Fundamentos de la programación estadística y Data Mining en R

Unidad 4. Árboles de Decisión en R

Dr. Germán Rosati (Digital House - UNTREF - UNSAM)

03 agosto, 2017

Implementación de un árbol de decisión mediante CART en R

Usaremos la librería `tree` para construir árboles de decisión y clasificación. Vamos a trabajar con el dataset `carseats`. En este dataset `Sales` (la variable dependiente) es una variable continua. Entonces, vamos a usar `ifelse()` para recodificar en dos valores. Y la agregamos a `Carseats`

```
library(tree)
library(ISLR)
attach(Carseats)
High <- ifelse(Sales <= 8, "No", "Yes")
Carseats <- data.frame(Carseats, High)
```

Ahora usamos la función `tree` para generar un árbol de clasificación que prediga la variable nueva (dicotómica que acabamos de crear). Como vamos a ver la sintaxis de `tree()` es muy similar a la de `lm()` y `glm()`.

```
tree.carseats <- tree(High ~ . - Sales, data = Carseats)
summary(tree.carseats)
```

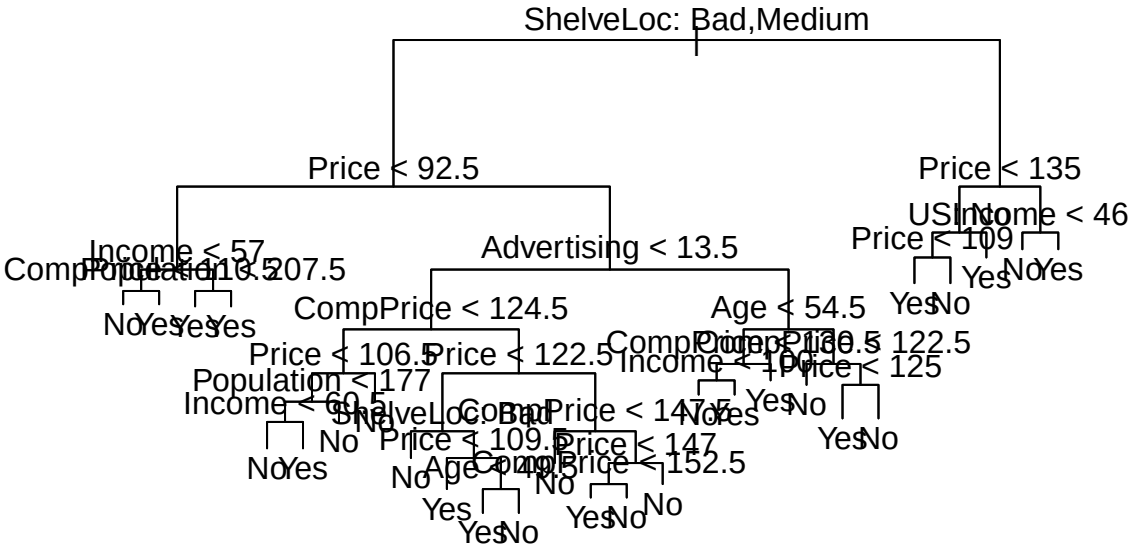
```
##
## Classification tree:
## tree(formula = High ~ . - Sales, data = Carseats)
## Variables actually used in tree construction:
## [1] "ShelveLoc"   "Price"       "Income"      "CompPrice"   "Population"
## [6] "Advertising" "Age"         "US"
## Number of terminal nodes:  27
## Residual mean deviance:  0.4575 = 170.7 / 373
## Misclassification error rate: 0.09 = 36 / 400
```

Vemos que el Training Error es de 9%. En árboles de clasificación lo que reporta la función como "mean deviance" en el output de `summary()` es: $-2\sum_m \sum_k n_{mk} * log * \hat{p}_{mk}$ donde n_{mk} es el número de observaciones en la m-ésima terminal que corresponde a la k-ésima clase. Una desviación pequeña indica un árbol que provee un buen ajuste para los datos (de entrenamiento). La resdial mean deviance es implemente ese valor dividido por $n - |T_0|$.

