Aprendizaje Automático (2014-2015)

Grado en Ingeniería Informática

Universidad de Granada

Informe Práctica 3

Mª Cristina Heredia Gómez

31 de mayo de 2015

Índice

Índice de figuras

1. Ejercicio.-1 (3 puntos) (comentar los resultados de todos los apartados) Usar el conjunto de datos OJ que es parte del paquete ISLR

1.1. Crear un conjunto de entrenamiento conteniendo una muestra aleatoria de 800 observaciones, y un conjunto de test conteniendo el resto de las observaciones. Ajustar un clasificador SVM (con núcleo lineal) a los datos de entrenamiento usando cost=0.01, con "Purchase" como la respuesta y las otras variables como predictores.

Dejando las etiquetas en la posición en la que se encuentran(columna 1):

```
> library(ISLR)
> attach(OJ)
> set.seed(1)
> nrow(OJ)
[1] 1070
> train0=sample(nrow(OJ),800)
> train=OJ[train0,]
> test=OJ[-train0,]
> dim(test)
[1] 270  18
> dim(train)
[1] 800  18
```

Figura 1.1: conjunto de entrenamiento y test

```
> svmfit = svm( Purchase~. ,data =train, kernel ="linear" , cost =0.01 )
> summary(svmfit)

Call:
svm(formula = Purchase ~ ., data = train, kernel = "linear", cost = 0.01)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  linear
       cost:  0.01
      gamma:  0.05555556

Number of Support Vectors:  432

 ( 215 217 )


Number of Classes:  2

Levels:
 CH MM
```

Figura 1.2: clasificador SVM a los datos de entrenamiento

Ahora, hacemos una prueba quitando la primera columna y poniéndola al final, es decir, cambiamos las etiquetas a la última posición:

```
> etiqueta=train[,1]
> train=train[ , -1]
>
> dat=data.frame(train, etiqueta=as.factor(etiqueta))
> library(e1071)
> svmfit = svm( etiqueta~. ,data =dat, kernel ="linear" , cost =0.01 )
> summary(svmfit)

Call:
svm(formula = etiqueta ~ ., data = dat, kernel = "linear", cost = 0.01)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  linear
       cost:  0.01
      gamma:  0.05555556

Number of Support Vectors:  432

 ( 215 217 )


Number of Classes:  2

Levels:
 CH MM
```

Figura 1.3: clasificador SVM a los datos de entrenamiento

y observamos que obtenemos el mismo ajuste, luego a la función svm no le importa la posición de las etiquetas en el conjunto.

1.2. Usar la función summary() para producir un resumen estadístico, y describir los resultados obtenidos. ¿Cuáles son las tasas de error de "training" y "test"?

Basándonos en el apartado anterior, con summary(), tenemos que en el ajuste se usa un núcleo lineal de coste=0.01, y que hay 432 soportes de vectores y dos clases; una clase con 215 y otra con 217.

```
> predict.train = predict(svmfit,train)
> table(predict.train,etiqueta)
             etiqueta
predict.train  CH  MM
           CH 439  78
           MM  55 228
> (55+78)/(439+228+55+78)
[1] 0.16625
```

Figura 1.4: tasa error training

```
> predict.test = predict(svmfit,test)
> table(predict.test,test$Purchase)

predict.test  CH  MM
          CH 141  31
          MM  18  80
> (18+31)/(141+31+80+18)
[1] 0.1814815
```

Figura 1.5: tasa error test

Vemos que el error de training y el de test son muy parecidos, pues para los datos training obtenemos un error del 16.625 % , mientras que para test tenemos un error del 18.15 %

1.3. Usar las función tune() para seleccionar un coste óptimo. Considerar los valores de "cost" del vector: [0.001, 0.01, 0.1, 1, 10]. Dibujar las curvas ROC para los diferentes valores del "cost".

```
> tune.out=tune(svm, etiqueta~. , data =dat, kernel ="linear" ,ranges=list(cost=c(0.001,0.01,0.1,1,10)))
> summary(tune.out)

Parameter tuning of 'svm':

- sampling method: 10-fold cross validation

- best parameters:
 cost
 0.01

- best performance: 0.165

- Detailed performance results:
   cost   error dispersion
1 1e-03 0.34250 0.05927806
2 1e-02 0.16500 0.04362084
3 1e-01 0.16875 0.04135299
4 1e+00 0.16750 0.04495368
5 1e+01 0.16625 0.04566256
```

Figura 1.6: seleccionando coste óptimo con tune

Vemos que, por defecto, tune() hace una cross-validation de 10 particiones. Tenemos que coste=0.01 es el menor error obtenido en la cross-validation. Además, podemos acceder al mejor modelo, para el que se ha obtenido este resultado, que resulta que es el que ajustamos en el apartado 1:

```
> bestmod=tune.out$best.model
> summary(bestmod)

Call:
best.tune(method = svm, train.x = etiqueta ~ ., data =


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  linear
       cost:  0.01
      gamma:  0.05555556

Number of Support Vectors:  432

 ( 215 217 )


Number of Classes:  2

Levels:
 CH MM
```

Figura 1.7: mejor modelo obtenido con tune

Volvemos a declarar conjuntos test y train, y creamos la función que pinta las curvas roc:

```
> library(ISLR)
> attach(OJ)
> set.seed(1)
> train0=sample(nrow(OJ),800)
> train=OJ[train0,]
> test=OJ[-train0]
> library(ROCR)
Loading required package: gplots

Attaching package: 'gplots'

The following object is masked from 'package:stats':

    lowess

> rocplot = function( pred , truth , ...) {
+ predob = prediction ( pred , truth )
+ perf = performance( predob , "tpr" , "fpr")
+ plot ( perf, ...) }
```

Figura 1.8: función ROCPLOT

```
> svmfit.opt = svm( Purchase~. , data = train , kernel ="linear" , cost =0.001 , decision.values=T )
> fitted = attributes( predict(svmfit.opt, train, decision.value=TRUE))$decision.values
> rocplot(fitted ,OJ[train0,"Purchase"], main=" Training Data" ,col="red",add=par("new"))
> par(new=T)
```

Figura 1.9: ajuste svm con núcleo lineal y coste=0.001

```
> svmfit2.opt = svm( Purchase~. , data = train , kernel ="linear" , cost =0.01 , decision.values=T )
> fitted2 = attributes( predict(svmfit2.opt, train, decision.value=TRUE))$decision.values
> rocplot(fitted2 ,OJ[train0,"Purchase"], main=" Training Data" ,col="navy",add=par("new"))
> par(new=T)
```

Figura 1.10: ajuste svm con núcleo lineal y coste=0.01

```
> #COSTE=0.1
> svmfit3.opt = svm( Purchase~. , data = train , kernel ="linear" , cost =0.1 , decision.values=T )
> fitted3 = attributes( predict(svmfit3.opt, train, decision.value=TRUE))$decision.values
> rocplot(fitted3 ,OJ[train0,"Purchase"], main=" Training Data" ,colorize=T)
```

Figura 1.11: ajuste svm con núcleo lineal y coste=0.1

```
> svmfit4.opt = svm( Purchase~. , data = train , kernel ="linear" , cost =1 , decision.values=T )
> fitted4 = attributes( predict(svmfit4.opt, train, decision.value=TRUE))$decision.values
> rocplot(fitted4 ,OJ[train0,"Purchase"], main=" Training Data" ,col="orange",add=par("new"))
> par(new=T)
```

Figura 1.12: ajuste svm con núcleo lineal y coste=1

```
> svmfit5.opt = svm( Purchase~. , data = train , kernel ="linear" , cost =10 , decision.values=T )
> fitted5 = attributes( predict(svmfit5.opt, train, decision.value=TRUE))$decision.values
> rocplot(fitted5 ,OJ[train0,"Purchase"], main=" Training Data" ,col="pink",add=par("new"))
> par(new=T)
>
> legend(0.70, 0.48,paste("coste=",c(0.001,0.01,0.1,1,10)), col=c("red","blue","green","orange","pin
k"),
+        text.col=c("red","navy","green","orange","pink"), lty=1,
+        merge=TRUE, bg="honeydew", cex=0.6, title="Costes:" ,title.col="black")
```

Figura 1.13: ajuste svm con núcleo lineal y coste=10

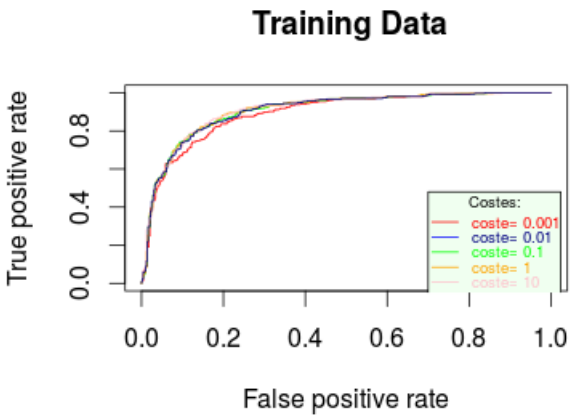

Figura 1.14: curvas ROC para cada uno de los costes

Y, vemos que la gráfica que parece ser la mejor es la azul navy, que representa al coste 0.01, que es el que nos dijo tune() que era el coste óptimo.

Conclusión: Para los valores posibles del cost, tenemos que el mejor ajuste de clasificador SVM **con núcleo lineal** viene dado por **coste=0.01**, para el que teníamos **432 soportes de vectores, con dos clases, una con 215 y otra con 217** elementos, y unos errores **train de 16.625 % y test de 18.15 %** (ver apartado 1.4).

1.4. Calcular las tasas de error de "training" y "test" usando el nuevo valor de coste óptimo.

(esto ya está resuelto en apartados anteriores).
Como el valor de coste óptimo era 0.01, predecimos usando ese modelo, que ya lo hemos creado en el apartado anterior y lo hemos llamado svmfit2.opt. Obtenemos así un error train del 16.125 % y un error test del 16.63 %. Observamos pues, que ambos errores son menores que los obtenidos en el apartado uno,en el que usábamos un coste no óptimo de 0.01.

```
> predict.train2 = predict(svmfit2.opt,train)
> table(predict.train2, train$Purchase)

predict.train2  CH  MM
            CH 439  78
            MM  55 228
> (55+78)/(439+228+55+78)
[1] 0.16625
```

Figura 1.15: error train usando el coste óptimo

```
> predict.test2 = predict(svmfit2.opt,test)
> table(predict.test2,test$Purchase)

predict.test2  CH  MM
           CH 141  31
           MM  18  80
> (18+31)/(141+80+18+31)
[1] 0.1814815
```

Figura 1.16: error test usando el coste óptimo

1.5. Repetir apartados (2) a (4) usando un SVM con núcleo radial. Usar valores de gamma en el rango [10, 1, 0.1, 0.01, 0.001]. Discutir los resultados

1.5.1. SVM con núcleo radial y γ =10

```
> svmfit1.rad = svm( Purchase~. , data = train , kernel ="radial" ,gamma=10, cost =0.001 , decision.values=T )
> summary(svmfit1.rad)

Call:
svm(formula = Purchase ~ ., data = train, kernel = "radial", gamma = 10, cost = 0.001, decision.values = T)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  radial
       cost:  0.001
      gamma:  10

Number of Support Vectors:  615

 ( 306 309 )


Number of Classes:  2

Levels:
 CH MM
```

Figura 1.17: clasificador svm con gamma=10 y coste=0.001

coste=0.001: Observamos que se ha ajustado un clasificador SVM , usando núcleo radial con $\gamma = 10$ y coste=0.01, y que hay 615 soportes de vectores y dos clases, una clase con 306 y otra con 309.

