Minería de datos: preprocesamiento y clasificación

Sesión 3

Curso 2017-18

Índice

Índice

Imputación de valores perdidos con paquete mice

Una vez conocido el patrón de datos perdidos, podemos proceder a su imputación. En el script siguiente veremos funciones para:

- conocer el número de instancias con datos perdidos
- conocer el número de instancias sin datos perdidos
- realizar la imputación con diferentes métodos
- completar los datos con los resultados de la imputación
- mostrar información sobre la forma en que se realizó la imputación

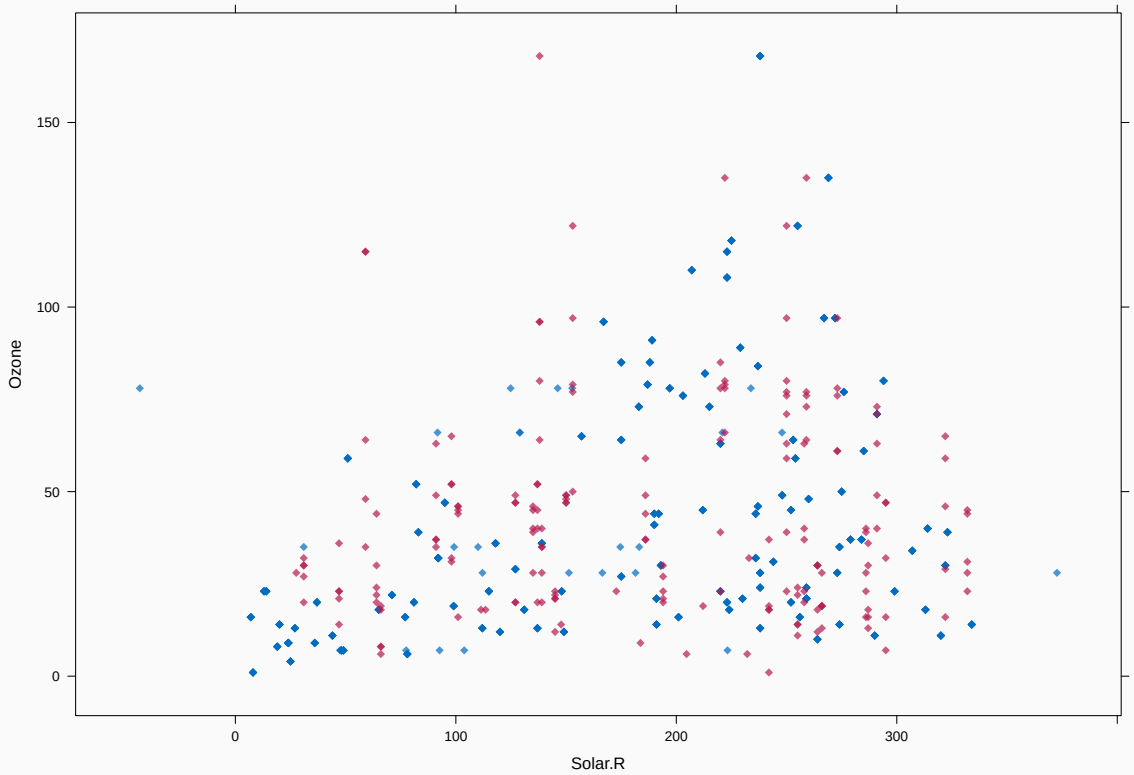
Script imputacionMice.R I

```r
# Este paquete puede usarse para imputar valores perdidos en
# variables de todo tipo
library(mice)
library(lattice)

# se usa el conjunto airquality
datos <- airquality

# se determina el numero de instancias sin datos perdidos y con datos
# perdidos. A observar la comodidad de uso de las funciones ncc e nic
completos <- mice::ncc(datos)
incompletos <- mice::nic(datos)
cat("Datos completos: ",completos, " e incompletos: ",incompletos,"\n")

# se realiza la imputacion
imputados <- mice::mice(datos, m=5, meth="pmm")

# dispone de algunos metodos que imputan siempre a un unico
# valor, como por ejemplo "mean"
imputadosMean <- mice::mice(datos, m=1, meth="mean")

# pmm es el metodo por defecto. Puedes verse todos los metodos
# disponibles de la siguiente forma
methods(mice)

# se completa el conjunto de datos con las imputaciones
datosImputados <- mice::complete(imputados)

# se determina el numero de instancias sin datos perdidos y con datos
```

Script imputacionMice.R II

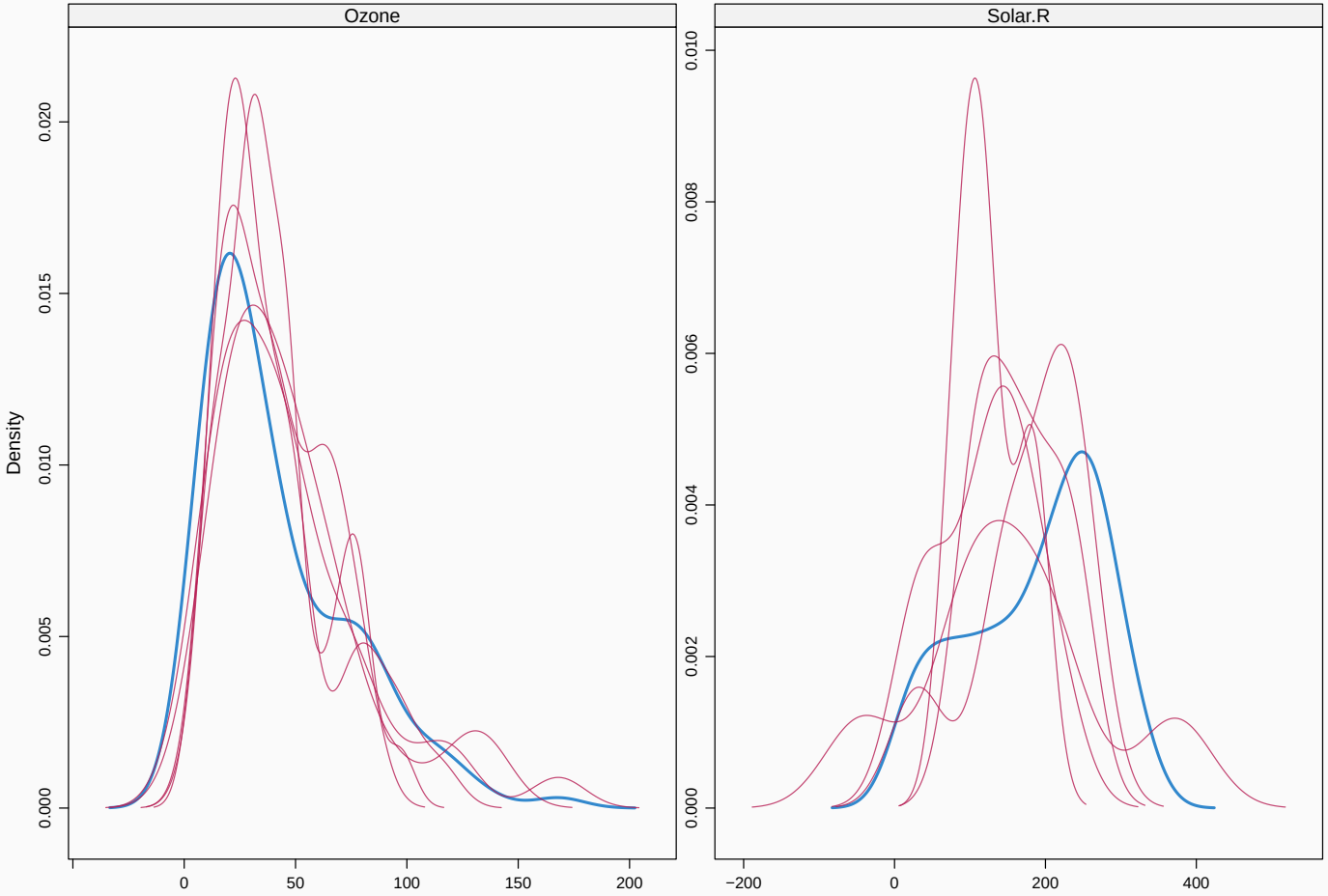
```
30 # perdidos en la parte ya limpia
31 completos <- mice::ncc(datosImputados)
32 incompletos <- mice::nic(datosImputados)
33 cat("Datos completos: ",completos, " e incompletos: ",incompletos,"\n")
34
35 # se muestra la imputacion para Ozone
36 imputados$imp$Ozone
37
38 # Se muestra un grafico para comprobar la distribucion de Ozone en los
39 # datos imputados en relacion a otras variables. Los puntos en azul
40 # repreentan datos observados y datos en rojo representan imputaciones
41 lattice::xyplot(imputados,Ozone ~ Solar.R,pch=18,cex=1)
42
43 # Se muestran las densidades de los datos imputados respecto de los
44 # observados
45 lattice::densityplot(imputados)
46
47 # Se muestran los diagramas de caja para las imputaciones
48 lattice::bwplot(imputados)
```

