alb # ci da la struttura dei nodi dell'albero
```

```
## n= 7501
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
```

```
##   1) root 7501 1528 no (0.79629383 0.20370617)
##     2) Age< 44.5 5756  746 no (0.87039611 0.12960389)
##       4) NumOfProducts< 2.5 5624  643 no (0.88566856 0.11433144) *
##       5) NumOfProducts>=2.5 132   29 yes (0.21969697 0.78030303) *
##     3) Age>=44.5 1745  782 no (0.55186246 0.44813754)
##       6) NumOfProducts< 2.5 1631  671 no (0.58859595 0.41140405)
##        12) NumOfProducts>=1.5 615  124 no (0.79837398 0.20162602) *
##        13) NumOfProducts< 1.5 1016  469 yes (0.46161417 0.53838583)
##          26) Age>=66.5 82   11 no (0.86585366 0.13414634) *
##          27) Age< 66.5 934  398 yes (0.42612420 0.57387580)
##            54) Geography=France,Spain 633  311 no (0.50868878 0.49131122)
##             108) Balance>=67948.14 404  153 no (0.62128713 0.37871287) *
##             109) Balance< 67948.14 229   71 yes (0.31004367 0.68995633) *
##            55) Geography=Germany 301   76 yes (0.25249169 0.74750831) *
##       7) NumOfProducts>=2.5 114    3 yes (0.02631579 0.97368421) *
```

```
# i successivi comandi disegnano l'albero (installare dapprima il package rpart.plot)
library(rpart.plot)
prp(alb)
```



Albero di classificazione con gestione della complessità (in rpart)

Abbiamo visto come possa convenire a volte far crescere l'albero in modo estremo (foglie con pochissimi casi), e quindi sovra adattato, per poi potarlo dei rami che comportano poco vantaggio. Possiamo decidere quanto debba esser profondo l'albero (depth), o quante suddivisioni vogliamo (split), o qual'è il numero di dati in una foglia al disotto del quale non si suddivide più (bucket). Oppure possimo decidere che uno split si fa se vi è un guadagno sufficiente in termini di riduzione dell'impurità. Questo ultimo aspetto in rpart può essere gestito con il parametro `cp`. Se esso è piccolo allora viene fatto crescere un nuovo ramo anche se c'è pochissimo vantaggio. Se esso è grande l'albero cresce poco. Nell'albero appena visto il valore era posto pari a 0.01. Ecco alcuni parametri di controllo dell'albero e i valori usati come default nella prova precedente.

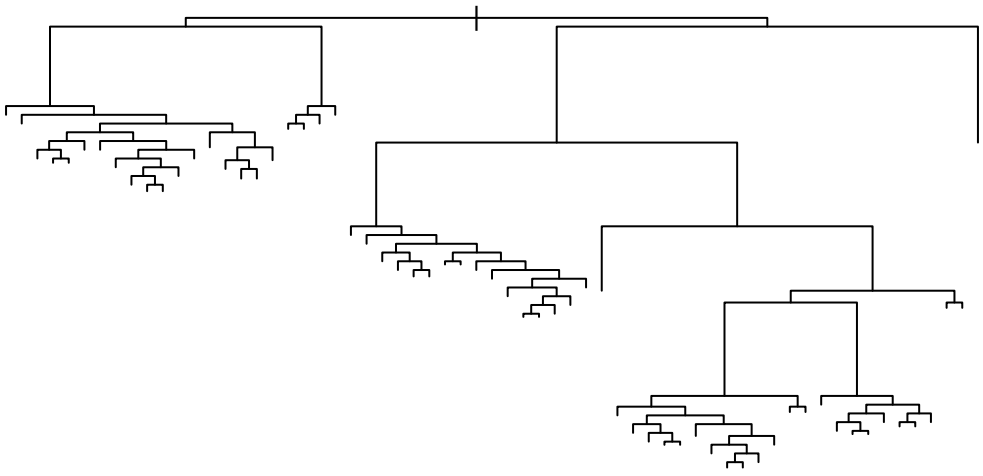
```
alb$control
```