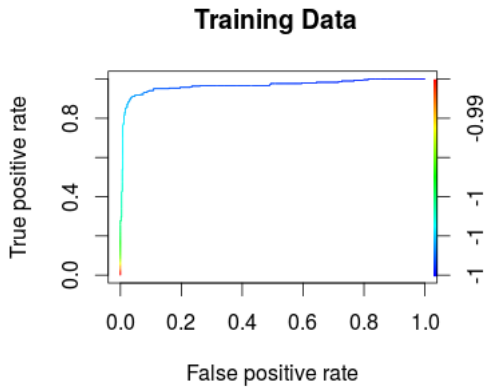

Figura 1.18: curva ROC para clasificador svm con kernel radial, para γ =10 y coste=0.01 sobre datos de train

```
> predict.trainRad1 = predict(svmfit1.rad,train)
> table(predict.trainRad1, train$Purchase)

predict.trainRad1  CH  MM
               CH 494 306
               MM   0   0
> (306)/(494+306)
[1] 0.3825
```

Figura 1.19: error de train para ajuste de clasificador svm con kernel radial, con $\gamma = 10$ y coste=0.01

```
> predict.testRad1 = predict(svmfit1.rad,test)
> table(predict.testRad1,test$Purchase)

predict.testRad1  CH  MM
              CH 159 111
              MM   0   0
> (111)/(159+111)
[1] 0.4111111
```

Figura 1.20: error de test para ajuste de clasificador svm con kernel radial, con $\gamma = 10$ y coste=0.01

Y vemos que para estos valores de γ y coste, obtenemos un error de train de 38.25 % y un error de test de 41.11 %.
Sin embargo,comprobamos que este proceso resulta muy tedioso si tenemos que repetirlo para cada uno de los valores de γ y cada uno de los posibles costes. Por eso, usaremos la función tune(), que nos dirá cuál es el mejor coste para un valor dado de γ , y nos centraremos en analizar sólo ese caso, para cada uno de los γ (para cada γ el mejor coste).

```
> set.seed(2)
> tune.outRad10=tune(svm, Purchase~. , data =train, kernel ="radial" ,ranges=list(cost=c(0.001,0.01,0.1,1,10)), gamma=10)
> bestmodRad10=tune.outRad10$best.model
> summary(bestmodRad10)

Call:
best.tune(method = svm, train.x = Purchase ~ ., data = train, ranges = list(cost = c(0.001, 0.01, 0.1, 1, 10)),
    kernel = "radial", gamma = 10)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  radial
       cost:  1
      gamma:  10

Number of Support Vectors:  647

 ( 266 381 )


Number of Classes:  2

Levels:
 CH MM
```

Figura 1.21: mejor coste para clasificador svm con kernel radial y γ =10

y vemos que el mejor coste para γ =10, es 1. Con este ajuste obtenemos un clasificador SVM con 647 soportes de vectores, y dos clases, una con 266 y otra con 381.

```
> svmfitRad.10best = svm( Purchase~. , data = train , kernel ="radial" ,gamma=10, cost =1 , decision.values=T )
> fittedrad10 = attributes( predict(svmfitRad.10best, train, decision.value=TRUE))$decision.values
> rocplot(fittedrad10 ,OJ[train0,"Purchase"], main=" Training Data" ,colorize=T)
> predict.trainRad10 = predict(svmfitRad.10best,train)
> table(predict.trainRad10, train$Purchase)

predict.trainRad10  CH  MM
                CH 482  37
                MM  12 269
> (12+37)/(482+269+12+37)
[1] 0.06125
```

Figura 1.22: error de train para ajuste de clasificador svm con kernel radial, con γ =10 y coste=1

```
> predict.testRad10 = predict(svmfitRad.10best,test)
> table(predict.testRad10,test$Purchase)

predict.testRad10  CH  MM
                CH 142  47
                MM  17  64
> (17+47)/(142+47+17+64)
[1] 0.237037
```

Figura 1.23: error de test para ajuste de clasificador svm con kernel radial, con γ =10 y coste=1

vemos que con este ajuste del clasificador,obtenemos un error de train del 6.125 % pero un error test de 23.70 %.

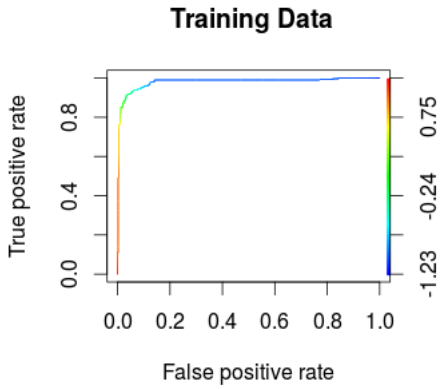

Figura 1.24: curva ROC para clasificador svm con kernel radial, para $\gamma = 10$ y coste=1 sobre datos de train

1.5.2. SVM con núcleo radial y γ =1

```
> tune.outRad1=tune(svm, Purchase~. , data =train, kernel ="radial" ,ranges=list(cost=c(0.001,0.01,0.1,1,10)),gam
ma=1)
> bestmodRad1=tune.outRad1$best.model
> summary(bestmodRad1)

Call:
best.tune(method = svm, train.x = Purchase ~ ., data = train, ranges = list(cost = c(0.001, 0.01,
    0.1, 1, 10)), kernel = "radial", gamma = 1)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  radial
       cost:  1
      gamma:  1

Number of Support Vectors:  483

 ( 218 265 )


Number of Classes:  2

Levels:
 CH MM
```

Figura 1.25: mejor coste para clasificador svm con kernel radial y γ =1

Obveramos que para γ =1, tenemos que el mejor coste es 1 también. Con este ajuste tenemos 483 soportes de vectores, y dos clases; una con 218 y otra con 265. Pintamos gráfica ROCPLOT y calculamos errores de train y test:

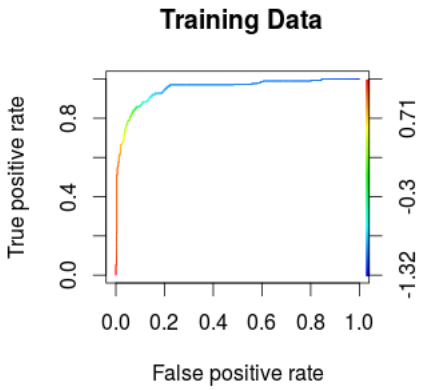

Figura 1.26: curva ROC para clasificador svm con kernel radial, para γ =1 y coste=1 sobre datos de train

```
> svmfitRad.1best = svm( Purchase~. , data = train , kernel ="radial" ,gamma=1, cost =1 , decision.values=T )
> fittedrad1 = attributes( predict(svmfitRad.1best, train, decision.value=TRUE))$decision.values
> rocplot(fittedrad1 ,OJ[train0,"Purchase"], main=" Training Data" ,colorize=T)
> predict.trainRad1 = predict(svmfitRad.1best,train)
> table(predict.trainRad1, train$Purchase)

predict.trainRad1  CH  MM
               CH 459  51
               MM  35 255
> (35+51)/(459+255+35+51)
[1] 0.1075
```

Figura 1.27: error de train para ajuste de clasificador svm con kernel radial, con γ =1 y coste=1

```
> predict.testRad1 = predict(svmfitRad.1best,test)
> table(predict.testRad1,test$Purchase)

predict.testRad1  CH  MM
              CH 136  33
              MM  23  78
> (23+33)/(136+33+23+78)
[1] 0.2074074
```

Figura 1.28: error de test para ajuste de clasificador svm con kernel radial, con γ =1 y coste=1

Y vemos que para este ajuste del clasificador,para el que γ =10, obtenemos un error de train del 10.75 % y un error de test de 20.74 %

1.5.3. SVM con núcleo radial y γ =0.1

```
> set.seed(2)
> tune.outRad0.1=tune(svm, Purchase~. , data =train, kernel ="radial" ,ranges=list(cost=c(0.001,0.01,0.1,1,10)),gamma=0.1)
> bestmodRad0.1=tune.outRad0.1$best.model
> summary(bestmodRad0.1)

Call:
best.tune(method = svm, train.x = Purchase ~ ., data = train, ranges = list(cost = c(0.001, 0.01, 0.1,
    1, 10)), kernel = "radial", gamma = 0.1)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  radial
       cost:  1
      gamma:  0.1

Number of Support Vectors:  380

 ( 185 195 )


Number of Classes:  2

Levels:
 CH MM
```

Figura 1.29: mejor coste para clasificador svm con kernel radial y γ =0.1

Vemos que ahora el mejor ajuste de svm con núcleo radial que se puede obtener para $\gamma = 0{,}1$ es con coste=1, en el cual tenemos un total de 380 soportes de vectores, donde 185 pertenecen a una clase y 195 pertenecen a otra (dos clases).Pintamos gráfica ROCPLOT y calculamos errores de train y test:

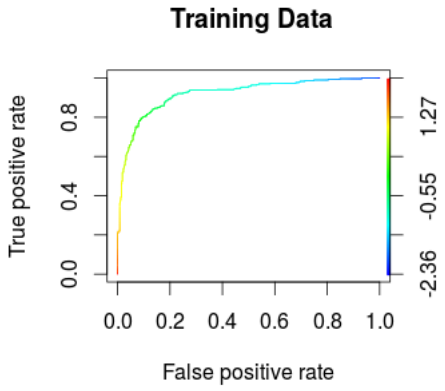

Figura 1.30: curva ROC para clasificador svm con kernel radial, para $\gamma = 0{,}1$ y coste=1 sobre datos de train

```
> svmfitRad0.1best = svm( Purchase~. , data = train , kernel ="radial" ,gamma=0.1, cost =1 , decision.values=T
)
> fittedrad0.1 = attributes( predict(svmfitRad0.1best, train, decision.value=TRUE))$decision.values
> rocplot(fittedrad0.1 ,OJ[train0,"Purchase"], main=" Training Data" ,colorize=T)
> predict.trainRad0.1 = predict(svmfitRad0.1best,train)
> table(predict.trainRad0.1, train$Purchase)

predict.trainRad0.1  CH  MM
                 CH 455  75
                 MM  39 231
> (39+75)/(455+231+39+75)
[1] 0.1425
```

Figura 1.31: error de train para ajuste de clasificador svm con kernel radial, con γ =0.1 y coste=1

```
> predict.testRad0.1 = predict(svmfitRad0.1best,test)
> table(predict.testRad0.1,test$Purchase)

predict.testRad0.1  CH  MM
                CH 597 103
                MM  56 314
> (56+103)/(597+314+56+103)
[1] 0.1485981
```

Figura 1.32: error de test para ajuste de clasificador svm con kernel radial, con γ =0.1 y coste=1

Y vemos que para este ajuste del clasificador,para el que γ es 1 y coste es 1, obtenemos un error de train del 14.25 % y un error de test de 16.66 % .