Script imputacionMice.R I

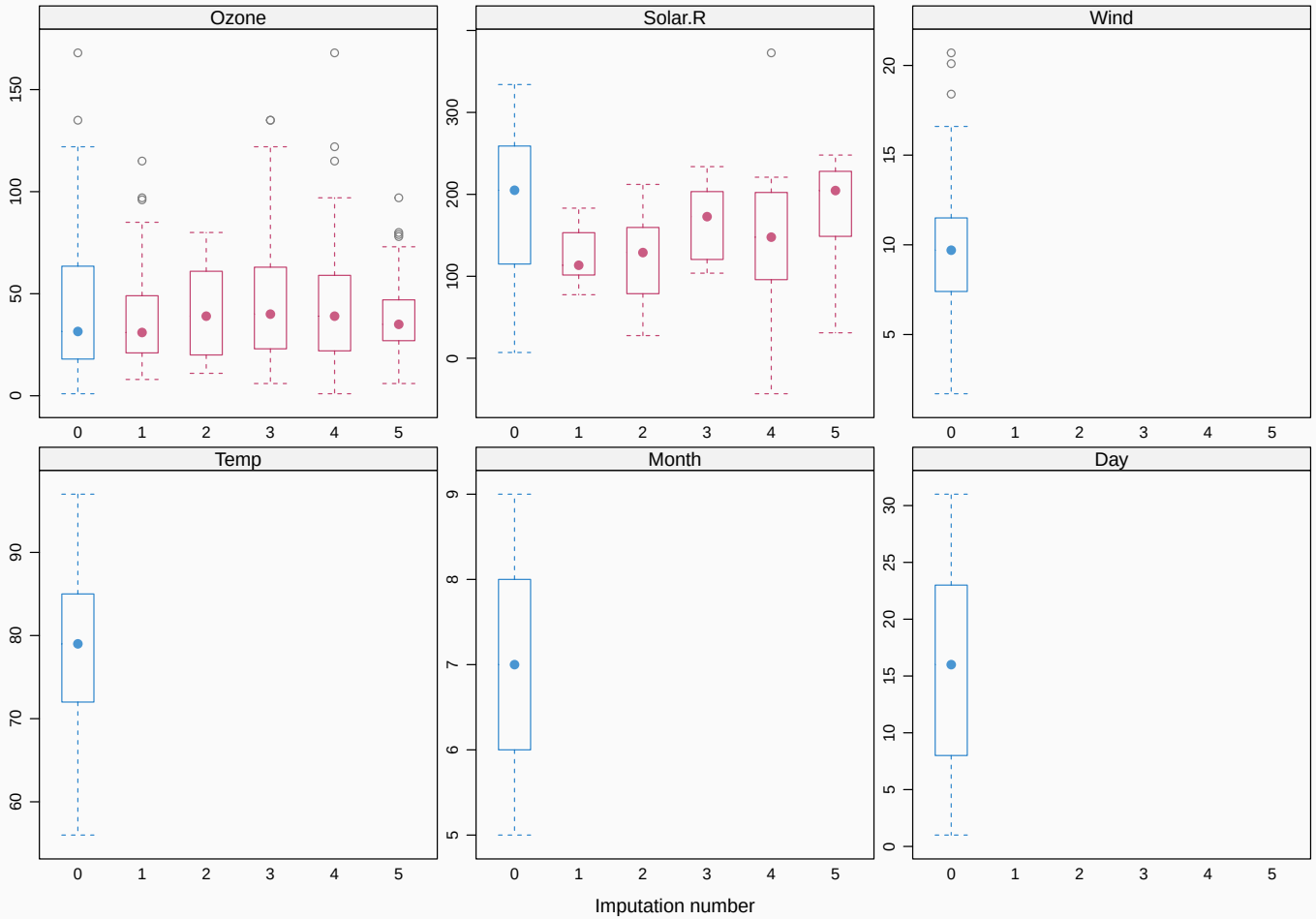


Puntos en rojo: imputaciones; puntos en azul: datos observados.

Script imputacionMice.R I



Script imputacionMice.R I

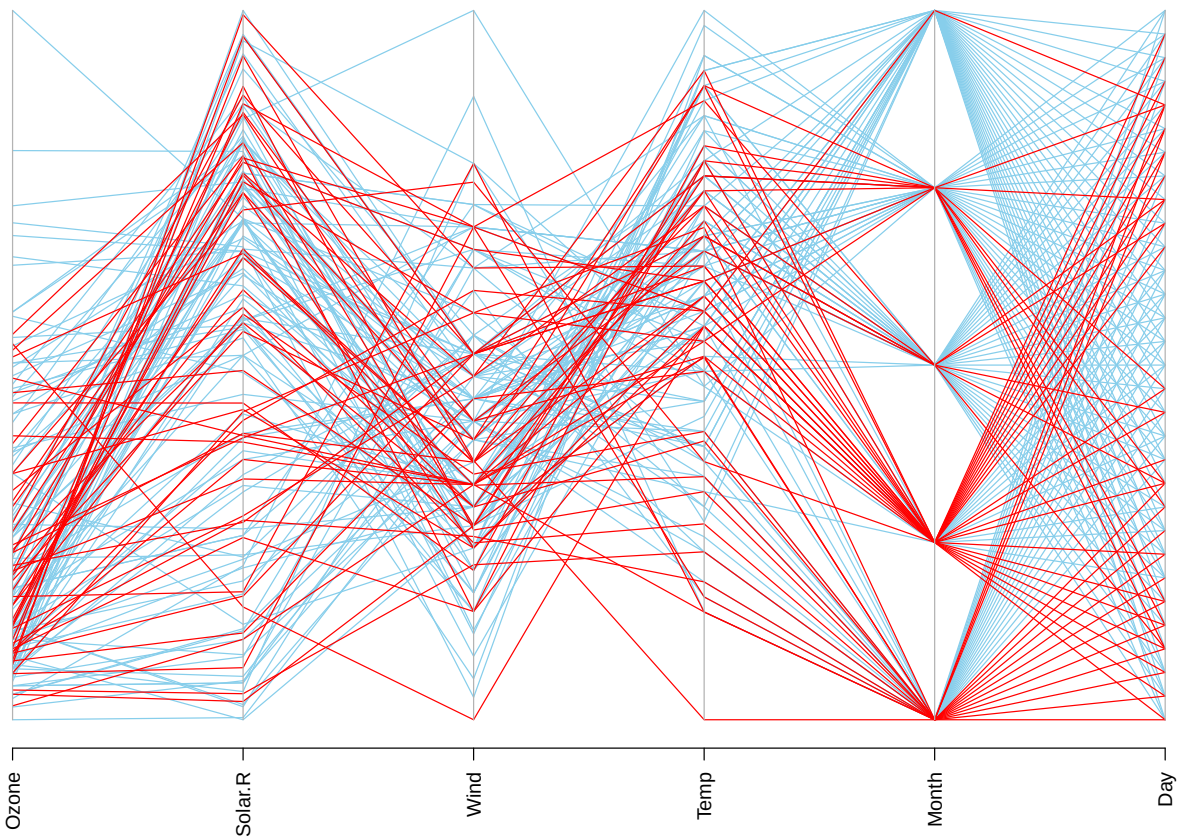


Índice

Script imputacionRobCompisitions.R I

```r
require(robCompositions)
require(mice)

# se usa el conjunto de datos de calidad del aire, en las
# mismas condiciones que vimos con anterioridad
datos <- airquality

# se determina el numero de instancias sin datos perdidos y con datos
# perdidos. A observar la comodidad de uso de las funciones ncc e nic
completos <- mice::ncc(datos)
incompletos <- mice::nic(datos)
cat("Datos completos: ",completos, " e incompletos: ",incompletos,"\n")

# se hace la imputacion
imputados <- robCompositions::impKNNa(datos)

# Ahora puede visualizarse alguna informacion sobre la forma
# en que se hizo la imputacion. El segundo argumento indica el
# tipo de grafico a obtener
plot(imputados, which=2)

# El conjunto de datos completo puede accederse de la siguiente forma
imputados$xImp
```

Script imputacionRobCompositions.R I



Índice

Índice

Script anomalias.R I

```r
library(outliers)
library(ggplot2)

# se carga el archivo con las funcione de lectura de datos
source("lecturaDatos.R")

path <- "./data/"
file <- "datos.csv"

# lectura de los datos
datos <- lecturaDatos(path,file)

# deteccion de anomalias para las variable 1 a 3. Observad
# que no tiene sentido considerar variables de tipo discreto
# en este analisis. La funcion devuelve el valor (o valores)
# considerados anomalos para las variable de interes. Este
# metodo solo considera las desviaciones con respecto a los
# valores de cada variable (no relaciones con otras variables)
anomalos <- outlier(datos[,1:3])
print(anomalos)

# la media de la variable separation es
mean(datos[,"separation"])

# se muestra la distribucion de separation en funcion del valor
# de la variable clase
ggplot(data = datos, aes(class, separation)) +
  geom_boxplot()