```
## $minsplit
```

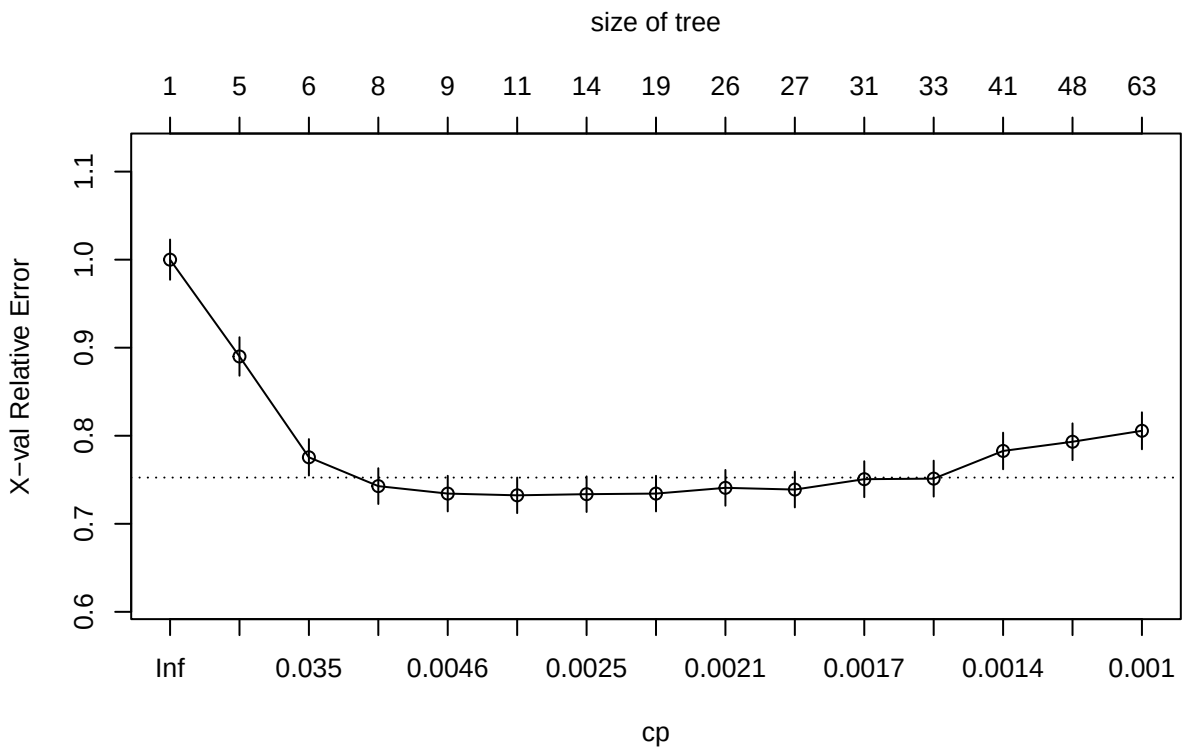
```
## [1] 20
##
## $minbucket
## [1] 7
##
## $cp
## [1] 0.01
##
## $maxcompete
## [1] 4
##
## $maxsurrogate
## [1] 5
##
## $usesurrogate
## [1] 2
##
## $surrogatestyle
## [1] 0
##
## $maxdepth
## [1] 30
##
## $xval
## [1] 10
```

Proviamo ora a costruire un albero piu complesso: useremo un valore di cp molto più basso. Questo farà crescere molto l'albero (basta un piccolissimo miglioramento per aprire dei nuovi rami, con cp più elevato l'abero sarà meno frondoso).

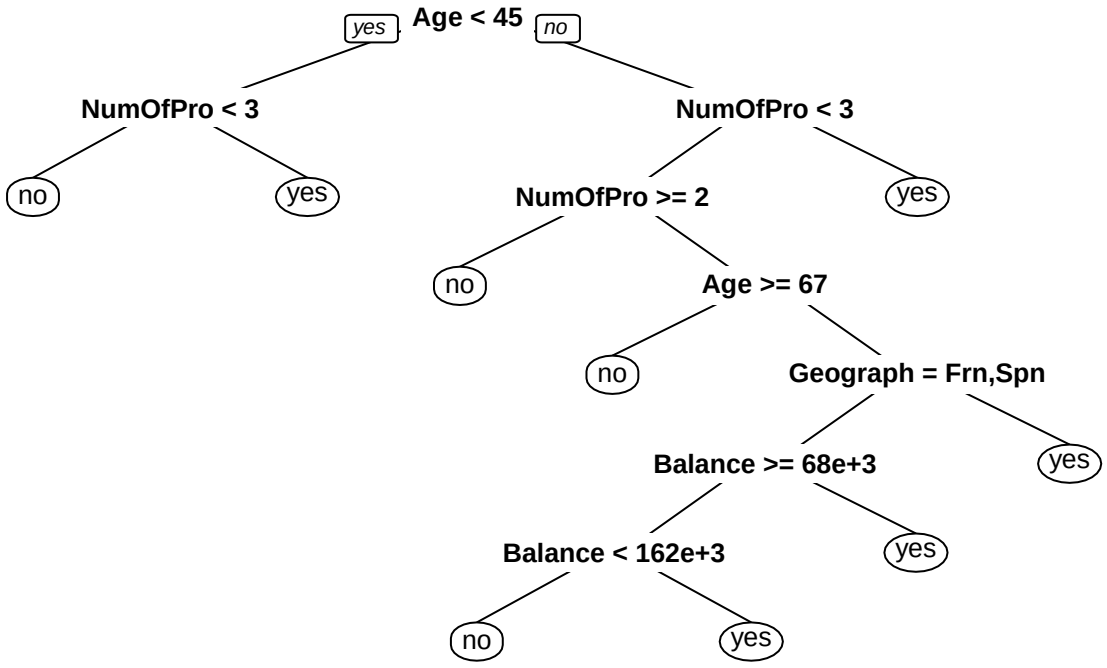
```
alb1<-rpart(Exited~Geography+Gender+HasCrCard+
              Age+Tenure+Balance+NumOfProducts+
              EstimatedSalary+CreditScore,data=train,
            method="class", control = rpart.control(cp = 0.001))
# alb1
plot(alb1)
```