Implementación de un árbol de decisión mediante CART en R: gráficos

Una cosa interesante y útil que proveen los árboles de decisión es que permiten tener una salida gráfica bastante útil e intuitiva del modelo. Usamos la función `plot()` para graficar la estructura y `text()` para agregarle las "etiquetas". El argumento `pretty=0` le dice a R que muestre las categorías completas de cada cualquier predictor cualitativo.

```
plot(tree.carseats)
text(tree.carseats, pretty = 0)
```



La ubicación en las repisas (`shelving location`) parece ser el mejor predictor de `Sales`, dado que la primera rama del árbol diferencia las buenas locaciones de las malas y medianas. Si tipeamos el nombre del objeto árbol R imprime un output correspondiente a cada rama. Imprime, además, el criterio de splitting (`Price` < 92.5), la cantidad de observciones en esa rama, el desvío, la predicción general de la rama (`Yes` o `No`) y la fracción de observaciones dentro de esa rama que toman valores `Yes` y `No`. Las ramas que llegan hasta las terminales aparecen con asteriscos.

```
tree.carseats
```

```
## node), split, n, deviance, yval, (yprob)
##       * denotes terminal node
##
##   1) root 400 541.500 No ( 0.59000 0.41000 )
##     2) ShelveLoc: Bad,Medium 315 390.600 No ( 0.68889 0.31111 )
##       4) Price < 92.5 46  56.530 Yes ( 0.30435 0.69565 )
##         8) Income < 57 10  12.220 No ( 0.70000 0.30000 )
##          16) CompPrice < 110.5 5   0.000 No ( 1.00000 0.00000 ) *
##          17) CompPrice > 110.5 5   6.730 Yes ( 0.40000 0.60000 ) *
##         9) Income > 57 36  35.470 Yes ( 0.19444 0.80556 )
##          18) Population < 207.5 16  21.170 Yes ( 0.37500 0.62500 ) *
##          19) Population > 207.5 20   7.941 Yes ( 0.05000 0.95000 ) *
##       5) Price > 92.5 269 299.800 No ( 0.75465 0.24535 )
##        10) Advertising < 13.5 224 213.200 No ( 0.81696 0.18304 )
##          20) CompPrice < 124.5 96  44.890 No ( 0.93750 0.06250 )
##            40) Price < 106.5 38  33.150 No ( 0.84211 0.15789 )
##              80) Population < 177 12  16.300 No ( 0.58333 0.41667 )
##               160) Income < 60.5 6   0.000 No ( 1.00000 0.00000 ) *
##               161) Income > 60.5 6   5.407 Yes ( 0.16667 0.83333 ) *
##              81) Population > 177 26   8.477 No ( 0.96154 0.03846 ) *
##            41) Price > 106.5 58   0.000 No ( 1.00000 0.00000 ) *
##          21) CompPrice > 124.5 128 150.200 No ( 0.72656 0.27344 )
##            42) Price < 122.5 51  70.680 Yes ( 0.49020 0.50980 )
##              84) ShelveLoc: Bad 11   6.702 No ( 0.90909 0.09091 ) *
##              85) ShelveLoc: Medium 40  52.930 Yes ( 0.37500 0.62500 )
##               170) Price < 109.5 16   7.481 Yes ( 0.06250 0.93750 ) *
##               171) Price > 109.5 24  32.600 No ( 0.58333 0.41667 )
##                 342) Age < 49.5 13  16.050 Yes ( 0.30769 0.69231 ) *
##                 343) Age > 49.5 11   6.702 No ( 0.90909 0.09091 ) *
```

```
##            43) Price > 122.5 77  55.540 No ( 0.88312 0.11688 )
##              86) CompPrice < 147.5 58  17.400 No ( 0.96552 0.03448 ) *
##              87) CompPrice > 147.5 19  25.010 No ( 0.63158 0.36842 )
##               174) Price < 147 12  16.300 Yes ( 0.41667 0.58333 )
##                 348) CompPrice < 152.5 7   5.742 Yes ( 0.14286 0.85714 ) *
##                 349) CompPrice > 152.5 5   5.004 No ( 0.80000 0.20000 ) *
##               175) Price > 147 7   0.000 No ( 1.00000 0.00000 ) *
##         11) Advertising > 13.5 45  61.830 Yes ( 0.44444 0.55556 )
##           22) Age < 54.5 25  25.020 Yes ( 0.20000 0.80000 )
##             44) CompPrice < 130.5 14  18.250 Yes ( 0.35714 0.64286 )
##               88) Income < 100 9  12.370 No ( 0.55556 0.44444 ) *
##               89) Income > 100 5   0.000 Yes ( 0.00000 1.00000 ) *
##             45) CompPrice > 130.5 11   0.000 Yes ( 0.00000 1.00000 ) *
##           23) Age > 54.5 20  22.490 No ( 0.75000 0.25000 )
##             46) CompPrice < 122.5 10   0.000 No ( 1.00000 0.00000 ) *
##             47) CompPrice > 122.5 10  13.860 No ( 0.50000 0.50000 )
##               94) Price < 125 5   0.000 Yes ( 0.00000 1.00000 ) *
##               95) Price > 125 5   0.000 No ( 1.00000 0.00000 ) *
##      3) ShelveLoc: Good 85  90.330 Yes ( 0.22353 0.77647 )
##        6) Price < 135 68  49.260 Yes ( 0.11765 0.88235 )
##         12) US: No 17  22.070 Yes ( 0.35294 0.64706 )
##           24) Price < 109 8   0.000 Yes ( 0.00000 1.00000 ) *
##           25) Price > 109 9  11.460 No ( 0.66667 0.33333 ) *
##         13) US: Yes 51  16.880 Yes ( 0.03922 0.96078 ) *
##        7) Price > 135 17  22.070 No ( 0.64706 0.35294 )
##         14) Income < 46 6   0.000 No ( 1.00000 0.00000 ) *
##         15) Income > 46 11  15.160 Yes ( 0.45455 0.54545 ) *
```