1.5.4. SVM con núcleo radial y γ =0.01

```
> set.seed(2)
> tune.outRad0.01=tune(svm, Purchase~. , data =train, kernel ="radial" ,ranges=list(cost=c(0.001,0.01,0.1,1,10)),gamma=0.01)
> bestmodRad0.01=tune.outRad0.01$best.model
> summary(bestmodRad0.01)

Call:
best.tune(method = svm, train.x = Purchase ~ ., data = train, ranges = list(cost = c(0.001,
    0.01, 0.1, 1, 10)), kernel = "radial", gamma = 0.01)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  radial
       cost:  1
      gamma:  0.01

Number of Support Vectors:  406

 ( 203 203 )


Number of Classes:  2

Levels:
 CH MM
```

Figura 1.33: mejor coste para clasificador svm con kernel radial y γ =0.01

Vemos que ahora el mejor ajuste de svm con núcleo radial que se puede obtener para γ =0.01 es con coste=1, en el cual tenemos un total de 406 soportes de vectores, donde 203 pertenecen a una clase y 203 pertenecen a otra (dos clases).Pintamos gráfica ROCPLOT y calculamos errores de train y test:

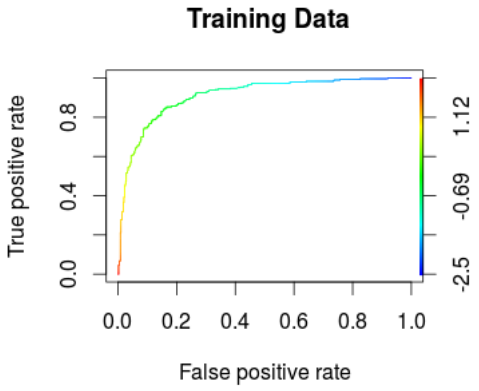

Figura 1.34: curva ROC para clasificador svm con kernel radial, para γ =0.1 y coste=1 sobre datos de train

```
> fittedrad0.01 = attributes( predict(svmfitRad0.01best, train, decision.value=TRUE))$decision.values
> rocplot(fittedrad0.01 ,OJ[train0,"Purchase"], main=" Training Data" ,colorize=T)
> predict.trainRad0.01 = predict(svmfitRad0.01best,train)
> table(predict.trainRad0.01, train$Purchase)

predict.trainRad0.01  CH  MM
                  CH 443  75
                  MM  51 231
> (51+75)/(443+231+51+75)
[1] 0.1575
```

Figura 1.35: error de train para ajuste de clasificador svm con kernel radial, con γ =0.1 y coste=1

```
> predict.testRad0.01 = predict(svmfitRad0.01best,test)
> table(predict.testRad0.01,test$Purchase)

predict.testRad0.01  CH  MM
                 CH 144  32
                 MM  15  79
> (15+32)/(144+79+15+32)
[1] 0.1740741
```

Figura 1.36: error de test para ajuste de clasificador svm con kernel radial, con γ =0.1 y coste=1

Y vemos que para este ajuste del clasificador, obtenemos un error de train del 15.75 % y un error de test de 17.41 %.

1.5.5. SVM con núcleo radial y γ =0.001

```
> set.seed(2)
> tune.outRad0.001=tune(svm, Purchase~. , data =train, kernel ="radial" ,ranges=list(cost=c(0.001,0.01,0.1,1,10)),
gamma=0.001)
> bestmodRad0.001=tune.outRad0.001$best.model
> summary(bestmodRad0.001)

Call:
best.tune(method = svm, train.x = Purchase ~ ., data = train, ranges = list(cost = c(0.001, 0.01,
    0.1, 1, 10)), kernel = "radial", gamma = 0.001)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  radial
       cost:  10
      gamma:  0.001

Number of Support Vectors:  394

 ( 195 199 )


Number of Classes:  2

Levels:
 CH MM
```

Figura 1.37: mejor coste para clasificador svm con kernel radial y γ =0.001

En este caso tenemos que el mejor ajuste de svm con núcleo radial que se puede obtener para γ =0.001 es con coste=10, obteniendo 394 soportes de vectores, donde 195 pertenecen a una clase y 199 pertenecen a otra (dos clases).Pintamos gráfica ROCPLOT y calculamos errores de train y test:

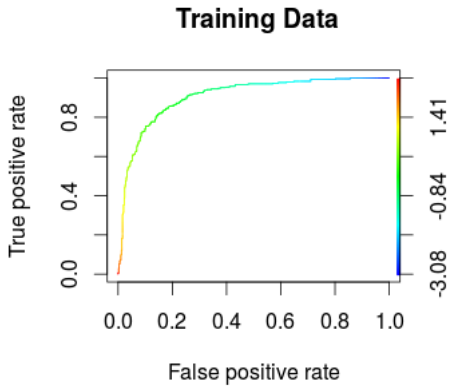

Figura 1.38: curva ROC para clasificador svm con kernel radial, para γ =0.001 y coste=10 sobre datos de train

```
> svmfitRad0.001best = svm( Purchase~. , data = train , kernel ="radial" ,gamma=0.001, cost =10 , decision.val
ues=T )
> fittedrad0.001 = attributes( predict(svmfitRad0.001best, train, decision.value=TRUE))$decision.values
> rocplot(fittedrad0.001 ,OJ[train0,"Purchase"], main=" Training Data" ,colorize=T)
> predict.trainRad0.001 = predict(svmfitRad0.001best,train)
> table(predict.trainRad0.001, train$Purchase)

predict.trainRad0.001  CH  MM
                   CH 436  72
                   MM  58 234
> (58+72)/(436+234+58+72)
[1] 0.1625
```

Figura 1.39: error de train para ajuste de clasificador svm con kernel radial, con γ =0.001 y coste=10

```
> predict.testRad0.001 = predict(svmfitRad0.001best,test)
> table(predict.testRad0.001,test$Purchase)

predict.testRad0.001  CH  MM
                   CH 140  30
                   MM  19  81
> (19+30)/(140+81+19+30)
[1] 0.1814815
```

Figura 1.40: error de test para ajuste de clasificador svm con kernel radial, con γ =0.001 y coste=10

Y vemos que para este ajuste del clasificador, obtenemos un error de train del 16.25 % y un error de test de 18.15 % .
Por último, le preguntamos a tune() que, para todos los posibles valores de γ y todos los posibles costes, cuál cree que es el mejor modelo de clasificación smv para un núcleo radial:

```
> set.seed(2)
> tune.outRad=tune(svm, Purchase~. , data =train, kernel ="radial" ,ranges=list(cost=c(0.001,0.01,0.1,1,10),gamma=c(10, 1, 0.
1, 0.01, 0.001)))
> bestmodRad=tune.outRad$best.model
> summary(bestmodRad)

Call:
best.tune(method = svm, train.x = Purchase ~ ., data = train, ranges = list(cost = c(0.001, 0.01, 0.1, 1,
    10), gamma = c(10, 1, 0.1, 0.01, 0.001)), kernel = "radial")


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  radial
       cost:  10
      gamma:  0.001

Number of Support Vectors:  394

 ( 195 199 )


Number of Classes:  2

Levels:
 CH MM
```

Figura 1.41: mejor valor de γ y coste para ajuste svm con núcleo radial

Conclusión: En función del valor al que inicialicemos la semilla, tendremos que el mejor modelo es uno de los siguientes:

1. **el de γ=0.01 y Coste 1 con error train=15.75 % y error test=17.41 %**
2. **el de γ=0.001 y Coste 10, con error train=16.25 % y error test=18.15 %**

El resto de modelos no son tan buenos. Revisemos porqué:

- γ=10, Coste=1 con error train=6.125 % y test=23.70 % no es óptimo porque gana mucho en train pero pierde mucho en test.
- γ=1, Coste=1 con error train=10.75 % y test=20.74 % no es óptimo por un motivo análogo al anterior.
- γ=0.1, Coste=1 con error train=14.25 % y test=16.66 % no es óptimo porque la diferencia entre error train y test es mayor que la diferencia de errores que se da en los modelos de arriba(los óptimos). Además, para este modelo tenemos 380 soportes de vectores, y dos clases, una con 195 elementos y otra con 185, por lo que tenemos un desequilibrio de elementos entre clases(no muy grande, pero notable) pues una tendrá 10 elementos más que la otra. Esto sin embargo no ocurre en los modelos óptimos, pues en el primero tenemos dos clases, cada una con 203 elementos, por lo que la división sería totalmente pura. Mientras que para el otro tenemos 4 elementos de diferencia entre sus dos clases.

1.6. Repetir apartados (2) a (4) usando un SVM con un núcleo polinómico. Usar degree con valores 2,3,4,5,6. Discutir los resultados

En este caso, para no extender demasiado el documento analizando para cada grado cada uno de los costes, vamos a analizar qué coste(de todos los posibles) es el mejor para cada uno de los posibles grados del polinomio, y posteriormente ajustaremos dicho modelo y nos centraremos en analizarlo.

1.6.1. SVM con núcleo polinómico y degree=2

Ajustamos, para grado 2, un modelo para cada uno de los costes y lo representamos:

```
> par(mfrow=c(1,1))
> set.seed(2)
> svmfitPOL0.001 = svm( Purchase~. , data = train , kernel ="polynomial" , cost =0.001 ,degree=2, decision.values=T )
> fittedPOL0.001 = attributes( predict(svmfitPOL0.001, train, decision.value=TRUE))$decision.values
> rocplot(fittedPOL0.001 ,OJ[train0,"Purchase"], main=" Training Data" ,col="blue2",add=par("new"))
> par(new=T)
> set.seed(2)
> svmfitPOL0.01 = svm( Purchase~. , data = train , kernel ="polynomial" , cost =0.01 ,degree=2, decision.values=T )
> fittedPOL0.01 = attributes( predict(svmfitPOL0.01, train, decision.value=TRUE))$decision.values
> rocplot(fittedPOL0.01 ,OJ[train0,"Purchase"], main=" Training Data" ,col="forestgreen",add=par("new"))
> par(new=T)
> set.seed(2)
> svmfitPOL0.1 = svm( Purchase~. , data = train , kernel ="polynomial" , cost =0.1 ,degree=2, decision.values=T )
> fittedPOL0.1 = attributes( predict(svmfitPOL0.1, train, decision.value=TRUE))$decision.values
> rocplot(fittedPOL0.1 ,OJ[train0,"Purchase"], main=" Training Data" ,col="darkorange1",add=par("new"))
> par(new=T)
> set.seed(2)
> svmfitPOL1 = svm( Purchase~. , data = train , kernel ="polynomial" , cost =1 ,degree=2, decision.values=T )
> fittedPOL1 = attributes( predict(svmfitPOL1, train, decision.value=TRUE))$decision.values
> rocplot(fittedPOL1 ,OJ[train0,"Purchase"], main=" Training Data" ,col="darkorchid1",add=par("new"))
> par(new=T)
> set.seed(2)
> svmfitPOL10 = svm( Purchase~. , data = train , kernel ="polynomial" , cost =10 ,degree=2, decision.values=T )
> fittedPOL10 = attributes( predict(svmfitPOL10, train, decision.value=TRUE))$decision.values
> rocplot(fittedPOL10 ,OJ[train0,"Purchase"], main=" Training Data" ,col="firebrick2",add=par("new"))
> par(new=T)
```

Figura 1.42: ajustes de svm con núcleo polinomial para degree=2 y cada uno de los costes