```

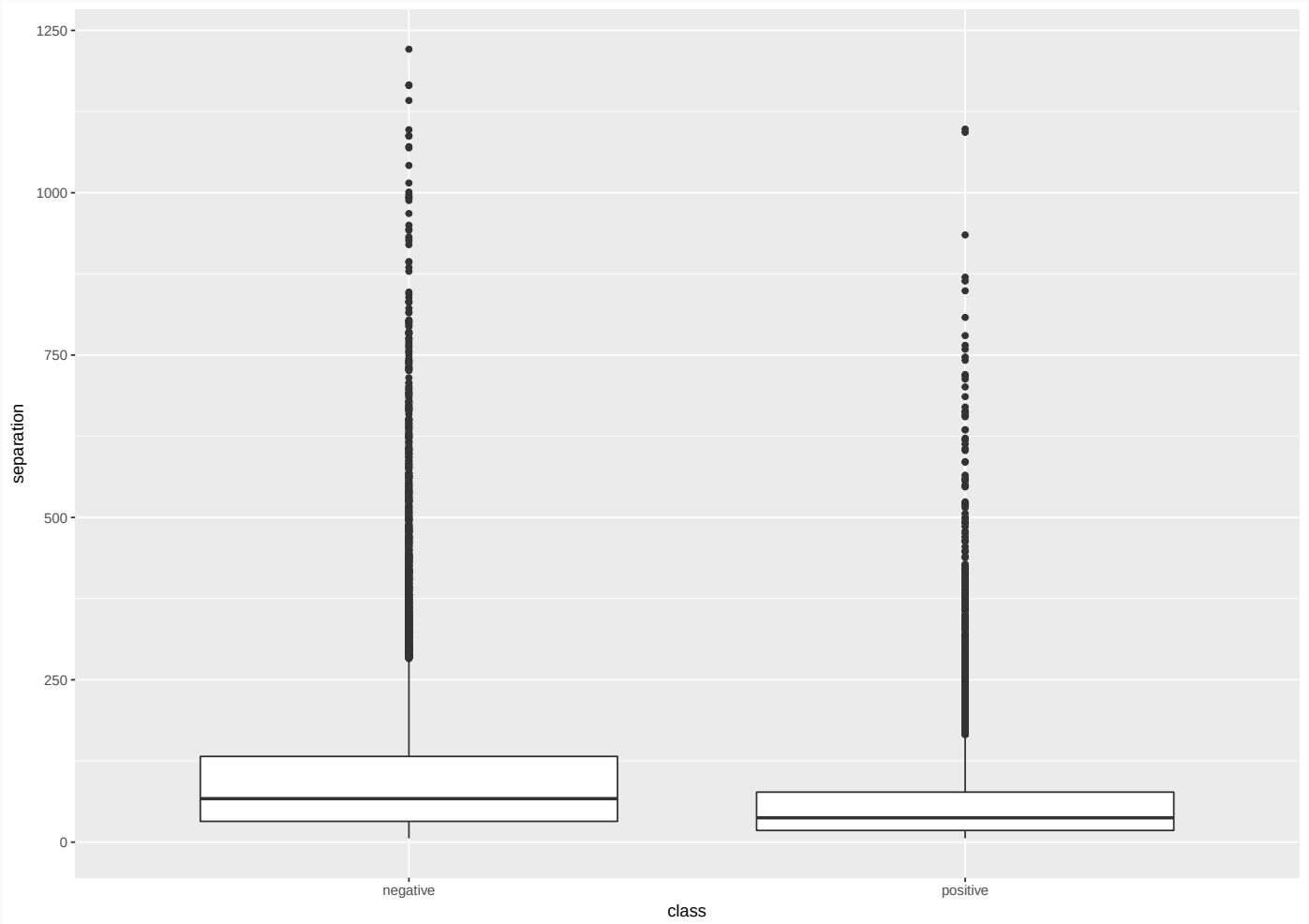
Script anomalias.R II

```
30 # se podria hacer igual con la variable propensity
31 ggplot(data = datos, aes(class, propensity)) +
32   geom_boxplot()
33
34
```

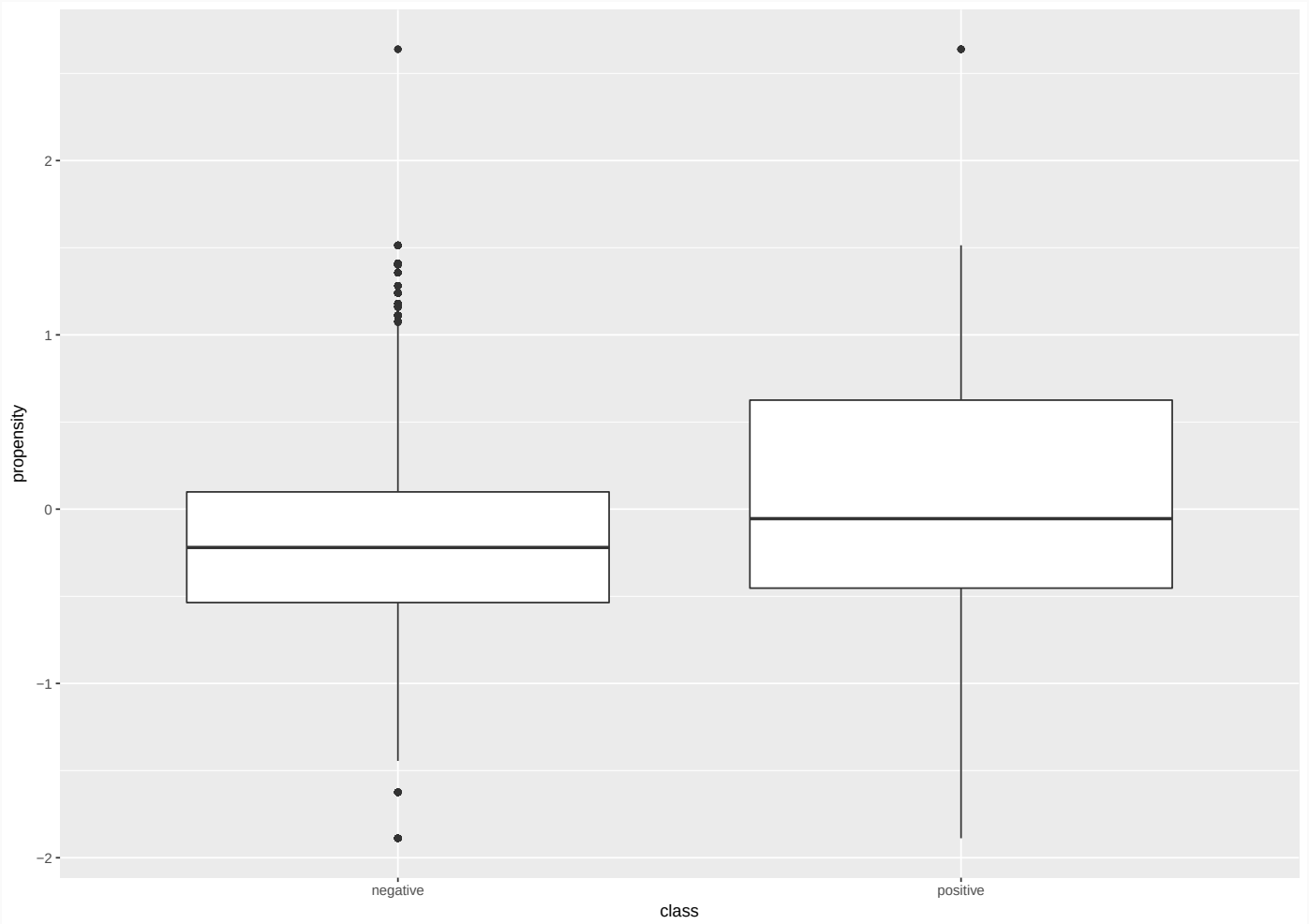
Sólo para variables numéricas.....

```
 separation  propensity      length
1221.000000    2.638864 1244.000000
```

Script anomalias.R I



Script anomalias.R I



Índice

Script anomaliasMvoutliers.R I

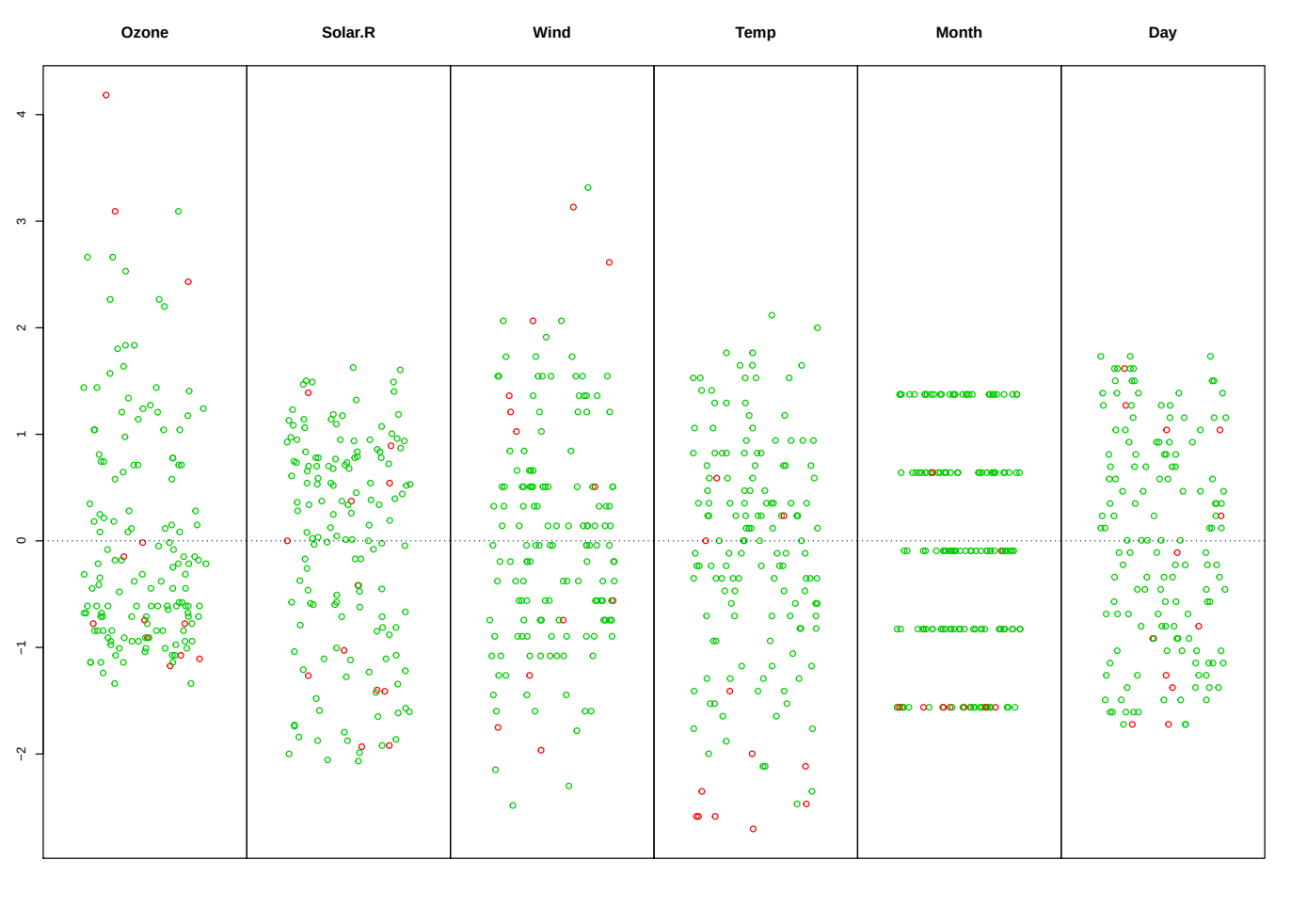
```r
require(mice)
require(mvoutlier)

# se usa el conjunto de datos de calidad del aire, en las
# mismas condiciones que vimos con anterioridad
datos <- airquality

# se determina el numero de instancias sin datos perdidos y con datos
# perdidos. A observar la comodidad de uso de las funciones ncc e nic
completos <- mice::ncc(datos)
incompletos <- mice::nic(datos)
cat("Datos completos: ",completos, " e incompletos: ",incompletos,"\n")

# se imputan los datos
imputados <- mice::mice(datos)
datos <- mice::complete(imputados)

# se analizan los datos en busca de anomalias. El grafico
# resultante muestra en rojo los datos considerados considerados
# como anomalos
resultados <- mvoutlier::uni.plot(datos)

# a partir de resultado es posible conocer las instancias en que
# aparece algun dato anomalo. Esto podria usarse para filtrar las
# instancias y quedarnos con aquellas en que no haya anomalias (o
# bien aplicar alguna tecnica para modificar sus valores)
print(resultados$outliers)