Implementación de un árbol de decisión mediante CART en R: training y test sets

Para evaluar la performance de clasificacione en estos datos, tensmo sque esimar el test error. Dividimos las observaciones en test y training sets, creamos un árbol en el training set y evaluamos en el test-set. La función `predict()` se puede usar (de la misma forma que el `glm()` o `lm()`). En el caso de un árbol de clasificación el argumento `type="class"` le dice a R que devuelva la categoría predicha. En este caso, vemos que la tasa de predicciones correctas cambia a 71.5 % sobre el test set.

```
set.seed(2)
train <- sample(1:nrow(Carseats), 200)
Carseats.test <- Carseats[-train, ]
High.test <- High[-train]
tree.carseats <- tree(High ~ . - Sales, Carseats, subset = train)
tree.pred <- predict(tree.carseats, Carseats.test, type = "class")
table(tree.pred, High.test)
```

```
##          High.test
## tree.pred No Yes
##       No  86  27
##       Yes 30  57
```

Luego, podemos pensar en "podar" el árbol para obtener mejores resultados. la función `cv.tree()` realiza una validación cruzada para determinar el nivel óptimo de complejidad del árbol. "Cost complexity prunning" (podado basado en costo-complejidad) se usa para seleccinar una secuencia de árboles. Usamos el argumento `FUN = prune` para indicar que queremos que el error de clasificación sea el valor que se use para el proceso