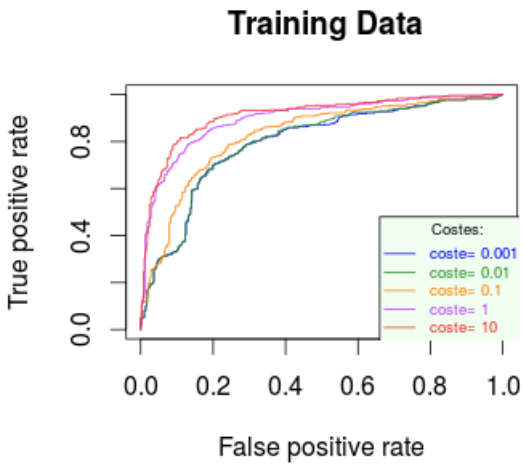

Figura 1.43: curvas ROC de los ajustes de svm con núcleo polinomial para degree=2 y cada uno de los costes

Y vemos que para degree=2, el mejor coste parece 10. Veamos que dice tune():

```
> set.seed(2)
> tune.outPol2=tune(svm, Purchase~. , data =train, kernel ="polynomial" ,ranges=list(cost=c(0.001,0.01,0.1
,1,10)),degree=2)
> bestmodPol2=tune.outPol2$best.model
> summary(bestmodPol2)

Call:
best.tune(method = svm, train.x = Purchase ~ ., data = train, ranges = list(cost = c(0.001,
    0.01, 0.1, 1, 10)), kernel = "polynomial", degree = 2)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  polynomial
       cost:  10
     degree:  2
      gamma:  0.05555556
     coef.0:  0

Number of Support Vectors:  342

 ( 170 172 )


Number of Classes:  2

Levels:
 CH MM
```

Figura 1.44: tune nos da el coste óptimo para degree=2

Y vemos que, efectivamente, tune() nos dice que el coste óptimo para degree=2 es 10. Para estos valores, tenemos un ajuste del casificador SMV con 342 soportes de vectores y dos clases, una con 170 y otra con 172.
Veamos cuáles son los errores de train y test para este ajuste con núcleo polinomial de coste=10 y grado2:

```
> svmfitPol2best = svm( Purchase~. , data = train , kernel ="polynomial" , cost =10 ,degree=2, decision.values=T )
> #error training:
> predict.trainPol2best = predict(svmfitPol2best,train)
> table(predict.trainPol2best, train$Purchase)

predict.trainPol2best  CH  MM
                   CH 450  72
                   MM  44 234
> #hacemos cálculo del error:
> (44+72)/(450+234+44+72)
[1] 0.145
```

Figura 1.45: error de train para ajuste clasificador SVM con núcleo polinomial, con grado=2 y coste=10

```
> predict.testPol2best = predict(svmfitPol2best,test)
> table(predict.testPol2best,test$Purchase)

predict.testPol2best  CH  MM
                   CH 140  31
                   MM  19  80
> (19+31)/(140+80+19+31)
[1] 0.1851852
```

Figura 1.46: error de test para ajuste clasificador SVM con núcleo polinomial, con grado=2 y coste=10

Obtenemos que, para este ajuste, tenemos un error de train de 14.5 % y para test tenemos un error de 18.52 %. Proseguimos a comprobar los resultados para otros valores del grado.

1.6.2. SVM con núcleo polinómico y degree=3

Ajustamos, para grado 3, un modelo para cada uno de los costes y lo representamos:

```
> set.seed(2)
> svmfitPOL0.001D3 = svm( Purchase~. , data = train , kernel ="polynomial" , cost =0.001 ,degree=3, decision.values=T )
> fittedPOL0.001D3 = attributes( predict(svmfitPOL0.001D3, train, decision.value=TRUE))$decision.values
> rocplot(fittedPOL0.001D3 ,OJ[train0,"Purchase"], main=" Training Data" ,col="green1",add=par("new"))
> par(new=T)
> set.seed(2)
> svmfitPOL0.01D3 = svm( Purchase~. , data = train , kernel ="polynomial" , cost =0.01 ,degree=3, decision.values=T )
> fittedPOL0.01D3 = attributes( predict(svmfitPOL0.01D3, train, decision.value=TRUE))$decision.values
> rocplot(fittedPOL0.01D3 ,OJ[train0,"Purchase"], main=" Training Data" ,col="mediumpurple",add=par("new"))
> par(new=T)
> #coste=0.1:
> set.seed(2)
> svmfitPOL0.1D3 = svm( Purchase~. , data = train , kernel ="polynomial" , cost =0.1 ,degree=3, decision.values=T )
> fittedPOL0.1D3 = attributes( predict(svmfitPOL0.1D3, train, decision.value=TRUE))$decision.values
> rocplot(fittedPOL0.1D3 ,OJ[train0,"Purchase"], main=" Training Data" ,col="red3",add=par("new"))
> par(new=T)
> #coste=1:
> set.seed(2)
> svmfitPOL1D3 = svm( Purchase~. , data = train , kernel ="polynomial" , cost =1 ,degree=3, decision.values=T )
> fittedPOL1D3 = attributes( predict(svmfitPOL1D3, train, decision.value=TRUE))$decision.values
> rocplot(fittedPOL1D3 ,OJ[train0,"Purchase"], main=" Training Data" ,col="yellow1",add=par("new"))
> par(new=T)
> #coste=10:
> set.seed(2)
> svmfitPOL10D3 = svm( Purchase~. , data = train , kernel ="polynomial" , cost =10 ,degree=3, decision.values=T )
> fittedPOL10D3 = attributes( predict(svmfitPOL10D3, train, decision.value=TRUE))$decision.values
> rocplot(fittedPOL10D3 ,OJ[train0,"Purchase"], main=" Training Data" ,col="yellowgreen",add=par("new"))
> par(new=T)
> legend(0.66, 0.48,paste("coste=",c(0.001,0.01,0.1,1,10)), col=c("green1","mediumpurple","red3","yellow1","yellowgreen"),
+        text.col=c("green1","mediumpurple","red3","yellow1","yellowgreen"), lty=1,
+        merge=TRUE, bg="honeydew", cex=0.6, title="Costes:" ,title.col="black")
```

Figura 1.47: ajustes de svm con núcleo polinomial para degree=3 y cada uno de los costes

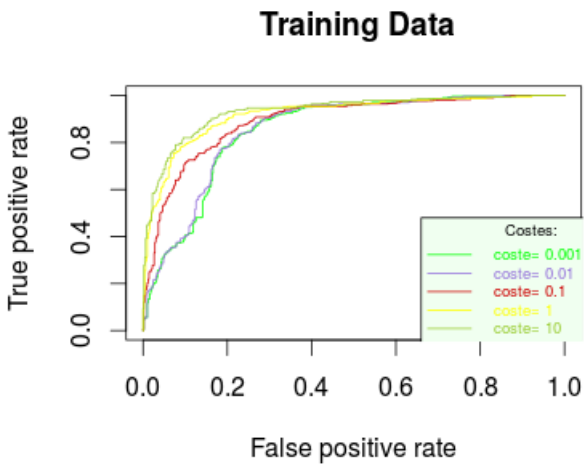

Figura 1.48: curvas ROC de los ajustes de svm con núcleo polinomial para degree=3 y cada uno de los costes

Y observamos en la gráfica que para degree=3, el mejor coste parece ser que va a ser el de 1, o el de 10. Veamos que dice tune():

```
> #mejor cost para degree=3:
> set.seed(2)
> tune.outPol3=tune(svm, Purchase~. , data =train, kernel ="polynomial" ,ranges=list(cost=c(0.001,0.01,0.1,1,10))
,degree=3)
> bestmodPol3=tune.outPol3$best.model
> summary(bestmodPol3)

Call:
best.tune(method = svm, train.x = Purchase ~ ., data = train, ranges = list(cost = c(0.001,
    0.01, 0.1, 1, 10)), kernel = "polynomial", degree = 3)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  polynomial
       cost:  1
     degree:  3
      gamma:  0.05555556
     coef.0:  0

Number of Support Vectors:  419

 ( 209 210 )


Number of Classes:  2

Levels:
 CH MM
```

Figura 1.49: tune nos da el coste óptimo para degree=3

Por su parte, tune() nos dice que el coste óptimo para degree=3 es 1 (línea amarilla en el gráfico). Para estos valores, tenemos un ajuste del casificador SMV con 419 soportes de vectores y dos clases, una con 209 y otra con 210.
Veamos cuáles son los errores de train y test para este ajuste con núcleo polinomial de coste 1 y grado 3:

```
> svmfitPol3best = svm( Purchase~. , data = train , kernel ="polynomial" , cost =1 ,degree=3, decision.values=T )
> #error training:
> predict.trainPol3best = predict(svmfitPol3best,train)
> table(predict.trainPol3best, train$Purchase)

predict.trainPol3best  CH  MM
                   CH 464  93
                   MM  30 213
> (30+93)/(464+213+30+93)
[1] 0.15375
```

Figura 1.50: error de train para ajuste clasificador SVM con núcleo polinomial, con grado=3 y coste=1

```
> predict.testPol3best = predict(svmfitPol3best,test)
> table(predict.testPol3best,test$Purchase)

predict.testPol3best  CH  MM
                  CH 146  41
                  MM  13  70
> (41+13)/(146+41+13+70)
[1] 0.2
```

Figura 1.51: error de test para ajuste clasificador SVM con núcleo polinomial, con grado=3 y coste=1

Obtenemos que, para este ajuste, tenemos un error de train de 15.37 % y para test tenemos un error de 20 %. Proseguimos a comprobar los resultados para otros valores del grado.

1.6.3. SVM con núcleo polinómico y degree=4

Ajustamos, para grado 4, un modelo para cada uno de los costes y lo representamos:

```
> set.seed(2)
> svmfitPOL0.001D4 = svm( Purchase~. , data = train , kernel ="polynomial" , cost =0.001 ,degree=4, decision.values=T )
> fittedPOL0.001D4 = attributes( predict(svmfitPOL0.001D4, train, decision.value=TRUE))$decision.values
> rocplot(fittedPOL0.001D4 ,OJ[train0,"Purchase"], main=" Training Data" ,col="chartreuse3",add=par("new"))
> par(new=T)
>
> #coste=0.01:
> set.seed(2)
> svmfitPOL0.01D4 = svm( Purchase~. , data = train , kernel ="polynomial" , cost =0.01 ,degree=4, decision.values=T)
> fittedPOL0.01D4 = attributes( predict(svmfitPOL0.01D4, train, decision.value=TRUE))$decision.values
> rocplot(fittedPOL0.01D4 ,OJ[train0,"Purchase"], main=" Training Data" ,col="chocolate2",add=par("new"))
> par(new=T)
>
> #coste=0.1:
> set.seed(2)
> svmfitPOL0.1D4 = svm( Purchase~. , data = train , kernel ="polynomial" , cost =0.1 ,degree=4, decision.values=T,scale=T )
> fittedPOL0.1D4 = attributes( predict(svmfitPOL0.1D4, train, decision.value=TRUE))$decision.values
> rocplot(fittedPOL0.1D4 ,OJ[train0,"Purchase"], main=" Training Data" ,col="firebrick2",add=par("new"))
> par(new=T)
>
> #coste=1:
> set.seed(2)
> svmfitPOL1D4 = svm( Purchase~. , data = train , kernel ="polynomial" , cost =1 ,degree=4, decision.values=T )
> fittedPOL1D4 = attributes( predict(svmfitPOL1D4, train, decision.value=TRUE))$decision.values
> rocplot(fittedPOL1D4 ,OJ[train0,"Purchase"], main=" Training Data" ,col="goldenrod1",add=par("new"))
> par(new=T)
>
> #coste=10:
> set.seed(2)
> svmfitPOL10D4 = svm( Purchase~. , data = train , kernel ="polynomial" , cost =10 ,degree=4, decision.values=T )
> fittedPOL10D4 = attributes( predict(svmfitPOL10D4, train, decision.value=TRUE))$decision.values
> rocplot(fittedPOL10D4 ,OJ[train0,"Purchase"], main=" Training Data" ,col="dodgerblue3",add=par("new"))
> par(new=T)
```