```

Script anomaliasMvoutliers.R II

```
29 # seleccion de instancias sin anomalias
30 datosFinales <- datos[!resultados$outliers, ]
```

```
 [1]  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE
[20]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[39] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE
[58] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[77] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[96] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[115] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[134] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
[153] FALSE
```

Script anomaliasMvoutliers.R I



Índice

Script transformacion.R I

Especialmente necesario en situaciones en que haya que considerar distancias entre instancias.... Más usuales: centrado y escalado.

```
require(mice)
require(caret)

# se usa el conjunto de datos de calidad del aire, en las
# mismas condiciones que vimos con anterioridad
datos <- airquality

# se determina el numero de instancias sin datos perdidos y con datos
# perdidos. A observar la comodidad de uso de las funciones ncc e nic
completos <- ncc(datos)
incompletos <- nic(datos)
cat("Datos completos: ",completos, " e incompletos: ",incompletos,"\n")

# se imputan los datos
imputados <- mice(datos)
datos <- complete(imputados)

# se aplica el centrado y escalado sobre el conjunto de datos,
# una vez eliminados los valores perdidos
valoresPreprocesados <- caret::preProcess(datos[,1:4],method=c("center","scale"))

# el resultado consiste en el escalado y centrado de las variables
# de la 1 a la 4 (las que pueden considerarse continuas). El resultado
# anterior se usa ahora para asignar a las variables los valores
```

Script transformacion.R II

```
25 # correspondientes de acuerdo a esta transformacion
26 valoresTransformados <- predict(valoresPreprocesados,datos[,1:4])
27
28 # y podemos generar un nuevo conjunto de datos con el que
29 # seguir aplicando tecnicas con las 4 variables transformadas
30 # y las dos que no se tocaron
31 datosFinales <- cbind(valoresTransformados,datos[,5:6])
```

Índice

Índice

Script transformacion.R I

Algoritmos disponibles:

- **ameva**: test **chi-cuadrado** para conseguir discretización óptima con mínimo número de intervalos y con mínima pérdida de información con respecto a la variable clase
- **caim**: considerando la dependencia con la variable clase
- **cacc**: considerando tabla de contingencias entre variable clase y variable a discretizar

```
require(discretization)

# se usa el conjunto de datos de calidad del aire, en las
# mismas condiciones que vimos con anterioridad
datos <- iris

# discretizacion mediante metodo CAIM
cm <- discretization::disc.Topdown(iris, method=1)

# se muestran los puntos de corte
cat("Puntos de corte metodo CAIM: \n")
print(cm$cutp)
```

Script transformacion.R II

```
13
14 # los datos discretizados se mostrarian de la
15 # forma siguiente
16 cat("Datos discretizados: \n")
17 print(cm$Disc.data)
18
19 # discretizacion mediante CACC
20 cmCacc <- disc.Topdown(datos, method=2)
21
22 # se muestran los puntos de corte
23 cat("Puntos de corte metodo CACC: \n")
24 print(cm$cutp)
25
26 # discretizacion mediante AMEVA
27 cmAmeva <- disc.Topdown(datos, method=3)
28
29 # se muestran los puntos de corte
30 cat("Puntos de corte metodo AMEVA: \n")
31 print(cm$cutp)
```

Script discretizacion.R I

Puntos de corte con método **CAIM**:

```
[[1]]
[1] 4.30 5.55 6.25 7.90

[[2]]
[1] 2.00 2.95 3.05 4.40

[[3]]
[1] 1.00 2.45 4.75 6.90

[[4]]
[1] 0.10 0.80 1.75 2.50
```

Script discretizacion.R I

Datos discretizados: $print(cm\$Disc.data)$

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	1	3	1	1	setosa
2	1	2	1	1	setosa
3	1	3	1	1	setosa
4	1	3	1	1	setosa
5	1	3	1	1	setosa
6	1	3	1	1	setosa

.....

Índice

Selección de características I

Aproximaciones:

- **filter**: se ofrece una ordenación de las variables de acuerdo a su importancia (medida de alguna forma, sin necesidad de construir modelos)
- **wrapper**: se crean modelos para determinar la importancia de las variables
- **embedded**: importancia de las variables determinada por la construcción del modelo

Índice

Paquete FSelector I

Aproximaciones filter:

- **chi-square**
- **correlation**
- **entropy based**: medidas de información
- **one R**
- **relief**: distancia entre instancias

Script fSelector-entropy.R I

```r
library(FSelector)
data(iris)

# se obtienen las medidas mediante ganancia de informacion
weights <- FSelector::information.gain(Species~., iris)

# se muestran los pesos y se seleccionan los mejores
print(weights)
subset <- cutoff.k(weights,2)
f <- as.simple.formula(subset,"Species")
print(f)

# igual, pero con ganancia de informacion
weights <- gain.ratio(Species~., iris)
print(weights)

# e igual con symmetrical.uncertainty
weights <- symmetrical.uncertainty(Species~., iris)
print(weights)
```

Script fSelector-entropy.R I

```
              attr_importance
Sepal.Length        0.4521286
Sepal.Width         0.2672750
Petal.Length        0.9402853
Petal.Width         0.9554360

Species ~ Petal.Width + Petal.Length
```

Paquete FSelector I

Aproximaciones wrapper:

- **best first search**
- **exhaustive search**
- **greedy search**: medidas de información
- **hill climbing**
- **cfs**: hace uso de best first search
- **consistency**: hace uso de best first search

Script fSelector-bestFirstSearch.R I

Esquema de validación cruzada con función de evaluación:

```
1  library(rpart)
2  library(FSelector)
3  data(iris)
4
5  # Se define una funcion de evaluacion: recibe como argumento un
6  # vector de atributos a evaluar
7  evaluator <- function(subset){
8    # se indica el numero de particiones a realizar en el proceso
9    # de validacion cruzada
10   k <- 10
11
12   # genera valores aleatorios (uniforme) para cada muestra del
13   # conjunto de datos
14   splits <- runif(nrow(iris))
15
16   # tratamiento de cada una de las particiones. Para cada valor de
17   # particion se aplica la funcion que se define a continuacion
18   results <- sapply(1:k, function(i) {
19     # se determina el indice de las muestras para test (aproximadamente
20     # una fraccion 1/k de las muestras del conjunto de datos)
21     test.idx <- (splits >= ((i-1)/k) & (splits < (i/k)))
22
23     # todas las demas muestras seran para training
24     train.idx <- !test.idx
25
26     # se seleccionan las muestras en si
```

Script fSelector-bestFirstSearch.R II

```
27      test <- iris[test.idx, ,drop=FALSE]
28      train <- iris[train.idx, , drop=FALSE]
29
30      # aprende el modelo sobre el conjunto de entrenamiento
31      tree <- rpart::rpart(as.simple.formula(subset,"Species"),train)
32
33      # calcula la tasa de error
34      error.rate <- sum(test$Species != predict(tree, test, type="class"))/nrow(test)
35
36      # devuelve la tasa de aciertos
37      return(1-error.rate)
38    })
39
40    # se muestra el subconjunto y la media de resultados y se devuelve
41    # la media de los resultados (un resultado por particion)
42    print(subset)
43    print(mean(results))
44    return(mean(results))
45  }
46
47  # con esta funcion de evaluacion la seleccion se haria de la forma siguiente
48  subset <- FSelector::best.first.search(names(iris)[-5], evaluator)
49  f <- as.simple.formula(subset,"Species")
50  print(f)
```

Paquete FSelector I

Aproximaciones embedded:

- **random forest importance**

Script fSelector-randomForest.R I

```r
library(mlbench)
library(FSelector)
data(HouseVotes84)

# se calculan los pesos
weights <- FSelector::random.forest.importance(Class~.,HouseVotes84, importance.ty
# se muestran los resultados
print(weights)
subset <- cutoff.k(weights,5)
f <- as.simple.formula(subset,"Class")
print(f)
```

Índice

Paquete caret: particionado (particionDatos.R) I

Facilidades para particionado de datos con diferentes estrategias, manteniendo la proporción de instancias de la variable clase en cada partición.

```
1  library(caret)
2  data(Sonar)
3  set.seed(107)
4
5  # se crea la particion: esto obtiene de forma aleatoria un
6  # porcentaje de instancias dado por p. El metodo mantiene
7  # la proporcion deinstancias para cada valor de la variable
8  # clase
9  inTrain <- caret::createDataPartition(y = Sonar$Class, p = .75,
10                                        list = FALSE)
11
12 # ahora se obtienen los conjuntos de test y de entrenamiento
13 training <- Sonar[ inTrain,]
14 testing  <- Sonar[-inTrain,]
15
16 # se muestra la proporcion de instancias para cada valor de la
17 # variable clase en el conjunto de datos original
18 summary(Sonar$Class)
19 ggplot(data=Sonar) +
20   geom_bar(mapping=aes(x=Class, y=..prop.., group=1))
21
22 # tambien en el de entrenamiento
23 summary(training$Class)
24 ggplot(data=training) +
```

Paquete caret: particionado (particionDatos.R) II

```
25   geom_bar(mapping=aes(x=Class, y=..prop.., group=1))
26
27 # y lo mismo con el de test
28 summary(testing$Class)
29 ggplot(data=testing) +
30   geom_bar(mapping=aes(x=Class, y=..prop.., group=1))
```

Los diagramas de cajas muestran la proporción similar de etiquetas de la variable clase en el conjunto de datos original y en las particiones de entrenamiento y de test.