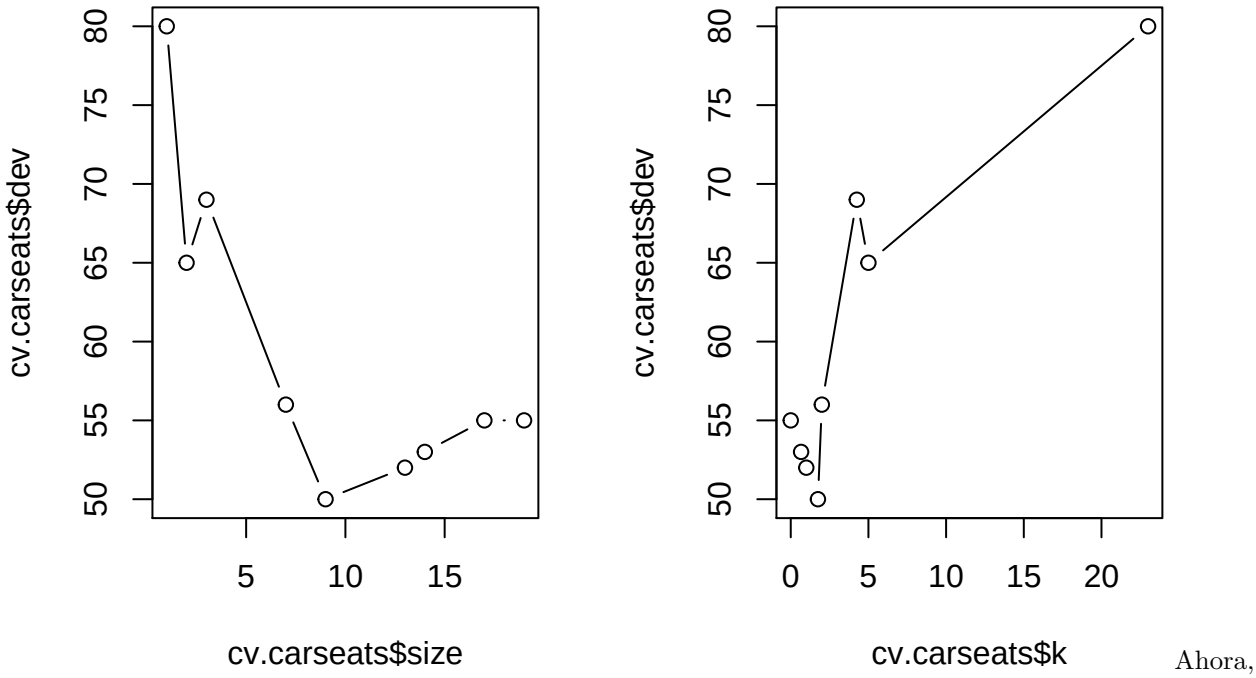
de podado. La función `cv.tree()` usa por defecto el desvío. A su vez, `cv.tree()` reporta la cantidad de nodos terminales en cada árbol considerado (size), así como el parámetro de costo-complejidad usado (k que se corresponde con lo que llamamos antes α).

```
set.seed(3)
cv.carseats <- cv.tree(tree.carseats, FUN = prune.misclass)
names(cv.carseats)
```

```
## [1] "size"   "dev"    "k"      "method"
```

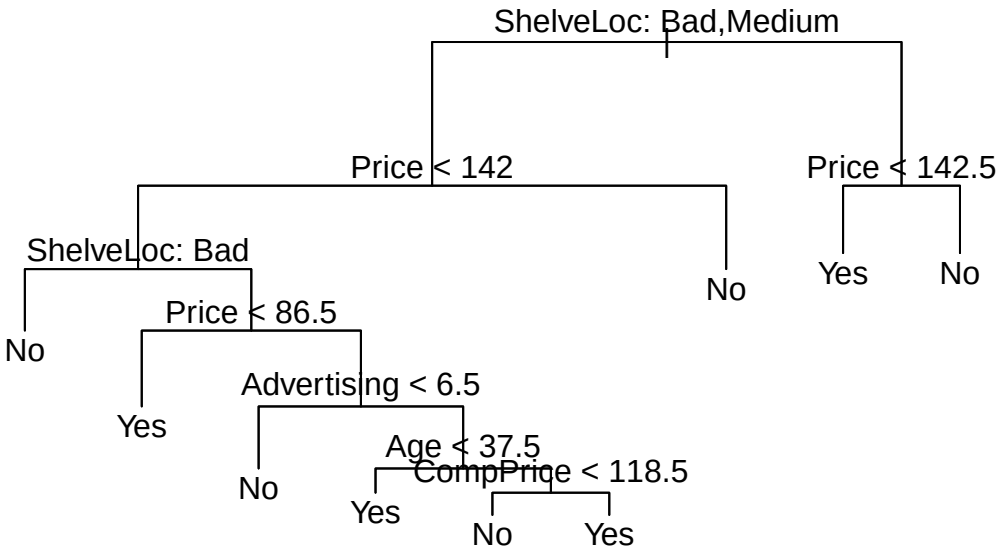
Notar que, a pesar del nombre `dev` corresponde a la tasa de error "cros-validada". El árbol con 9 nodos terminales es el que tiene el menor error de clasificación "cros-validado" (50). Graficamos la tasa de error como una función del tamaño del árbol y de k.

```
par(mfrow = c(1, 2))
plot(cv.carseats$size, cv.carseats$dev, type = "b")
plot(cv.carseats$k, cv.carseats$dev, type = "b")
```