```
# come si vede l'albero è ora molto più frondoso

plotcp(alb1)# Possiamo ottenere un grafico che dà i valori della
```




```
            # funzione obiettivo, che combina adattamento e complessità
            # ottenuti con cross-validation per diversi valori di cp.

# si può ora potare l'albero (pruning)  al valore di cp migliore che deriva
# osservando il grafico precedente: ovvero 0.0046

alb2<-prune(alb1, cp=0.0046)
prp(alb2)
```



Valutare le performance (dell'albero)

Per valutare le performance del classificatore, in questo caso l'albero, il principale strumento è la matrice di confusione. Per ottenerla sarà necessario il vettore delle classificazioni fornite dall'albero. Occorre usare la funzione `predict` applicata all'oggetto in cui abbiamo salvato l'albero, ad esempio `alb` per i dati che abbiamo conservato nel **test set**

```
# possiamo chiedere quindi in outupt il vettore delle classificazioni
# per i dati del test set

pred<-predict(alb,test, type="vector")

# si noti che in questo caso l'albero produce la classificazione usando la soglia 0.5

pred=factor(pred)
levels(pred)<-c("no","yes") #in questo modo usiamo per i valori previsti                    #

# infine per la matrice di confusione usiamo semplicemente la funzione table
# (le colonne si riferiscono ai valori veri)

table(pred,test$Exited)
```

```
##
## pred    no  yes
##   no  1921  311
##   yes   69  198
```

```
# e poi da questa otteniamo ad esempio l'accuratezza
sum(diag(table(pred,test$Exited)))/length(pred)
```

```
## [1] 0.8479392
```

In output possiamo chiedere le probabilità e si ottiene una matrice con due colonne. A noi basterà la prima (che è la probabilità di non cambiare banca) per poter fissare la soglia

```
pred1<-predict(alb,test, type="prob")
pred1[1:10,] # Si osservi che i due valori per riga sommano ovviamente a 1
```

```
##           no       yes
## 3  0.2196970 0.7803030
## 8  0.2196970 0.7803030
## 10 0.8856686 0.1143314
## 11 0.8856686 0.1143314
## 16 0.7983740 0.2016260
## 24 0.7983740 0.2016260
## 26 0.8856686 0.1143314
## 28 0.8856686 0.1143314
## 32 0.8856686 0.1143314
## 35 0.8856686 0.1143314
```