Figura 1.52: ajustes de svm con núcleo polinomial para degree=4 y cada uno de los costes

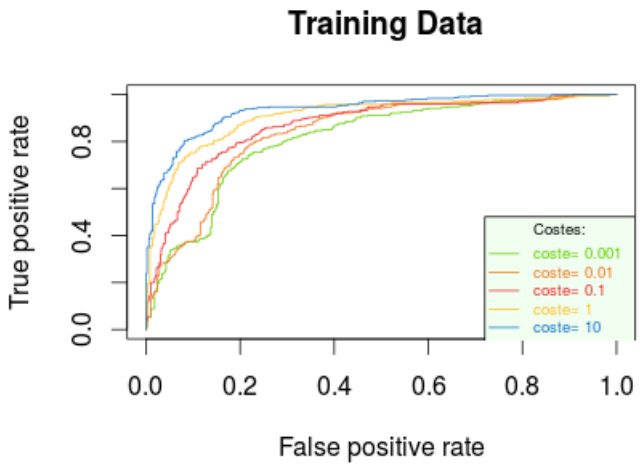

Figura 1.53: curvas ROC de los ajustes de svm con núcleo polinomial para degree=4 y cada uno de los costes

Y observamos en la gráfica que para degree=4, el mejor coste parece ser que va a ser el de 10. Veamos que dice tune():

```
> #mejor cost para degree=4:
> set.seed(2)
> tune.outPol4=tune(svm, Purchase~. , data =train, kernel ="polynomial" ,ranges=list(cost=c(0.001,0.01,0.1,1,10)),degree=4)
> bestmodPol4=tune.outPol4$best.model
> summary(bestmodPol4)

Call:
best.tune(method = svm, train.x = Purchase ~ ., data = train, ranges = list(cost = c(0.001,
    0.01, 0.1, 1, 10)), kernel = "polynomial", degree = 4)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  polynomial
       cost:  10
     degree:  4
      gamma:  0.05555556
     coef.0:  0

Number of Support Vectors:  370

 ( 178 192 )


Number of Classes:  2

Levels:
 CH MM
```

Figura 1.54: tune nos da el coste óptimo para degree=4

Por su parte, tune() nos dice que el coste óptimo para degree=4 es 10 (línea azul en el gráfico). Para estos valores, tenemos un ajuste del casificador SMV con 370 soportes de vectores y dos clases, una con 178 y otra con 192. Veamos cuáles son los errores de train y test para este ajuste con núcleo polinomial de coste 10 y grado 4:

```
> #ajustamos el modelo anterior que tune nos ha dicho ( degree=4, cost=10)
> svmfitPol4best = svm( Purchase~. , data = train , kernel ="polynomial" , cost =10 ,degree=4, decision.values=T )
> #error training:
> predict.trainPol4best = predict(svmfitPol4best,train)
> table(predict.trainPol4best, train$Purchase)

predict.trainPol4best  CH  MM
                   CH 462  80
                   MM  32 226
> (32+80)/(462+226+80+32)
[1] 0.14
```

Figura 1.55: error de train para ajuste clasificador SVM con núcleo polinomial, con grado=4 y coste=10

```
> predict.testPol4best = predict(svmfitPol4best,test)
> table(predict.testPol4best,test$Purchase)

predict.testPol4best  CH  MM
                   CH 144  37
                   MM  15  74
> (15+37)/(144+74+15+37)
[1] 0.1925926
```

Figura 1.56: error de test para ajuste clasificador SVM con núcleo polinomial, con grado=4 y coste=10

Obtenemos que, para este ajuste, tenemos un error de train de 14 % y para test tenemos un error de 19.26 %. Proseguimos a comprobar los resultados para otros valores del grado.

1.6.4. SVM con núcleo polinómico y degree=5

Ajustamos, para grado 5, un modelo para cada uno de los costes y lo representamos:

```
> set.seed(2)
> svmfitPOL0.001D5 = svm( Purchase~. , data = train , kernel ="polynomial" , cost =0.001 ,degree=5, decision.values=T )
> fittedPOL0.001D5 = attributes( predict(svmfitPOL0.001D5, train, decision.value=TRUE))$decision.values
> rocplot(fittedPOL0.001D5 ,OJ[train0,"Purchase"], main=" Training Data" ,col="blue2",add=par("new"))
> par(new=T)
> #coste=0.01:
> set.seed(2)
> svmfitPOL0.01D5 = svm( Purchase~. , data = train , kernel ="polynomial" , cost =0.01 ,degree=5, decision.values=T )
> fittedPOL0.01D5 = attributes( predict(svmfitPOL0.01D5, train, decision.value=TRUE))$decision.values
> rocplot(fittedPOL0.01D5 ,OJ[train0,"Purchase"], main=" Training Data" ,col="forestgreen",add=par("new"))
> par(new=T)
> #coste=0.1:
> set.seed(2)
> svmfitPOL0.1D5 = svm( Purchase~. , data = train , kernel ="polynomial" , cost =0.1 ,degree=5, decision.values=T )
> fittedPOL0.1D5 = attributes( predict(svmfitPOL0.1D5, train, decision.value=TRUE))$decision.values
> rocplot(fittedPOL0.1D5 ,OJ[train0,"Purchase"], main=" Training Data" ,col="darkorange1",add=par("new"))
> par(new=T)
> #coste=1:
> set.seed(2)
> svmfitPOL1D5 = svm( Purchase~. , data = train , kernel ="polynomial" , cost =1 ,degree=5, decision.values=T)
> fittedPOL1D5 = attributes( predict(svmfitPOL1D5, train, decision.value=TRUE))$decision.values
> rocplot(fittedPOL1D5 ,OJ[train0,"Purchase"], main=" Training Data" ,col="darkorchid1",add=par("new"))
> par(new=T)
> #coste=10:
> set.seed(2)
> svmfitPOL10D5 = svm( Purchase~. , data = train , kernel ="polynomial" , cost =10 ,degree=5, decision.values=T)
> fittedPOL10D5 = attributes( predict(svmfitPOL10D5, train, decision.value=TRUE))$decision.values
> rocplot(fittedPOL10D5 ,OJ[train0,"Purchase"], main=" Training Data" ,col="firebrick2",add=par("new"))
> par(new=T)
> legend(0.66, 0.48,paste("coste=",c(0.001,0.01,0.1,1,10)), col=c("blue2","forestgreen","darkorange1","darkorchid1","firebrick2"),
+        text.col=c("blue2","forestgreen","darkorange1","darkorchid1","firebrick2"), lty=1,
+        merge=TRUE, bg="honeydew", cex=0.6, title="Costes:" ,title.col="black")
```

Figura 1.57: ajustes de svm con núcleo polinomial para degree=5 y cada uno de los costes

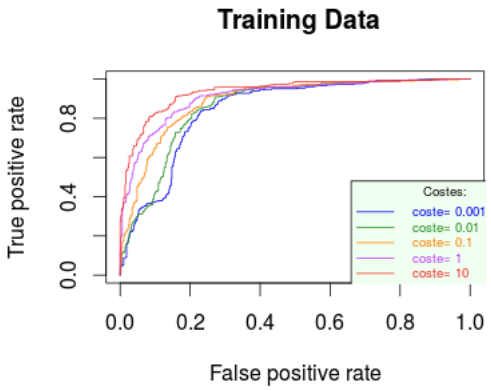

Figura 1.58: curvas ROC de los ajustes de svm con núcleo polinomial para degree=5 y cada uno de los costes

Y observamos en la gráfica que para degree=5, el mejor coste parece ser que va a ser el de 10. Veamos que dice tune():

```
> #mejor cost para degree=5:
> set.seed(2)
> tune.outPol5=tune(svm, Purchase~. , data =train, kernel ="polynomial" ,ranges=list(cost=c(0.001,0.01,0.1,1,10)),degree=5)
> bestmodPol5=tune.outPol5$best.model
> summary(bestmodPol5)

Call:
best.tune(method = svm, train.x = Purchase ~ ., data = train, ranges = list(cost = c(0.001,
    0.01, 0.1, 1, 10)), kernel = "polynomial", degree = 5)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  polynomial
       cost:  10
     degree:  5
      gamma:  0.05555556
     coef.0:  0

Number of Support Vectors:  380

 ( 184 196 )


Number of Classes:  2

Levels:
 CH MM
```

Figura 1.59: tune nos da el coste óptimo para degree=5

tune() nos dice que el coste óptimo para degree=5 es 10 (línea roja en el gráfico). Para estos valores, tenemos un ajuste del casificador SMV con 380 soportes de vectores y dos clases, una con 184 y otra con 196. Veamos cuáles son los errores de train y test para este ajuste con núcleo polinomial de coste 10 y grado 5:

```
> svmfitPol5best = svm( Purchase~. , data = train , kernel ="polynomial" , cost =10 ,degree=5, decision.values=T )
> #error training:
> predict.trainPol5best = predict(svmfitPol5best,train)
> table(predict.trainPol5best, train$Purchase)

predict.trainPol5best  CH  MM
                   CH 466  91
                   MM  28 215
> (28+91)/(466+215+28+91)
[1] 0.14875
```

Figura 1.60: error de train para ajuste clasificador SVM con núcleo polinomial, con grado=5 y coste=10

```
> #error test:
> predict.testPol5best = predict(svmfitPol5best,test)
> table(predict.testPol5best,test$Purchase)

predict.testPol5best  CH  MM
                  CH 608 134
                  MM  45 283
> (45+134)/(608+283+134+45)
[1] 0.1672897
```

Figura 1.61: error de test para ajuste clasificador SVM con núcleo polinomial, con grado=5 y coste=10

Obtenemos que, para este ajuste, tenemos un error de train de 14.87 % y para test tenemos un error de 22.22 %. Proseguimos a comprobar los resultados para grado 6.