Ahora, podemos aplciar`prune.misclass()`para podar el árbol y quedarnos con el de 9 nodos.

```
prune.carseats <- prune.misclass(tree.carseats, best = 9)
plot(prune.carseats)
text(prune.carseats, pretty = 0)
```



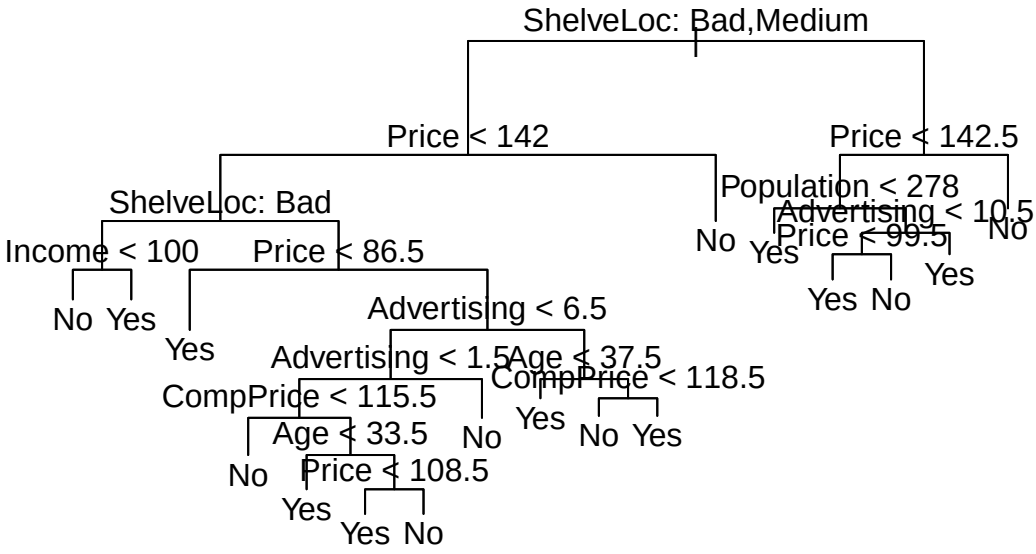
¿Qué tan bien funciona este árbol de 9 nodos en el test-set? Usamos de nuevo `predict()`:

```
tree.pred <- predict(prune.carseats, Carseats.test, type = "class")
table(tree.pred, High.test)
```

```
##          High.test
## tree.pred No Yes
##       No  94  24
##       Yes 22  60
```

Ahora, el 77% de las observaciones están bien clasificadas. Es decir, que este proceso de cost-complexity prunning generó no solamente un árbol más interpretable sino que además mejoró la performance preditiva. Si incrementamos el valor de `best`, obtenemos un árbol más grande y con peor performance de clasificación.

```
prune.carseats <- prune.misclass(tree.carseats, best = 15)
plot(prune.carseats)
text(prune.carseats, pretty = 0)
```




```
tree.pred <- predict(prune.carseats, Carseats.test, type = "class")
table(tree.pred, High.test)
```

```
##          High.test
## tree.pred No Yes
##       No  86  22
##       Yes 30  62
```