```
predclass<-pred1[,1]<0.75 # questa è la soglia fissata
predclass<-factor(predclass)
levels(predclass)<-c("no","yes")
table(predclass,test$Exited)
```

```
##
## predclass   no  yes
##       no  1842  253
```

```
##        yes  148  256
```

Si noti che con la soglia più elevata facciamo più errori sui positivi (quindi un pò più bassa la sensibilità) ma meno sui negativi (migliora la specificità).

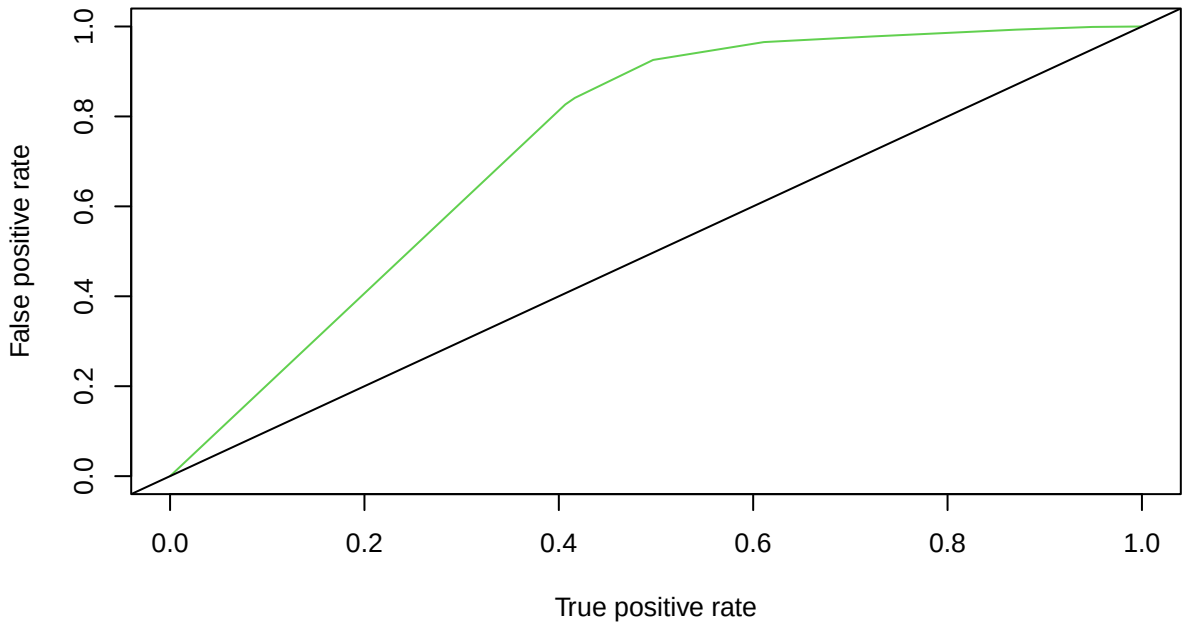
Occorre interrogarsi su quale sia il costo dei due errori in realazione al problema: è più grave non identificare correttamente un cliente che intende andar via oppure sbagliare a classificare un cliente fedele come uno che vuole lasciare la banca?

Disegnare la curva ROC e ottenere vari indici di accuratezza

La curva ROC consente di valutare quale sia la qualità del classificatore prescindendo dalla soglia adottata. Vi sono diversi strumenti per disegnarla. Un pacchetto è `RROC`

Usare ROCR per ottenere la curva ROC

```
library(ROCR)

# occorrerà fornire il vettore delle probabilità oltre che il valore osservato
previsto <- prediction(pred1[,1],test$Exited)