1.6.5. SVM con núcleo polinómico y degree=6

Ajustamos, para grado 6, un modelo para cada uno de los costes y lo representamos:

```
> set.seed(2)
> svmfitPOL0.001D6 = svm( Purchase~. , data = train , kernel ="polynomial" , cost =0.001 ,degree=6, decision.values=T )
> fittedPOL0.001D6 = attributes( predict(svmfitPOL0.001D6, train, decision.value=TRUE))$decision.values
> rocplot(fittedPOL0.001D6 ,OJ[train0,"Purchase"], main=" Training Data" ,col="chartreuse3",add=par("new"))
> par(new=T)
>
> #coste=0.01:
> set.seed(2)
> svmfitPOL0.01D6 = svm( Purchase~. , data = train , kernel ="polynomial" , cost =0.01 ,degree=6, decision.values=T)
> fittedPOL0.01D6 = attributes( predict(svmfitPOL0.01D6, train, decision.value=TRUE))$decision.values
> rocplot(fittedPOL0.01D6 ,OJ[train0,"Purchase"], main=" Training Data" ,col="chocolate2",add=par("new"))
> par(new=T)
>
> #coste=0.1:
> set.seed(2)
> svmfitPOL0.1D6 = svm( Purchase~. , data = train , kernel ="polynomial" , cost =0.1 ,degree=6, decision.values=T)
> fittedPOL0.1D6 = attributes( predict(svmfitPOL0.1D6, train, decision.value=TRUE))$decision.values
> rocplot(fittedPOL0.1D6 ,OJ[train0,"Purchase"], main=" Training Data" ,col="firebrick2",add=par("new"))
> par(new=T)
>
> #coste=1:
> set.seed(2)
> svmfitPOL1D6 = svm( Purchase~. , data = train , kernel ="polynomial" , cost =1 ,degree=6, decision.values=T )
> fittedPOL1D6 = attributes( predict(svmfitPOL1D6, train, decision.value=TRUE))$decision.values
> rocplot(fittedPOL1D6 ,OJ[train0,"Purchase"], main=" Training Data" ,col="goldenrod1",add=par("new"))
> par(new=T)
>
> #coste=10:
> set.seed(2)
> svmfitPOL10D6 = svm( Purchase~. , data = train , kernel ="polynomial" , cost =10 ,degree=6, decision.values=T )
> fittedPOL10D6 = attributes( predict(svmfitPOL10D6, train, decision.value=TRUE))$decision.values
> rocplot(fittedPOL10D6 ,OJ[train0,"Purchase"], main=" Training Data" ,col="dodgerblue3",add=par("new"))
> par(new=T)
```

Figura 1.62: ajustes de svm con núcleo polinomial para degree=6 y cada uno de los costes

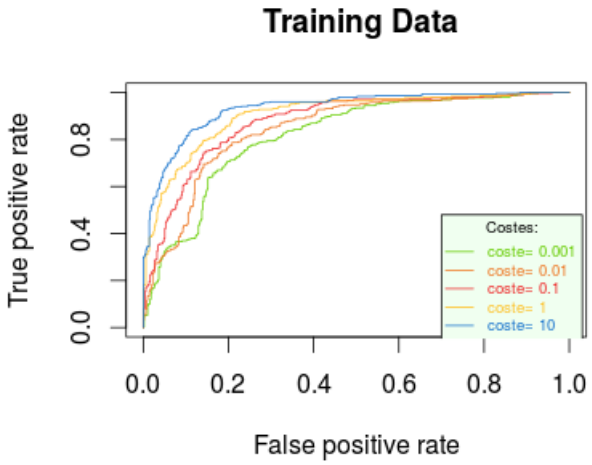

Figura 1.63: curvas ROC de los ajustes de svm con núcleo polinomial para degree=6 y cada uno de los costes

Y observamos en la gráfica que para degree=6, el mejor coste parece ser que va a ser el de 10(línea azul). Veamos si tune() dice lo mismo:

```
> set.seed(2)
> tune.outPol6=tune(svm, Purchase~. , data =train, kernel ="polynomial" ,ranges=list(cost=c(0.001,0.01,0
.1,1,10)),degree=6)
> bestmodPol6=tune.outPol6$best.model
> summary(bestmodPol6)

Call:
best.tune(method = svm, train.x = Purchase ~ ., data = train, ranges = list(cost = c(0.001,
    0.01, 0.1, 1, 10)), kernel = "polynomial", degree = 6)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  polynomial
       cost:  10
     degree:  6
      gamma:  0.05555556
     coef.0:  0

Number of Support Vectors:  410

 ( 198 212 )


Number of Classes:  2

Levels:
 CH MM
```

Figura 1.64: tune nos da el coste óptimo para degree=6

tune() nos dice que el coste óptimo para degree=6 es, efectivamente, 10. Para estos valores, tenemos un ajuste del casificador SMV con 410 soportes de vectores y dos clases, una con 198 y otra con 212. Veamos cuáles son los errores de train y test para este ajuste con núcleo polinomial de coste 10 y grado 6:

```
> svmfitPol6best = svm( Purchase~. , data = train , kernel ="polynomial" , cost =10 ,degree=6, decision.valu
es=T )
> predict.trainPol6best = predict(svmfitPol6best,train)
> table(predict.trainPol6best, train$Purchase)

predict.trainPol6best  CH  MM
                   CH 469 101
                   MM  25 205
> (25+101)/(469+205+25+101)
[1] 0.1575
```

Figura 1.65: error de train para ajuste clasificador SVM con núcleo polinomial, con grado=6 y coste=10

```
> predict.testPol6best = predict(svmfitPol6best,test)
> table(predict.testPol6best,test$Purchase)

predict.testPol6best  CH  MM
                  CH 141  46
                  MM  18  65
> (18+46)/(141+65+18+46)
[1] 0.237037
```

Figura 1.66: error de test para ajuste clasificador SVM con núcleo polinomial, con grado=6 y coste=10

Obtenemos que, para este ajuste, tenemos un error de train de 15.75 % y para test tenemos un error de 23.70 %.

Conclusión: Tenemos que el mejor ajuste de clasificador SVM con **núcleo Polinomial** es aquel con degree 2 y coste 10:

```
> #mejor cost para cualquier degree:
> set.seed(2)
> tune.outP=tune(svm, Purchase~. , data =train, kernel ="polynomial" ,ranges=list(cost=c(0.001,0.01,0.1,1,10
),degree=c(2,3,4,5,6)))
> bestmodPol=tune.outP$best.model
> summary(bestmodPol)

Call:
best.tune(method = svm, train.x = Purchase ~ ., data = train, ranges = list(cost = c(0.001,
    0.01, 0.1, 1, 10), degree = c(2, 3, 4, 5, 6)), kernel = "polynomial")


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  polynomial
       cost:  10
     degree:  2
      gamma:  0.05555556
     coef.0:  0

Number of Support Vectors:  342

 ( 170 172 )


Number of Classes:  2

Levels:
 CH MM
```

Figura 1.67: mejor ajuste de SVM con núcleo polinomial para el rango dado de valores

Que es el modelo que ajustaba el clasificador SVM con **342 soportes de vectores y dos clases, una con 170 y otra con 172**.

1.7. En global, ¿qué aproximación da el mejor resultado sobre estos datos?

Quedándonos con el mejor modelo para cada tipo de núcleo, tenemos:

1. **Lineal** con coste= 0.01: 432 SVM y dos clases (215,217). E.Train=16.625 % , E.Test= 18.15 %
2. **Radial** con γ=0.001 y coste=10: 394 SVM y dos clases(195,199). E.Train= 16.25 %, E.Test=18.15 %
3. **Polinomial** con grado=2 y coste=10: 342 SVM y dos clases(170,172). E.Train=14.5 %, E.Test=18.52 %

Como se puede observar, los tres modelos dan errores muy parecidos para test. En cuanto a train, tenemos que el modelo que menos error da es el polinómico, mientras que el radial y el lineal dan errores de train más parecidos. Además el polinómico ajusta 342 MSV que son menos de las que ajustan los otros modelos, lo cual implica una reducción de la dimensionalidad. Por tanto, parece que el que mejor resultado parece dar sobre, al menos, estos datos es el modelo con núcleo polinómico para grado=2 y coste=10.

2. Ejercicio.-2 (3 puntos) (comentar los resultados de todos los apartados) Usar el conjunto de datos OJ que es parte del paquete ISLR

2.1. Crear un conjunto de entrenamiento conteniendo una muestra aleatoria de 800 observaciones, y un conjunto de test conteniendo el resto de las observaciones. Ajustar un árbol a los datos de "training", usando "Purchase" como la respuesta y las otras variables excepto "Buy" como predictores.

```
> names(OJ)
 [1] "Purchase"       "WeekofPurchase" "StoreID"        "PriceCH"        "PriceMM"        "DiscCH"
 [7] "DiscMM"         "SpecialCH"      "SpecialMM"      "LoyalCH"        "SalePriceMM"    "SalePriceCH"
[13] "PriceDiff"      "Store7"         "PctDiscMM"      "PctDiscCH"      "ListPriceDiff"  "STORE"
```

Figura 2.1: variables del conjunto

Vemos que la variable "Buy"no existe en el conjunto. Ajustamos pues, usando Purchase como response y el resto de variables como predictors.

```
> set.seed(1)
> train0=sample(nrow(OJ),800)
> train=OJ[train0,]
> test=OJ[-train0]
>
> tree.OJ=tree(Purchase~. , OJ, subset=train0)
```

Figura 2.2: ajuste de árbol para datos training usando Purchase como response

2.2. Usar la función summary() para generar un resumen estadístico acerca del árbol y describir los resultados obtenidos: tasa de error de "training", número de nodos del árbol, etc. Teclee el nombre del objeto árbol y obtendrá una salida en texto. Elija un nodo e interprete su contenido.

```
> summary(tree.OJ)

Classification tree:
tree(formula = Purchase ~ ., data = OJ, subset = train0)
Variables actually used in tree construction:
[1] "LoyalCH"       "PriceDiff"     "SpecialCH"     "ListPriceDiff"
Number of terminal nodes:  8
Residual mean deviance:  0.7305 = 578.6 / 792
Misclassification error rate: 0.165 = 132 / 800
```

Figura 2.3: descripción gráfica del árbol ajustado

Observamos, haciendo un summary() del modelo, que tenemos que para la construcción del árbol se han utilizado las variables: "LoyalCH", "PriceDiff", "SpecialCH","ListPriceDiff", que además, este tiene 8 nodos terminales, que la media de la desviación residual es 0.7305 y que el error de trainin es: 0.165.

```
> tree.OJ
node), split, n, deviance, yval, (yprob)
      * denotes terminal node

 1) root 800 1064.00 CH ( 0.61750 0.38250 )
   2) LoyalCH < 0.508643 350  409.30 MM ( 0.27143 0.72857 )
     4) LoyalCH < 0.264232 166  122.10 MM ( 0.12048 0.87952 )
       8) LoyalCH < 0.0356415 57   10.07 MM ( 0.01754 0.98246 ) *
       9) LoyalCH > 0.0356415 109  100.90 MM ( 0.17431 0.82569 ) *
     5) LoyalCH > 0.264232 184  248.80 MM ( 0.40761 0.59239 )
      10) PriceDiff < 0.195 83   91.66 MM ( 0.24096 0.75904 )
        20) SpecialCH < 0.5 70   60.89 MM ( 0.15714 0.84286 ) *
        21) SpecialCH > 0.5 13   16.05 CH ( 0.69231 0.30769 ) *
      11) PriceDiff > 0.195 101  139.20 CH ( 0.54455 0.45545 ) *
   3) LoyalCH > 0.508643 450  318.10 CH ( 0.88667 0.11333 )
     6) LoyalCH < 0.764572 172  188.90 CH ( 0.76163 0.23837 )
      12) ListPriceDiff < 0.235 70   95.61 CH ( 0.57143 0.42857 ) *
      13) ListPriceDiff > 0.235 102   69.76 CH ( 0.89216 0.10784 ) *
     7) LoyalCH > 0.764572 278   86.14 CH ( 0.96403 0.03597 ) *
```

Figura 2.4: descripción gráfica del árbol ajustado

Tecleando el nombre del árbol, se nos muesta en modo texto. Vamos a ver cómo se interpreta un nodo. Por ejemplo, vamos a interpretar el nodo 20: tenemos que es un nodo terminal(nos lo indica la estrellita del final), además el criterio que sigue para clasificar los elementos es SpecialCH <0.5 (si SpecialCH <0.5, van a un lado, sino al otro). El número 70, nos indica el número de observaciones que hay en la rama, y con 60.89 nos está diciendo la desviación de las mismas con una predicción para todos estos, sobre MM. Por último, con (0.15714 0.84286) lo que nos dice es, que el 16 % de elementos van a la clase con etiqueta CH, mientras que los 84 % restantes van a la clase con etiqueta MM.