Paquete caret: correlación (correlacion.R) I

Presenta método sencillo para determinar la correlación entre variables:

```
1  library(caret)
2  library(mlbench)
3
4  # se hace accesible el conjunto de datos PimaIndiansDiabetes
5  data(PimaIndiansDiabetes)
6
7  # se obtiene la matriz de correlacion de las variables predictoras
8  correlationMatrix <- cor(PimaIndiansDiabetes[,1:8])
9
10 # se encuentran aquellas variables que presentan valores de correlacion
11 # por encima del valor umbral
12 highlyCorrelated <- caret::findCorrelation(correlationMatrix,
13                                             cutoff=0.3)
14 print(highlyCorrelated)
```

Salida de la última sentencia: 4 5 8 (índices de variables con alto grado de correlación). Puede visualizarse también la matriz de correlación.

Paquete caret: importancia variables (caret-importance.R) I

Presenta método sencillo para determinar la importancia de las variables:

```
1  library(caret)
2  library(pROC)
3
4  # se fija la semilla para asegurar la reproducibilidad de los
5  # resultados
6  set.seed(7)
7
8  # carga el conjunto de datos
9  data(PimaIndiansDiabetes)
10
11 # prepara el esquema de entrenamiento
12 control <- caret::trainControl(method="repeatedcv", number=10,
13                                 repeats=3)
14
15 # aprende el modelo
16 modelo <- caret::train(diabetes~., data=PimaIndiansDiabetes,
17                        method="lvq", preProcess="scale",
18                        trControl=control)
19
20 # estima la importancia de las variables a partir del modelo
21 importance <- caret::varImp(modelo, scale=FALSE)
22
23 # muestra los datos del analisis
24 print(importance)
25
```

Paquete caret: importancia variables (caret-importance.R) II

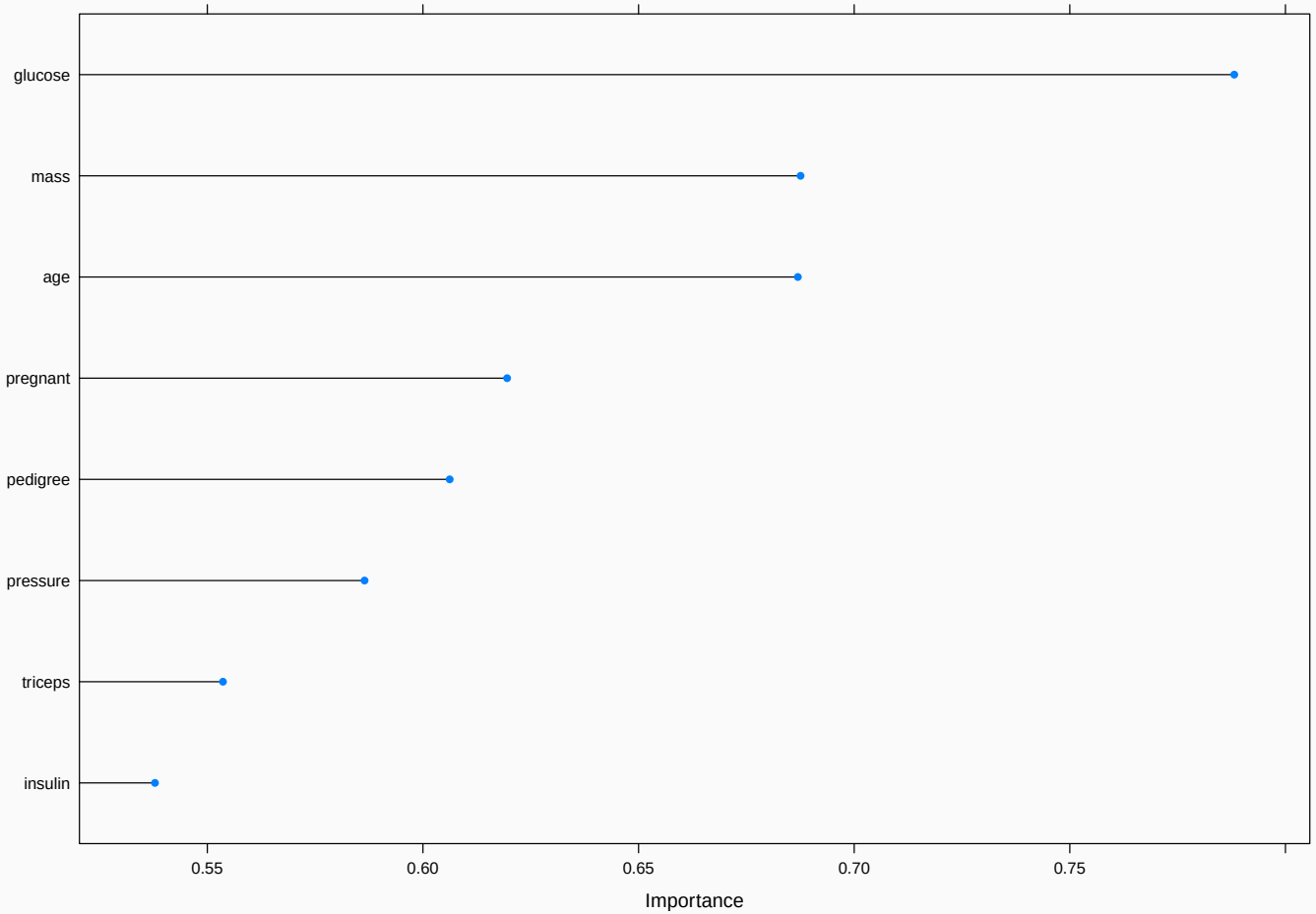
```
26 # representa graficamente los resultados
27 plot(importance,lw=3)
```

A tener en cuenta:

- repetición de 3 procesos de validación cruzada con 10 particiones
- construcción de modelo (aproximación wrapper)
- la salida muestra la importancia de las variables

```
          Importance
glucose       0.7881
mass          0.6876
age           0.6869
pregnant      0.6195
pedigree      0.6062
pressure      0.5865
triceps       0.5536
insulin       0.5379
```

Paquete caret: importancia variables (caret-importance.R) I



Paquete caret: importancia variables (caret-randomForest.R) I

Se puede usar modelo ensamble para determinar la importancia de las variables:

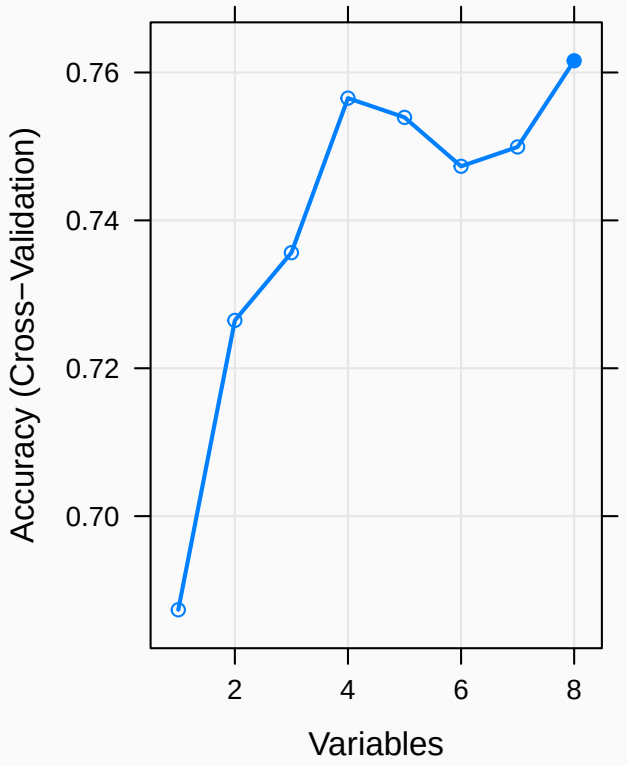
```
1  library(caret)
2
3  # se asigna la semilla para asegura la reproducibilidad de los
4  # resultados
5  set.seed(7)
6
7  # carga el conjunto de datos
8  data(PimaIndiansDiabetes)
9
10 # define el control usando la funcion de seleccion mediante
11 # random forest
12 control <- caret::rfeControl(functions=rfFuncs, method="cv",
13                               number=10)
14
15 # ejecuta el metodo
16 results <- caret::rfe(PimaIndiansDiabetes[,1:8],
17                       PimaIndiansDiabetes[,9], sizes=c(1:8),
18                       rfeControl=control)
19
20 # muestra los resultados
21 print(results)
22
23 # muestra las caracteristicas elegidas
24 predictors(results)
```

Paquete caret: importancia variables (caret-randomForest.R) II

```
25
26 # realiza un grafico de los resultados. El grafico muestra que con
27 # 4 atributos se obtienen resultados simulares a usar los 8 atributos
28 # iniciales
29 plot(results, type=c("g", "o"), lw=2)
```

```
> predictors(results)
[1] "glucose"  "mass"     "age"      "pregnant" "pedigree" "insulin"  "triceps"  "pressure"
```

Paquete caret: importancia variables (caret-randomForest.R) I



Índice

Paquete Boruta: importancia variables (boruta-estadisticas.R) I

Se puede usar modelo ensamble para determinar la importancia de las variables:

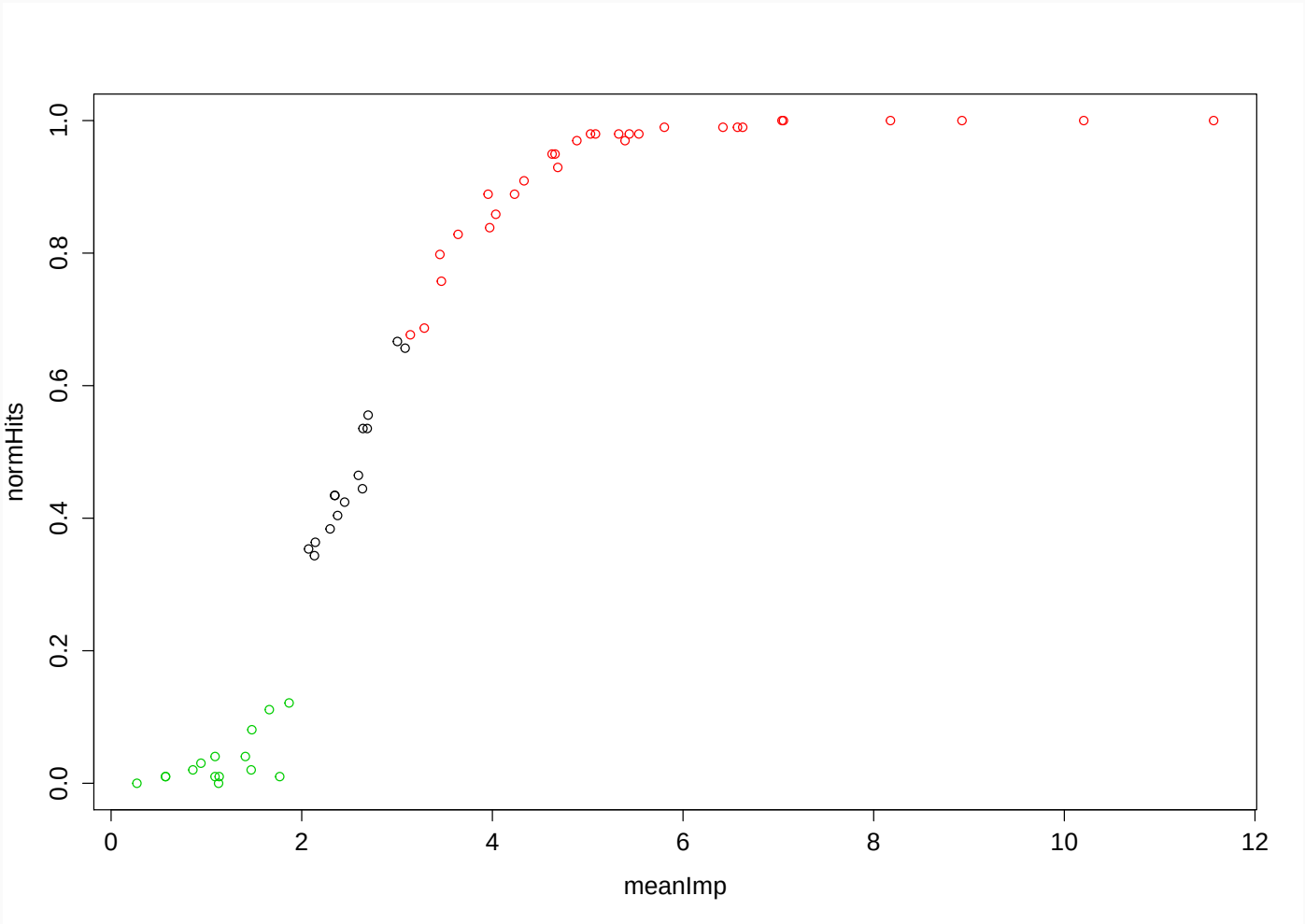
```
1  library(Boruta)
2
3  # carga el conjunto de datos
4  data(Sonar)
5
6  # aprende el modelo
7  Bor.son <- Boruta(Class~.,data=Sonar,doTrace=2)
8
9  # muestra los resultados
10 print(Bor.son)
11
12 # se ven los resultados de decision de cada variable
13 print(Bor.son$finalDecision)
14
15 # imprime las estadisticas
16 stats <- attStats(Bor.son)
17 print(stats)
18
19 # se muestran los resultados en forma grafica
20 plot(Bor.son)
21
22 # muestra un grafico de los resultado: los valores en
```

Paquete Boruta: importancia variables (boruta-estadisticas.R) II

```
23 # rojo estan relacionados con las variables confirmadas
24 # mientras que los verdes con variables descartadas
25 plot(normHits~meanImp,col=stats$decision,data=stats)
```

```
Boruta performed 99 iterations in 6.914203 secs.
 33 attributes confirmed important: V1, V10, V11, V12, V13 and 28 more;
 17 attributes confirmed unimportant: V24, V25, V3, V33, V38 and 12 more;
 10 tentative attributes left: V14, V2, V29, V30, V32 and 5 more;
```

Paquete caret: importancia variables (boruta-estadisticas.R) I



Paquete caret: importancia variables (boruta-estadisticas.R) II

Puede usarse en combinación con random forest (**boruta-randomForest.R**).

Índice

Paquete IPF (ipf.R) I

```r
library(NoiseFiltersR)

# se selecciona el conjunto de datos iris
data(iris)

# se inicializa la semilla aleatoria para reproducir los resultados
set.seed(1)

# se aplica el algoritmo
out <- IPF(Species~., data = iris, s = 2)

# se muestran los indices de las instancias con ruido
summary(out, explicit = TRUE)

# el conjunto de datos sin ruido se obtiene de la siguiente forma
out$cleanData

# tambien podriamos obtenerlo de forma directa eliminando los
# indices de las instancias consideradas como ruidosas
datosSinRuido <- iris[setdiff(1:nrow(iris),out$remIdx),]
```

Paquete IPF (ipf.R) II

```
summary(out, explicit = TRUE)

Filter IPF applied to dataset iris

Call:
IPF(formula = Species ~ ., data = iris, s = 2)

Parameters:
nfolds: 5
consensus: FALSE
p: 0.01
s: 2
y: 0.5

Results:
Number of removed instances: 4 (2.666667 %)
Number of repaired instances: 0 (0 %)

Explicit indexes for removed instances:
71 120 134 135
```

Índice

Clasificación I

Problemas asociados

- validación del rendimiento
- particionado del conjunto de datos
- sobreajuste (poca capacidad de generalización)
- medidas de rendimiento

Índice

Particionado (particionado.R) I

```r
# libreria que incluye algunos conjuntos de datos
# y asociada al libro de igual titulo
library(AppliedPredictiveModeling)
library(caret)

# se selecciona el conjunto de datos de interes: mtcars. Contiene
# 32 instancias y 11 variables: todas son numericas
data(mtcars)

# se obtiene resumen de los datos
summary(mtcars)

# se fija la semilla para el generador de numeros aleatorios,
# con lo que el experimento es reproducible. Para comportamiento
# normal deberia desactivarse esta opcion
set.seed(1)

# se genera un vector con tantos indices como instancias haya en
# el conjunto de datos
indices <- seq(1,nrow(mtcars),by=1)

# se pasa como argumento el vector de indices y de el se van
# seleccionando valores, con la probabilidad indicada. De esta
# forma, el vector resultante deberia tener aproximadamente
# el 80% de valores del pasado como primer argumento
indicesEntrenamiento <- caret::createDataPartition(indices, p=0.8,
                                                   list = FALSE)

# ahora se seleccionan los indices de los datos del conjunto de test
```

Particionado (particionado.R) II

```
30 indicesTest <-indices[-indicesEntrenamiento]
31
32 # con esto es facil ahora seleccionar los conjuntos de datos
33 datosEntrenamiento <- mtcars[indicesEntrenamiento, ]
34 datosTest <- mtcars[indicesTest, ]
35
36 # si necesitamos obtener varias particiones, por ejemplo, para hacer
37 # validacion cruzada mediante metodo Montecarlo, podemos repetir
38 # el proceso el numero de veces que deseemos. Ahora el resultado
39 # sera una matriz con tantas filas como muestras y tantas columnas
40 # como repeticiones se hayan indicado
41 particionadosEntrenamiento <- caret::createDataPartition(indices,
42                                                           p = .80,
43                                                           list= FALSE,
44                                                           times = 10)
45
46 # De esta forma, cada columna contiene los indices de las instancias
47 # seleccionada de una de las particiones
48 particion1 <- particionadosEntrenamiento[,1]
49 particion10 <- particionadosEntrenamiento[,10]
50
51 # ahora podemos obtener los indices de las instancias de test para
52 # cada caso. En primer lugar se calcula el numero de instancias en
53 # el conjunto test. El objetivo es crear una matriz con tantas
54 # filas como tenga el conjunto de test y tantas columnas como
55 # variantes del particionado se hayan generado (en este caso 10).
56 # De esta forma, cada columna servira para almacenar un conjunto
57 # de test. En primer lugar se determina el tam. del conjunto
58 # de test
59 tamConjuntoTest <- nrow(mtcars) - nrow(particionadosEntrenamiento)
```

Particionado (particionado.R) III

```
60
61 # se crea una matriz para almacenar las sentencias de test de
62 # todas las particiones
63 particionadosTest <- matrix(nrow=tamConjuntoTest, ncol = 10)
64
65 # se obtienen de forma iterativa considerando las particiones de
66 # entrenamiento
67 for(i in 1:10){
68   particionadosTest[,i] <- indices[-particionadosEntrenamiento[,i]]
69 }
70
71 # tambien es posible usar createResamples (para bootstrap),
72 # createFolds (para k-fold cross-validation) y createMultiFolds
73 # (para repeticiones de cross-validation). Por ejemplo, imaginemos
74 # deseamos crear 10 particiones para validacion cruzada: el siguiente
75 # metodo me ofrece las particiones de forma que pueda usarlas
76 # posteriormente
77 set.seed(8)
78 particionesEntrenamiento <- caret::createFolds(indices, k = 10,
79                                                list = TRUE,
80                                            returnTrain = TRUE)
81
82 # el objeto devuelto ahora por R es una lista con tantas entradas
83 # como indique k. Cada una de las entradas es un vector con los
84 # indices de las instancias generadas
85 particion1 <- particionesEntrenamiento[[1]]
86 particion10 <- particionesEntrenamiento[[10]]
87
88 # de nuevo podemos usar la misma estrategia de antes para generar los
89 # indices de las correspondientes particiones de test
```