# la libreria RROC cotruisce un oggetto che poi può essere
# usato per avere vari indici di qualità della classificazione
# utilizzandola funzione performance

perf <- performance(previsto, measure = "fpr", x.measure = "tpr", )
plot(perf, col=3)
abline(0,1)
```

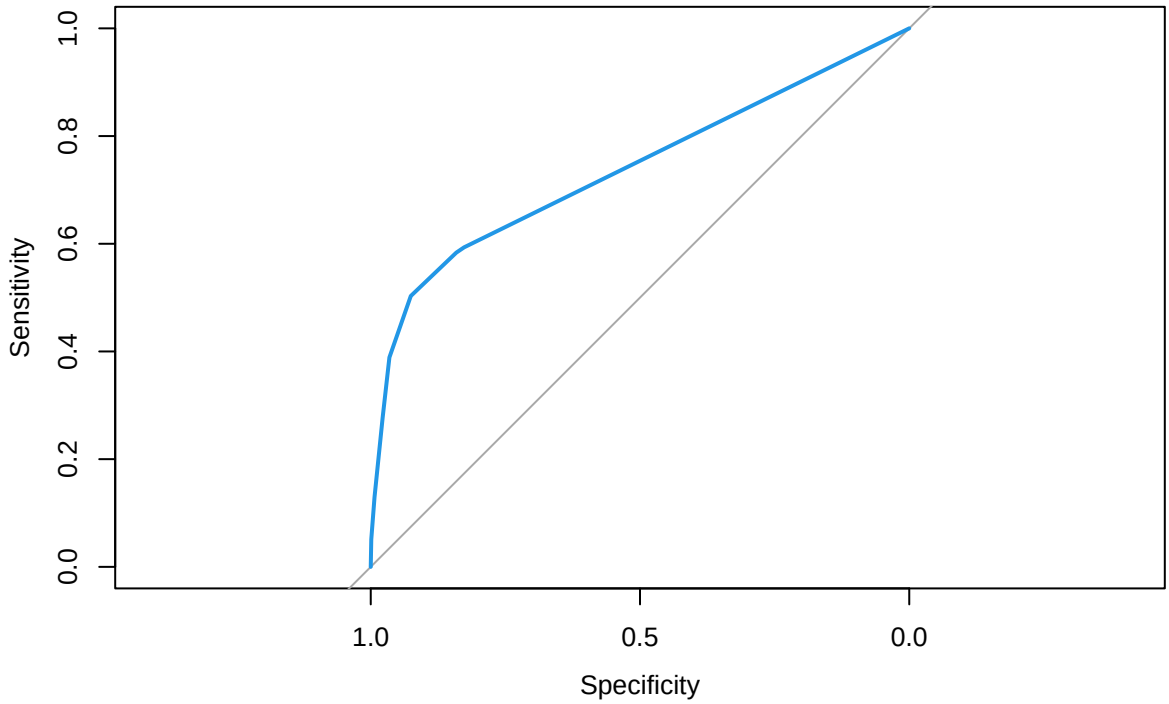


Usare pROC per ottenere la curva ROC (più semplice)

un'altra funzione è pROC, che forse è più semplice della prima, anche se meno generale-

```
library(pROC)
# anche in questto caso va fornito il vettore delle probabilità
```

```
roc(test$Exited, pred1[,1], plot=T, col=4)
```




```
##
## Call:
## roc.default(response = test$Exited, predictor = pred1[, 1], plot = T,     col = 4)
##
## Data: pred1[, 1] in 1990 controls (test$Exited no) > 509 cases (test$Exited yes).
## Area under the curve: 0.7385
```

```
# Si noti che la scala sull'asse orizzontale è invertita
# ma non cambia l'interpretazione
```

Usare caret per allenare un modello

Come detto `caret` contiene numerose funzioni che consentono di utilizzare, ad esempio i diversi metodi per ottenere un algoritmo di classificazione (o previsione). Di solito il modello ottenuto è ottimizzato usando, ove possibile cross-validation.

Il comnado per chiamare un modello e adattarlo al training set è `train()` che contiene al suo interno il parametro `method=` in cui si sceglie il modello fra moltissimi possibili modelli (che includono tutti quelli visti nel corso).

Vediamo ora come adattore un albero usando `caret()`, ma in modo simile si possono usare anche per la regressione logistica, i gam, etc.