2.3. Crear un dibujo del árbol. Extraiga las reglas de clasificación más relevantes definidas por el árbol (al menos 4).

```
> plot(tree.OJ)
> text(tree.OJ,pretty=0)
```

Figura 2.5: comando plot para dibujar el árbol

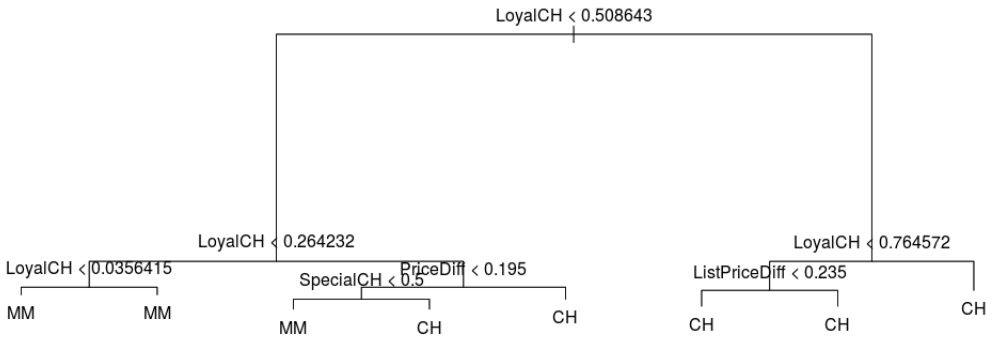

Figura 2.6: descripción gráfica del árbol ajustado

Y vemos que hemos ajustado un árbol que comienza dividiendo en función de si el valor de LoyalCH(la fidelidad del usuario con respecto a el zumo de la empresa Citrus Hill) queda por encima o por debajo de 0.508643. Luego, vuelve a subdividir para clasificar otra vez en función de la fidelidad del usuario con esta empresa; si su fidelidad inicial era menor que 0.508643 vuelve a dividir en función de si su fidelidad es menor que 0.264232, mientras que si su fidelidad era mayor que 0.508643, divide en función de si LoyalCH es menor que 0.764572 o no. Después vuelve a subdividir y así para cada nodo. Pero vemos que hay una serie de reglas de clasificación relevantes, y otras que no lo son tanto, veamos:
Reglas de clasificación relevantes:

- LoyalCH <0.508643 es muy relevante.
- LoyalCH <0.264232 también es importante.
- PriceDiff <0.195 también es importante, pues en función de ella podemos deducir que un usuario prefiera comprar Minute Maid(MM) o citrus Hill(CH).
- SpecialCH <0.5 también es importante, por un motivo análogo al anterior.

Reglas de clasificación poco o nada relevantes:

- LoyalCH <0.0356415 no es relevante, pues vemos que sea LoyalCH mayor o menor que este valor, el usuario preferirá comprar zumo Minute Maid(MM).
- ListPriceDiff <0.235 no es relavante, pues vemos que sea el valor de esta variable mayor o menor que 0.235, el usuario preferirá comprar zumo de Citrus Hill(CH).

- LoyalCH <0.764572 me atrevería a decir que tampoco es relavante, pues ya sea LoyalCH mayor o menor que quese valor, al final, acaba en CH. Por lo que para usuarios con LoyalCH>0.508643 tenemos que van a preferir directamente, el zumo de Citrus Hill(CH), y por lo tanto, hacer otras dos divisiones para decir lo mismo es algo de lo que se podría prescindir.

2.4. Predecir la respuesta de los datos de test, y generar e interpretar la matriz de confusión de los datos de test. ¿Cuál es la tasa de error del test? ¿Cuál es la precisión del test?

```
> tree.pred=predict(tree.OJ, test , type ="class" )
> table(tree.pred,test$Purchase)

tree.pred  CH  MM
       CH 147  49
       MM  12  62
> (12+49)/(147+49+12+62)
[1] 0.2259259
> 1-0.2259259
[1] 0.7740741
```

Figura 2.7: tasa de error de test y precisión de test

En la matriz de confusión vemos que hay 147 datos bien clasificados para CH y 62 datos bien clasificados para MM, sin embargo, tenemos 12+49=61 mal clasificados. Y vemos que tenemos que obtenemos un error de test del 22.59 % y por lo tanto, una precisión test del 1-0.2259=77.41 %

2.5. Aplicar la función cv.tree() al conjunto de "training" para determinar el tamaño óptimo del árbol.

```
> #óptimo diciéndole que tome el error de clasificación
> set.seed(1)
> cv.OJ=cv.tree(tree.OJ,FUN=prune.misclass)
> cv.OJ
$size
[1] 8 5 2 1

$dev
[1] 152 152 161 306

$k
[1]       -Inf   0.000000   4.666667 160.000000

$method
[1] "misclass"

attr(,"class")
[1] "prune"         "tree.sequence"
```

Figura 2.8: función cv.tree() para obtener el tamaño óptimo del árbol usando error de clasificación

Podemos ver que el menor error de validación cruzada viene dado por 152, que se corresponde con el árbol que tiene 8 nodos terminales y con el que tiene 5 nodos terminales también.

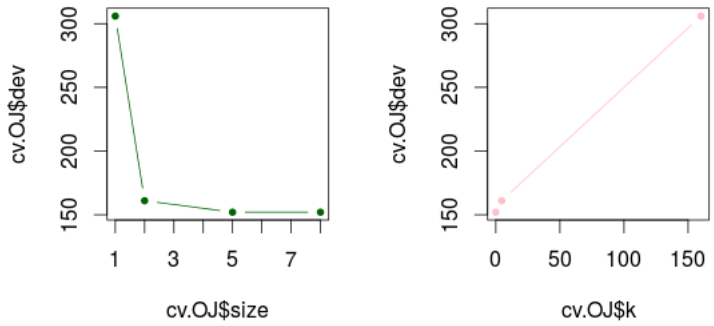

Figura 2.9: gráfica que representa el error frente a size y el error frente a k

2.6. Generar un gráfico con el tamaño del árbol en el eje x y la tasa de error de validación cruzada en el eje y. ¿Qué tamaño de árbol corresponde a la tasa más pequeña de error de clasificación por validación cruzada?

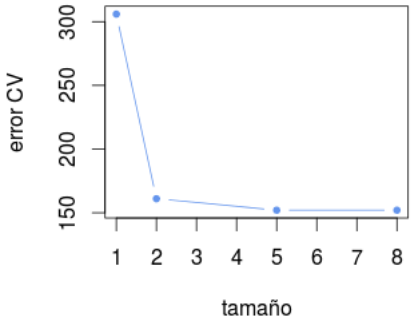

Figura 2.10: gráfica para determinar el tamaño óptimo del árbol

Y vemos en la gráfica que el coste óptimo está en 5, pues es donde el tamaño se corresponde con el mínimo error de cross-validation (aunque en el 8 se mantiene).

2.7. Ajustar el árbol podado correspondiente al valor óptimo obtenido en 6. Comparar los errores sobre el conjunto de training y test de los árboles ajustados en 6 con el árbol podado. ¿Cuál es mayor?

```
> prune.OJ= prune.misclass(tree.OJ, best =5)
> plot( prune.OJ)
> text( prune.OJ , pretty =0)
```

Figura 2.11: poda correspondiente al valor óptimo obtenido en el apartado6, que era 5

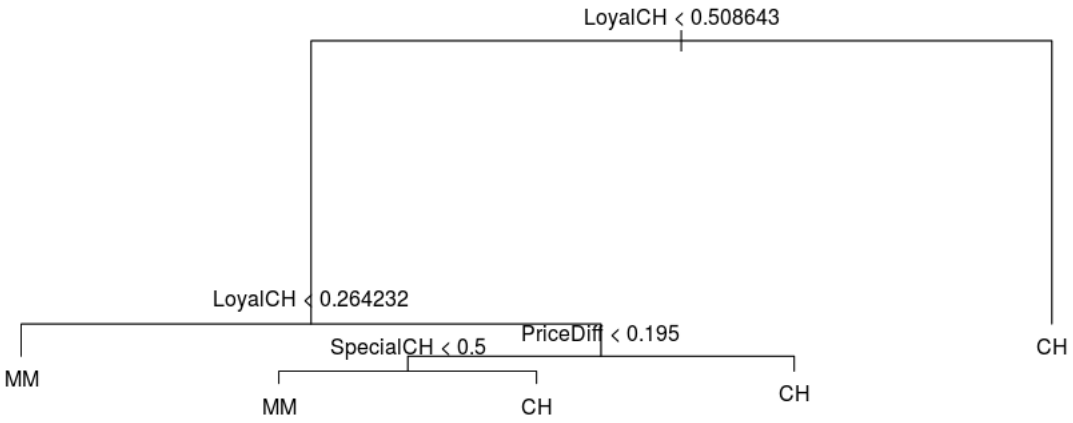

Figura 2.12: árbol con poda

Por último, calculamos los errores de training y test para este nuevo ajuste ajuste de árbol:

```
> summary(prune.OJ)

Classification tree:
snip.tree(tree = tree.OJ, nodes = 3:4)
Variables actually used in tree construction:
[1] "LoyalCH"   "PriceDiff" "SpecialCH"
Number of terminal nodes:  5
Residual mean deviance:  0.8256 = 656.4 / 795
Misclassification error rate: 0.165 = 132 / 800
```

Figura 2.13: summary() del ajuste del árbol podado

```
> tree.predPODA=predict(prune.OJ, test , type ="class" )
> table(tree.predPODA,test$Purchase)

tree.predPODA  CH  MM
           CH 147  49
           MM  12  62
> (12+49)/(147+62+12+49)
[1] 0.2259259
> 1-0.2259259
[1] 0.7740741
```

Figura 2.14: error test para ajuste del árbol podado

Observamos que para el ajuste con árbol podado obtenemos un error de training de 16.5 % y un error de test de 22.59 % , mientras que la precisión de test que obtenemos es de 77.41 % , que vemos que no mejora los errores que obtuvimos en el apartado 4 para el árbol sin poda, por lo tanto, ambos modelos son igualmente buenos clasificando, aunque uno es más óptimo que otro, pues para el árbol podado tenemos que se usan menos variables para construir el árbol de clasificación, que el número de nodos terminales es menor...