Particionado (particionado.R) IV

```
90  particionesTest <- list()
91  for(i in 1:10){
92    particionesTest[[i]] <- indices[-particionesEntrenamiento[[i]]]
93  }
94
95  # para que sea mas visible, vamos a considerar que tenemos 10
96  # instancias
97  indices <- seq(1,10)
98  particionesEntrenamiento <- caret::createFolds(indices, k = 10,
99                                                 returnTrain = TRUE)
100
101 # e igual para los indices de test
102 particionesTest <- list()
103 for(i in 1:10){
104   particionesTest[[i]] <- indices[-particionesEntrenamiento[[i]]]
105 }
106
107 # probamos el uso de la tecnica de bootstrap: ahora habra
108 # muestras repetidas y ademas todas las particiones tienen
109 # el mismo numero de muestras que el conjunto de datos original
110 particionBootstrap <- caret::createResample(indices, times=10)
111
112 # tambien podemos probar la creacion de multiplesParticiones: en este
113 # caso 3 particionados completos, de 10 particiones cada una de ellas
114 multiplesParticiones <- caret::createMultiFolds(indices, k=10,
115                                                 times=3)
```

Índice

Construcción de modelos (construccionModelos.R) I

```
1  library(caret)
2  library(e1071)
3  data(GermanCredit)
4
5  # ********************** PARTE 1 ********************************
6
7  # se crean las particiones del conjunto de datos. En este caso
8  # se usa con conjunto de datos con 1000 instancias y 62 variables.
9  # Se generan las particiones del conjunto de datos mediante
10 # la funcion createFolds, que genera 10 particiones
11 indices <- seq(1,nrow(GermanCredit),by=1)
12 particionesEntrenamiento <- createFolds(indices, k = 10,
13                                          returnTrain = TRUE)
14
15 # genero de la misma forma las particiones de test
16 particionesTest <- list()
17 for(i in 1:10){
18   particionesTest[[i]] <- indices[-particionesEntrenamiento[[i]]]
19 }
20
21 # usaremos estas particiones para hacer validacion cruzada
22 # Bucle de generacion de modelos
23 errores <- c()
24 aciertos <- c()
25 for(i in 1:10){
26   # hay varias formas de especificar el objetivo del modelo a
27   # construir. Una de ellas es la formula: se compone de dos
28   # terminos separados por el simbolo ~: a la izquierda va
29   # la variable a predecir (clasificacion o regresion) y a la
```

Construcción de modelos (construccionModelos.R) II

```
30    # derecha las variables a usar como predictoras. En el ejemplo
31    # siguiente el punto a la derecha indica que se usan todos los
32    # atributos como atributos. Como ejemplo veremos la aplicacion
33    # del metodo de aprendizaje a una de las particiones
34    modelo <- train(Class ~ .,
35                    data = GermanCredit[particionesEntrenamiento[[i]], ],
36                    method = "svmRadial")
37    cat("Aprendido modelo ",i,"\n")
38
39    # se realizan las predicciones sobre el conjunto de test
40    # asociado
41    # por supuesto es posible obtener las predicciones para las
42    # instancias del conjunto de test
43    predicciones <- predict(modelo, GermanCredit[particionesTest[[i]], ])
44
45    # se determinan las diferencias entre prediccion y valor real de
46    # la clase
47    diferencias = (GermanCredit[particionesTest[[i]],"Class"] != predicciones)
48    errores[i] <- length(which(diferencias == TRUE))
49    aciertos[i] <- length(which(diferencias == FALSE))
50  }
51
52  # se calcula la tasa de aciertos
53  tasaAciertos <- mean(aciertos/(errores+aciertos))
54
55  # el modelo general propuesto se construiria ahora con todos los
56  # datos
57  modeloFinal <- train(Class ~ .,
58                       data = GermanCredit, method = "svmRadial")
59
```

Construcción de modelos (construccionModelos.R) III

```
60 # ********************** PARTE 2 ********************************
61
62 # tambien es posible aplicar algunas operaciones de preprocesamiento
63 # sobre los datos: centrado y escalado, por ejemplo, al emplear una
64 # tecnica de aprendizaje sensible a las dimensiones y escala de los
65 # datos
66 modelo2 <- train(Class ~ .,
67                  data = GermanCredit[particionesEntrenamiento[[1]], ],
68                  method = "svmRadial", preProc = c("center", "scale"))
69
70 # por supuesto es posible obtener las predicciones para las
71 # instancias del conjunto de test
72 predicciones <- predict(modelo2, GermanCredit[particionesTest[[1]], ])
73 str(predicciones)
74
75 # se determinan las diferencias entre prediccion y valor real de
76 # la clase
77 diferencias= (GermanCredit[particionesTest[[1]],"Class"] != predicciones)
78 errores <- length(which(diferencias == TRUE))
79 aciertos <- length(which(diferencias == FALSE))
80
81 # ********************** PARTE 3 ********************************
82
83 # este metodo tiene un parametro de coste que regula el coste asociado
84 # a los errores de prediccion: las diferencias entre el valor predicho
85 # y el real. Es posible evaluar diferentes valores de coste directamente,
86 # haciendo uso del ultimo argumento (costes 2^-2, 2^-1,
87 # 2^0, .... 2^7)
88 modelo3 <- train(Class ~ .,
89                  data = GermanCredit[particionesEntrenamiento[[1]], ],
```

Construcción de modelos (construccionModelos.R) IV

```
90                    method = "svmRadial", preProc = c("center", "scale"),
91                    tuneLength = 10)
92
93  # y volvemos a hacer las predicciones
94  # por supuesto es posible obtener las predicciones para las
95  # instancias del conjunto de test
96  predicciones <- predict(modelo3, GermanCredit[particionesTest[[1]], ])
97  str(predicciones)
98
99  # se determinan las diferencias entre prediccion y valor real de
100 # la clase
101 diferencias= (GermanCredit[particionesTest[[1]],"Class"] != predicciones)
102 errores <- length(which(diferencias == TRUE))
103 aciertos <- length(which(diferencias == FALSE))
104
105 # puede imprimirse la informacion de los modelos considerados
106 plot(modelo3, scales = list(x = list(log = 2)))
107
108 # ********************** PARTE 4 *********************************
109
110 # tambien se puede modificar esta llamada para que se utilicen
111 # diferentes particionados (que se generan de forma automatica
112 # por parte de la funcion de aprendizaje). En este ejemplo se
113 # realizan 5 repeticiones de validacion cruzada con k=10
114 # se inicializa la semilla (si interesa). De esta forma se
115 # automatiza de forma completo el proceso de aprendizaje y
116 # de estimacion. Se observa informacion sobre el particionado
117 # al analizar la salida del modelo
118 set.seed(1)
119 modelo4 <- train(Class ~ ., data = GermanCredit,
```

Construcción de modelos (construccionModelos.R) V

```
120                   method = "svmRadial", preProc = c("center", "scale"),
121                   tuneLength=10,
122                   trControl = trainControl(method = "repeatedcv",
123                                            repeats=5))
124
125 # se muestra la relacion entre el valor de C usado y la fiabilidad
126 # predictiva
127 plot(modelo4)
128
129 # ********************** PARTE 5 ********************************
130
131 # tambien es posible hacer comparacion con otros modelos. Imaginemos
132 # un modelo de regresion logistica (clasificacion) para estos datos.
133 # Seproduce la misma inicializacion de semilla
134 set.seed(1)
135 modeloRegLog <- train(Class ~ ., data=GermanCredit, method="glm",
136                       trControl= trainControl(method="repeatedcv",
137                                               repeats=5))
138
139 # la funcion resamples nos sirve para comparar estos modelos
140 resamp <- resamples(list(SVM = modelo4, Logistic = modeloRegLog))
141 summary(resamp)
142
143 # se obtienen las diferencias entre ambos modelos: supone realizar
144 # un test estadistico, siendo la hipotesis nula la igualdad (lo que
145 # indicaria que los modelos se comportan de forma similar)
146 diferencias <- diff(resamp)
147 summary(diferencias)
148
```

Construcción de modelos (construccionModelos.R) VI

```
149 # Si los p-valores son muy bajos se acepta la hipotesis
150 # nula: los modelos son similares
```

Índice

Máquinas de soporte vectorial I

```r
library(e1071)
library(caret)

# se cargan los datos
data(iris)

# se crea la particion: esto obtiene de forma aleatoria un porcentaje
# de instancias dado por p
inTrain <- caret::createDataPartition(y = iris$Species, p = .75, list = FALSE)

# ahora se obtienen los conjuntos de test y de entrenamiento
training <- iris[ inTrain,]
testing  <- iris[-inTrain,]

# se construye el modelo
model <- e1071::svm(Species~., data=training, method="C-classification",
              kernel="radial", cost=10, gamma=0.1)

# se muestra informacion sobre el modelo
summary(model)

# se hace prediccion
pred <- predict(model, testing, decision.values = TRUE)

# para interpretarlo bien se trata como una tabla
tab <- table(pred = pred, true = testing[,5])

# se muestra la tabla
print(tab)
```

Índice

Árboles de clasificación I

```
1  library(party)
2  library(caret)
3
4  # se carga la funcionalidad de lectura de datos
5  source("./lecturaDatos.R")
6
7  # se fija ls ruta de localizacion de los datos
8  path <- "./data/"
9  file <- "datos.csv"
10
11 # se llama a la funcion que carga los datos
12 datos <- lecturaDatos(path,file)
13
14 # se obtiene el nombre de la variable clase
15 # primero se obtiene la posicion de la variable clase: se asume
16 # que es la ultima (pero no podemos estar seguros)
17 posicionClase <- length(names(datos))
18 variableClase <- names(datos)[posicionClase]
19
20 # se compone una formula con el nombre de la variable clase y ~
21 formulaClase <- as.formula(paste(variableClase,"~.",sep=""))
22
23 # se divide el conjunto de datos en train y test
24 # se crea la particion: esto obtiene de forma aleatoria un porcentaje
25 # de instancias dado por p. Esta funcionalidad esta disponible en el
26 # paquete caret
27 inTrain <- createDataPartition(y=datos[,posicionClase], p = .75,
28                                list = FALSE)
29
```