```
mod_albero = train(Exited~., data = train,
                   method = "rpart") # l'albero viene ottimizzato cercando i valore di cp migliore
pred_alb = predict(mod_albero, test,type = "prob")[,2] # la funzione predict funziona come prima

mod_albero$results
```

```
##           cp  Accuracy     Kappa  AccuracySD    KappaSD
## 1 0.03141361 0.8425480 0.4121080 0.008431833 0.05289229
```

```
## 2 0.04842932 0.8309391 0.3762396 0.012513346 0.04404675
## 3 0.08540576 0.8029652 0.0698189 0.015450502 0.14257749
```

```
# poi si può procedere come prima per costruire la matrice di confusione
predcaret<-pred_alb>0.5
predcaret<-factor(predcaret)
levels(predcaret)<-c("no","yes")
table(predcaret,test$Exited)
```

```
##
## predcaret   no  yes
##       no  1891  299
##       yes   99  210
```

```
# possiamo usare poi un'altra funzione di caret (che può esser usata indipendentemente)
# per ottenere matrice di confuzione e misure varie relative alla classificazione

confusionMatrix(table(predcaret,test$Exited))
```

```
## Confusion Matrix and Statistics
##
##
## predcaret   no  yes
##       no  1891  299
##       yes   99  210
##
##                Accuracy : 0.8407
##                  95% CI : (0.8258, 0.8549)
##     No Information Rate : 0.7963
##     P-Value [Acc > NIR] : 8.084e-09
##
##                   Kappa : 0.425
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
##             Sensitivity : 0.9503
##             Specificity : 0.4126
##          Pos Pred Value : 0.8635
##          Neg Pred Value : 0.6796
##              Prevalence : 0.7963
##          Detection Rate : 0.7567
##    Detection Prevalence : 0.8764
##       Balanced Accuracy : 0.6814
##
##        'Positive' Class : no
##
```

Altri algoritmi di calssificazione: naiveBayes e KNN

Come detto esistono specifici pacchetti che contengono le funzioni per ottenere classificazioni con altre tecniche. Ad esempio mostriamo come ottenere una classificazione con naiveBayes e KNN. Serve il package e1071.

Classificazione con naive Bayes

```r
library(e1071)

nbclass<-naiveBayes(Exited~Geography+
                      Gender+HasCrCard+Age+
                      Tenure+Balance+NumOfProducts+
                      EstimatedSalary, data=train)

# la struttura è simile a quanto già visto.

nbpred<-predict(nbclass, test, type = c( "raw")) # come per l'albero otteniamo le probabilità

# utilizziamo la stessa strategia per ottenere le matrici di confusione
prednb<-nbpred[,1]<0.75
prednb<-factor(prednb)
levels(prednb)<-c("no","yes")
table(prednb,test$Exited)
```

```
##
## prednb   no  yes
##    no  1619  210
##    yes  371  299
```

Classificazione con KNN

La funzione `gknn()` sempre nello stesso package, usa una distanza generalizzata per valutare un classificatore di tipo KNN

```r
KNNclass<-gknn(Exited~Geography+Gender+
                 HasCrCard+Age+Tenure+Balance+
                 NumOfProducts+EstimatedSalary,
               data = train, k=15, scale = TRUE)
# la funzione ha struttura simile, tuttavia conviene standardizzare le variabili (scale=TRUE) e
# abbiamo provato a fissare k=15. Se non si fissa nulla viene utilizzato
# come default k=1 che è certamente sovradattato

predKNN<-predict(KNNclass, test, type="prob")
# anche in questo caso si hanno le probabilità e tutto procede come prima
KNNclass<-predKNN[,1]<0.5
KNNclass<-factor(KNNclass)
levels(KNNclass)<-c("no","yes")
table(KNNclass,test$Exited)
```

```
##
## KNNclass   no  yes
##      no  1898  304
##      yes   92  205
```

```r
# possiamo costruire le varie metriche come per l'albero
confusionMatrix(table(KNNclass,test$Exited))
```

```
## Confusion Matrix and Statistics
##
##
## KNNclass   no  yes
##      no  1898  304
```

```
##      yes   92  205
##
##                Accuracy : 0.8415
##                  95% CI : (0.8266, 0.8556)
##     No Information Rate : 0.7963
##     P-Value [Acc > NIR] : 4.346e-09
##
##                   Kappa : 0.4219
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
##             Sensitivity : 0.9538
##             Specificity : 0.4028
##          Pos Pred Value : 0.8619
##          Neg Pred Value : 0.6902
##              Prevalence : 0.7963
##          Detection Rate : 0.7595
##    Detection Prevalence : 0.8812
##       Balanced Accuracy : 0.6783
##
##        'Positive' Class : no
##
```