3. Ejercicio.-3 (3 puntos) (comentar los resultados de todos los apartados) Usar el conjunto de datos Hitters

3.1. Eliminar las observaciones para las que la información del salario es desconocido y aplicar una transformación logarítmica al resto de valores de salario. Crear un conjunto de "training" con 200 observaciones y un conjunto de "test" con el resto

```
> library(ISLR)
> attach(Hitters)
> names(Hitters)
 [1] "AtBat"     "Hits"      "HmRun"     "Runs"      "RBI"       "Walks"     "Years"     "CAtBat"    "CHits"     "CHmRun"

[11] "CRuns"     "CRBI"      "CWalks"    "League"    "Division"  "PutOuts"   "Assists"   "Errors"    "Salary"    "NewLeague
"
> sum(is.na(Hitters$Salary))
[1] 59
> Hitters = na.omit(Hitters)
> sum(is.na(Hitters))
[1] 0
```

Figura 3.1: eliminación de observaciones para las que el salario es desconocido

Con **is.na(...)** vemos que hay 59 observaciones para las que faltan información del salario. Con **na.omit(...)** eliminamos dichas obversaciones, viendo que ahora nos quedan 0 observaciones para las que faltan datos (todas las observaciones del conjunto están completas).

```
> #aplicamos transformación logaritmica al resto de valores del salario:
> Hitters$Salary=log(Hitters$Salary)
```

Figura 3.2: aplicación de tranfs.Logarítmica para el resto de observaciones del salario

Y ahora, ya estamos en situación de crear los conjuntos de training y test:

```
> set.seed(1)
> trainHit=sample(nrow(Hitters),200)
> train=Hitters[trainHit,]
> test=Hitters[-trainHit,]
> dim(train)
[1] 200  20
> dim(test)
[1] 63 20
```

Figura 3.3: conjunto de train con 200 observaciones y test con el resto (63 observaciones)

3.2. Realizar boosting sobre el conjunto de entrenamiento con 1,000 árboles para un rango de valores del parámetro de ponderación λ. Realizar un gráfico con el eje x mostrando diferentes valores de λ y los correspondientes valores de MSE de "training" sobre el eje y.

```
> library(gbm)
> set.seed(1)
> lambda=10^seq(-3, -0.2, by = 0.01)
> lambda.tam=length(lambda)
> mse=rep(NA,lambda.tam)
>
> for (i in (1:lambda.tam)) {
+   boosting.HITTERS=gbm(Salary ~ ., data =train, distribution = "gaussian", n.trees = 1000, shrinkage = lambda[i])
+   pred.trainHITTERS=predict(boosting.HITTERS, train, n.trees = 1000)
+   mse[i]=mean((pred.trainHITTERS - train$Salary)^2)
+ }
> plot(lambda, mse, type = "b", xlab = "lambda", ylab = " MSE para training", pch=20, col="darkred")
```

Figura 3.4: boosting para 281 valores de lambda tomados desde 0.001 a 0.63 de 0.01 en 0.01

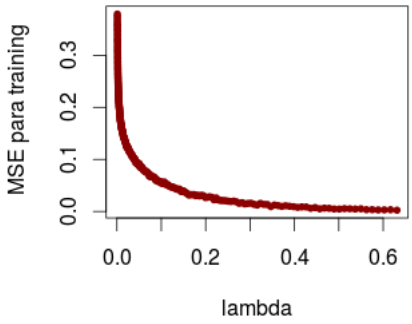

Figura 3.5: representación de los valores λ con su correspondiente MSE de training

Podemos ver como a partir de λ=0.6 , el MSE se va haciendo más pequeño.

3.3. Realizar el mismo gráfico del punto anterior pero usando los valores de MSE del conjunto de test. Comparar los valores de MSE obtenidos con boosting para el conjunto test con los obtenidos con los métodos de regresión múltiple y LASSO respectivamente para los mismos datos.

3.3.1. Boosting

```
> set.seed(1)
> lambda=10^seq(-3, -0.2, by = 0.01)
> lambda.tam=length(lambda)
> mse.test=rep(NA,lambda.tam)
> for (i in (1:lambda.tam)) {
+   boosting.HITTERS= gbm(Salary ~ ., data = train, distribution = "gaussian", n.trees = 1000, shrinkage = lambda[i])
+   pred.testHITTERS=predict(boosting.HITTERS, test, n.trees = 1000)
+   mse.test[i]=mean((pred.testHITTERS - test$Salary)^2)
+ }
> plot(lambda, mse.test, type = "b", xlab = "lambda", ylab = " MSE para test", pch=20, col="darkmagenta")
```

Figura 3.6: boosting para 281 valores de λ tomados desde 0.001 a 0.63 de 0.01 en 0.01

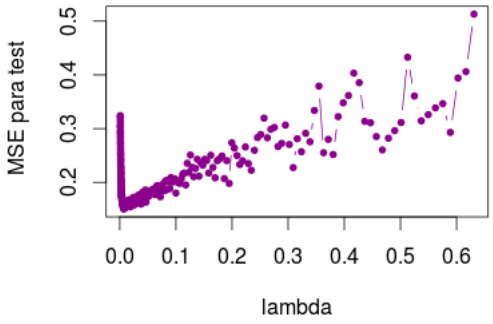

Figura 3.7: representación de los 281 valores λ con su correspondiente MSE de test

Comprobemos cuál es el valor esacto de λ para el que el mse de test es más chico:

```
> which.min(mse.test)
[1] 85
> lambda[85]
[1] 0.00691831
```

Figura 3.8: λ para el que obtemenos el menor MSE en test

y vemos que el menor MSE, que es 0.1502715(error test del 15.03 %) se obtiene para el elemento 85 del array de lambdas, que es λ=0.00691831.

```
> min(mse.test)
[1] 0.1502715
```

Figura 3.9: MSE mínimo para test obtenido con boosting

3.3.2. Regresión Múltiple

```
> lm.fit=lm(Salary~. , data=train)
> predict.lm=predict(lm.fit, test)
> mean((predict.lm - test$Salary)^2)
[1] 0.3938666
```

Figura 3.10: error de test para modelo de Regresión Lineal Múltiple

podemos observar que tenemos un error del 39.39 % para test con este modelo de Regresión Lineal Múltiple.

3.3.3. LASSO

```
> library(glmnet)
> lambda=10^seq(-3, -0.2, by = 0.01)
> x=model.matrix(Salary~., train)
> x.test=model.matrix(Salary~., test)
>
> lasso.mod=glmnet(x,train$Salary,alpha=1,lambda=lambda)
> plot(lasso.mod)
```

Figura 3.11: ajuste modelo Lasso para 281 posibles valores de λ entre 0.001 y 0.63

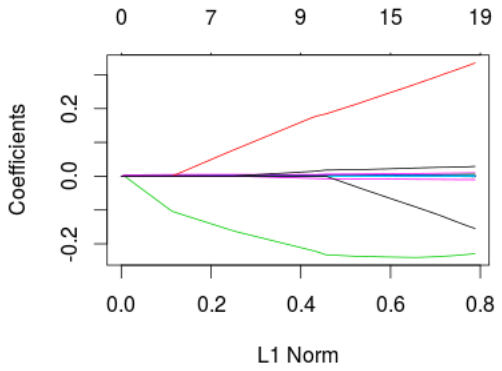

Figura 3.12: Imagen modelo Lasso para 281 posibles valores de λ entre 0.001 y 0.63

Como se ve en la gráfica, dependiento de la selección que se haga del $shrinkage$ algunos de los coeficientes pueden irse a cero. Aplicamos validación cruzada y calculamos el error de test:

```
> set.seed(1)
> cv.out=cv.glmnet(x,train$Salary,alpha=1)
> #plot(cv.out)
> bestlam=cv.out$lambda.min
> lasso.pred=predict(lasso.mod,s=bestlam, newx=x.test )
> mean((lasso.pred -test$Salary))
[1] 0.1574888
> bestlam
[1] 0.02349477
```

Figura 3.13: Imagen modelo Lasso para 281 posibles valores de λ entre 0.001 y 0.63

Y vemos que, para Lasso, tenemos que el error de test del modelo es del 15.75 % y que el mejor λ sería λ= 0.02349477.

Conclusión: Finalmente, tenemos que el modelo que tiene menor error test es en el que aplicamos Boosting, pues tenía un error test del 15.03 % frente al 39.39 % de error test que tenía el modelo de

regresión. Por otra parte, tenemos que lasso da resultados parecidos a Boosting (error test del 15.75 %) aunque inferiores.

3.4. ¿Qué variables aparecen como las más importantes en el modelo de "boosting"?

```
> #ajustamos modelo boosting para mejor lambda:
> boosting.best= gbm(Salary ~ ., data = train, distribution = "gaussian", n.trees = 1000,
shrinkage =lambda[which.min(mse.test)])
> summary(boosting.best)
                var    rel.inf
CAtBat       CAtBat 19.2060087
CHits         CHits 16.8497330
CRuns         CRuns 15.9371209
CRBI           CRBI 11.3275534
CWalks       CWalks  6.6234691
Walks         Walks  5.2393365
Years         Years  4.8829277
Hits           Hits  4.8246049
CHmRun       CHmRun  3.8717610
PutOuts     PutOuts  2.5287646
HmRun         HmRun  2.2166368
RBI             RBI  1.5766446
Errors       Errors  1.5364209
Division   Division  1.4915737
AtBat         AtBat  0.6878842
League       League  0.5979105
Runs           Runs  0.2400909
NewLeague NewLeague  0.1826678
Assists     Assists  0.1788909
```

Figura 3.14: summary del modelo de Boosting

Vemos que las variable que parece ser las más importantes, con diferencia, son CAtBat y CHits, donde la primera es,de las dos, la más relevante. Luego hay otras relevantes, como CRuns, CRBI o CWalks. Además, summary() nos muestra un gráfico de la influencia relativa de las variables:

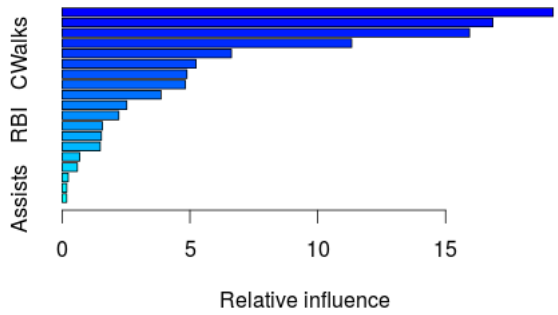

Figura 3.15: gráfico de la influencia relativa de las varibles

3.5. Aplicar bagging al conjunto de "training" y volver a estimar el modelo. ¿Cuál es el valor de MSE para el conjunto de test en este caso?

```
> set.seed(1)
> bag.HITTERS = randomForest( Salary~. , data =train, mtry =19 , importance = TRUE )
> yhat.bag = predict( bag.HITTERS , newdata = test)
> plot(yhat.bag,test$Salary, pch=20, col="olivedrab2")
> abline(0,1,col="black")
> mean((yhat.bag - test$Salary)^2)
[1] 0.1198233
```

Figura 3.16: aplicación de bagging al conjunto de training y predicción

Y vemos que aplicando bagging sobre los datos de traning, hemos obtenido un MSE para test del 11.98 %.

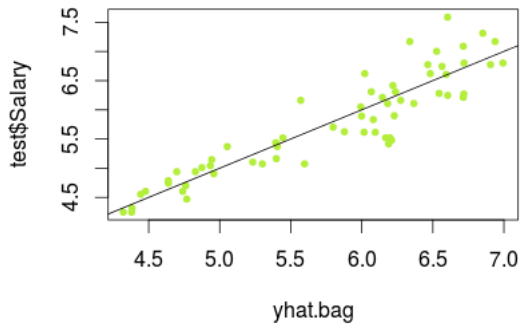

Figura 3.17: ajuste bagging sobre datos training