Árboles de clasificación II

```
30 # ahora se obtienen los conjuntos de test y de entrenamiento
31 training <- datos[ inTrain,]
32 testing  <- datos[-inTrain,]
33
34 # construye el modelo
35 ct <- ctree(formulaClase, training)
36
37 # muestra el arbol de forma grafica, pero no tiene demasiado sentido
38 # al ser demasiado grande
39 # plot(ct)
40
41 # se muestra en modo texto: observad que este metodo no precisa
42 # discretizacion previa al poder ir considerando diferentes cortes
43 # en variables numericas. Esto hace que puedan aparecer varias
44 # veces en el arbol
45 print(ct)
46
47 # se realiza la prediccion
48 testPred <- predict(ct, newdata = testing)
49
50 # se compara con los datos de test
51 results <- table(testPred, testing$class)
52
53 # se suman los valores de la diagonal
54 sumDiag <- sum(diag(results))
55
56 # se suman todos los valores de la matriz
57 sumTotal <- sum(results)
58
59 # se calcula el porcentaje de aciertos
```

Árboles de clasificación III

```
60 fiabilidad <- sumDiag/sumTotal
61
62 # Se calcula el error
63 error <- 1-fiabilidad
64
65 # usamos un conjunto de datos mas sencillo para ver la forma
66 # que tendria el arbol generado
67 datos <- iris
68
69 # se obtiene el nombre de la variable clase
70 # primero se obtiene la posicion de la variable clase: se asume
71 # que es la ultima (pero no podemos estar seguros)
72 posicionClase <- length(names(datos))
73 variableClase <- names(datos)[posicionClase]
74
75 # se compone una formula con el nombre de la variable clase y ~
76 formulaClase <- as.formula(paste(variableClase,"~.",sep=""))
77
78 # se divide el conjunto de datos en train y test
79 # se crea la particion: esto obtiene de forma aleatoria un porcentaje
80 # de instancias dado por p. Esta funcionalidad esta disponible en el
81 # paquete caret
82 inTrain <- createDataPartition(y=datos[,posicionClase], p = .75,
83                                list = FALSE)
84
85 # ahora se obtienen los conjuntos de test y de entrenamiento
86 training <- datos[ inTrain,]
87 testing  <- datos[-inTrain,]
88
89 # construye el modelo
```

Árboles de clasificación IV

```
90  ct <- ctree(formulaClase, training)
91
92  # ahora si vale la pena visualizar el arbol
93  plot(ct)
94
95  # se visualiza la tabla con las predicciones realizadas
96  predicciones <- predict(ct, testing)
97
98  # tambien pueden estimarse las probabilidades de asignacion
99  # de cada instancias a cada clase
100 probabilidades <- predict(ct, testing, type="prob")
```

Índice

Métodos ensamble: bagging I

```r
# la libreria rpart se usa para disponer de metodos de
# construccion de arboles de clasificacion
library(rpart)

# aqui hay datos considerados como de referencia
library(mlbench)

# libreria necesaria para funcionalidad de construccion
# de ensembles mediante bagging
library(adabag)

# se definen los datos a usar
data(Vehicle)

# se crean las particiones del conjunto de datos. En este caso
# se usa con conjunto de datos con 1000 instancias y 62
# variables. Se generan las particiones del conjunto de datos
# mediante la funcion createFolds, que genera 10 particiones
indices <- seq(1,nrow(Vehicle),by=1)
particionesEntrenamiento <- createFolds(indices, k = 10,
                                        returnTrain = TRUE)

# genero de la misma forma las particiones de test
particionesTest <- list()
for(i in 1:10){
  particionesTest[[i]] <- indices[-particionesEntrenamiento[[i]]]
}

# usaremos estas particiones para hacer validacion cruzada
```

Métodos ensamble: bagging II

```
30 # Bucle de generacion de modelos
31 errores <- c()
32 aciertos <- c()
33 for(i in 1:10){
34   # hay varias formas de especificar el objetivo del modelo a
35   # construir. Una de ellas es la formula: se compone de dos
36   # terminos separados por el simbolo ~: a la izquierda va
37   # la variable a predecir (clasificacion o regresion) y a la
38   # derecha las variables a usar como predictoras. En el ejemplo
39   # siguiente el punto a la derecha indica que se usan todos los
40   # atributos como atributos. Como ejemplo veremos la aplicacion
41   # del metodo de aprendizaje a una de las particiones
42   modelo <- adabag::bagging(Class ~ .,
43                             data = Vehicle[particionesEntrenamiento[[i]], ],
44                             control=rpart::rpart.control(maxdepth=5, minsplit=15))
45   cat("Aprendido modelo ",i,"\n")
46
47   # se realizan las predicciones sobre el conjunto de test
48   # asociado. Por supuesto es posible obtener las predicciones
49   # para las instancias del conjunto de test
50   predicciones <- adabag::predict.bagging(modelo,
51                                           newdata=Vehicle[particionesTest[[i]], ])
52
53   # se determinan las diferencias entre prediccion y valor real de
54   # la clase
55   errores[i] <- predicciones$error
56   aciertos[i] <- 1-predicciones$error
57   cat("  Aciertos:", aciertos[i], " Errores: ", errores[i], "\n")
58 }
59
```

Métodos ensamble: bagging III

```
60 # se calcula la tasa de aciertos
61 tasaAciertos <- mean(aciertos/(errores+aciertos))
62
63 # usaremos estas particiones para hacer validacion cruzada
64 # Bucle de generacion de modelos
65 errores <- c()
66 aciertos <- c()
67 for(i in 1:10){
68   # se cambian los parametros de control para la construccion de
69   # los arboles. Aqui el modelo final contendra unicamente 5 arboles
70   # y se limita su tam. a profuncidad maxima de tres
71   modelo <- adabag::bagging(Class ~ .,
72                     data = Vehicle[particionesEntrenamiento[[i]], ],
73                     mfinal=20,
74                     control=rpart::rpart.control(maxdepth=3, minsplit=5))
75   cat("Aprendido modelo ",i,"\n")
76
77   # se realizan las predicciones sobre el conjunto de test
78   # asociado. Por supuesto es posible obtener las predicciones
79   # para las instancias del conjunto de test
80   predicciones <- adabag::predict.bagging(modelo,
81                                           newdata=Vehicle[particionesTest[[i]], ])
82
83   # se determinan las diferencias entre prediccion y valor real de
84   # la clase
85   errores[i] <- predicciones$error
86   aciertos[i] <- 1-predicciones$error
87   cat("  Aciertos:", aciertos[i], " Errores: ", errores[i], "\n")
88 }
89
```

Métodos ensamble: bagging IV

```
90 # se calcula la tasa de aciertos
91 tasaAciertos <- mean(aciertos/(errores+aciertos))
```

Métodos ensamble: random forest I

```r
library(caret)
library(randomForest)

# se carga el conjunto de datos
data("GermanCredit")

# se aprende el modelo: random forest. Podemos especificar el
# numero de arboles a incluir en el bosque
modelo <- randomForest::randomForest(Class ~ ., data=GermanCredit, ntree=10)

# se muestra informacion del modelo
print(modelo)

# muestra la importancia de los atributos, teniendo en cuenta
# el modelo construido
randomForest::importance(modelo)

# se muestra informacion sobre los errores para cada una de las
# clases: la linea en negro indica el error medio
plot(modelo)

# se aprende otrol modelo con mas arboles
modelo2 <- randomForest(Class ~ ., data=GermanCredit, ntree=100)

# se muestran los errores para cada etiqueta de la variable clase
plot(modelo2)
```

Métodos ensamble: random forest I

```r
library(adabag)
library(caret)

# se carga el conjunto de datos
data(Vehicle)

# se hace un particionado sencillo del conjunto de datos
indices <- seq(1,nrow(Vehicle),by=1)
indicesEntrenamiento <- caret::createDataPartition(indices, p=0.8, list=FALSE)
datosEntrenamiento <- Vehicle[indicesEntrenamiento, ]
datosTest <- Vehicle[-indicesEntrenamiento, ]

# se realiza el aprendizaje. El argumento mfinal indica el numero
# de iteraciones del proceso, o lo que es lo mismo, el numeor de
# modelos que se construyen. El argumento maxdepth se usa por parte
# del paquete rpart, que es el usado para construir los arboles
# del ensamblado
modelo <- adabag::boosting(Class ~ ., data = datosEntrenamiento,
                           mfinal = 10,
                           control = rpart::rpart.control(maxdepth = 2))

# se muestra un grafico indicando la importancia relativa
# de las variables
barplot(modelo$imp[order(modelo$imp, decreasing = TRUE)],
          ylim = c(0, 100), main = "Variables Relative Importance",
          col = "lightblue")

# la prediccion se realizaria de la siguiente forma
prediccion <- predict.boosting(modelo, newdata = datosTest)
```

Métodos ensamble: random forest II

```
30
31 # se muestra el resultado. Se observa que la prediccion es de tipo probabilistico.
32 # Para cada una de las 36 instancias del conjunto de test se muestra la probabilid
33 # de pertenencia a cada clase
34 print(modelo)
35
36 # se calcula el error sobre el conjunto de test
37 evol.test <- errorevol(modelo,newdata=datosTest)
38
39 # se calcula el error sobre el conjunto de entrenamiento
40 evol.train <- errorevol(modelo,newdata=datosEntrenamiento)
41
42 # se muestra la evolucion del error a medida que se construyen
43 # mas arboles
44 plot(evol.test$error, type="l", main="AdaBoost error Vs numero arboles",
45      xlab="Arboles", ylab="Error", col = "red")
46
47 # igual con el error en el conjunto de entrenamiento
48 plot(evol.train$error, col="blue",type="l",
49      main="AdaBoost error Vs numero arboles",
50      xlab="Arboles", ylab="Error")
```