```
# si invita a provare cosa accade  con diversi valori di k
```

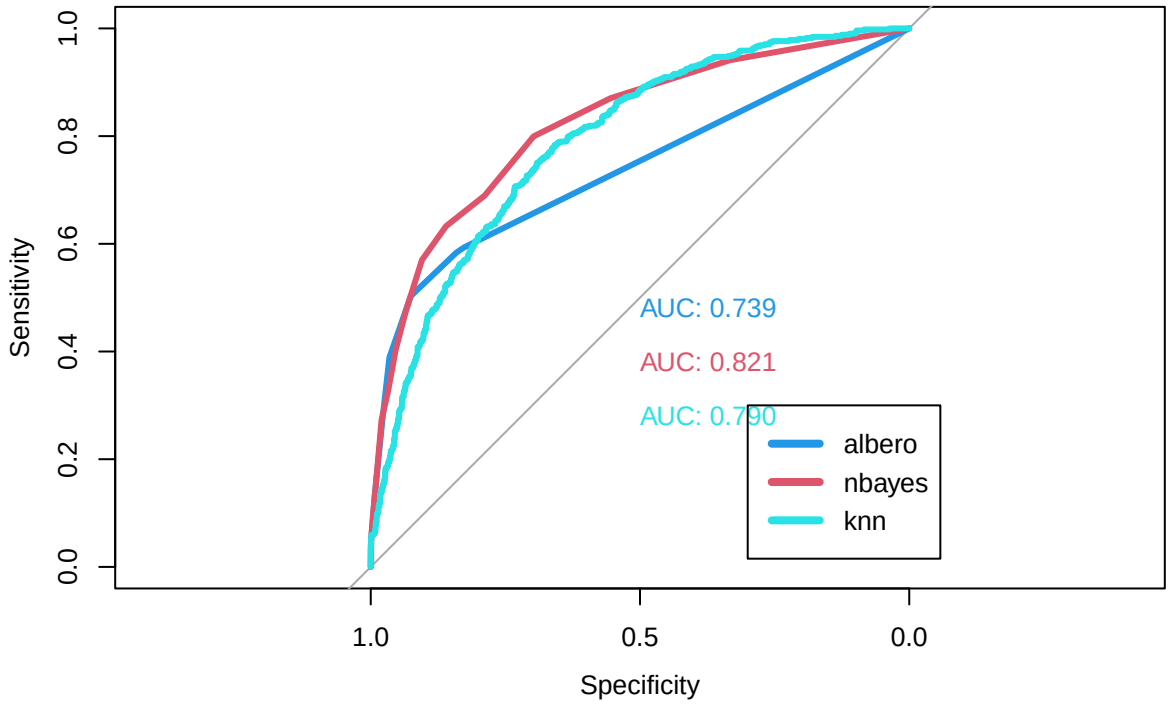
Sovrapporre più curve ROC con pROC

Proviamo ora a ottenere un grafico delle diverse curve ROC, sovrapponendole e calcolando AUC.

```
# occorre fornire i valore del target nel test set e le probabilità
roc(test$Exited, pred1[,1], plot=T, col=4, print.auc=T, lwd=3)
```

```
##
## Call:
## roc.default(response = test$Exited, predictor = pred1[, 1], plot = T,     col = 4, print.auc = T, lwd
##
## Data: pred1[, 1] in 1990 controls (test$Exited no) > 509 cases (test$Exited yes).
## Area under the curve: 0.7385
```

```
#i comandi successivi permettono di sovrapporre le curve per altri classificatori
plot.roc(test$Exited,predKNN[,1], add=T, col=2, print.auc=T,lwd=3, print.auc.y=0.4)
plot.roc(test$Exited,nbpred[,1], add=T, col=5, print.auc=T,lwd=3, print.auc.y=0.3)
legend(0.3,0.3, c("albero", "nbayes", "knn"), col=c(4,2,5), lwd=4)
```



Se ci si basa sulle curve ROC si osservi che naiveBayes sembra quello che fornisce i risulatti migliori.