Proyecto Final - Aprendizaje Automático

David López Pretel y Néstor Rodriguez Vico

7 de junio de 2017

Introducción

La base de datos que hemos cogido es *Breast Cancer Wisconsin (Diagnostic)*. Se trata de una base de datos con 569 ejemplos y con 30 características más la clase. Se trata de un problema de clasificación binaria donde las clases son *Malignant* y *Benign*. Las 30 características son de 3 imágenes de 3 células distintas en las cuales se han medido 10 características para cada imagen.

Preparación de los datos

En cuanto a la preparación de los datos hemos quitado la primera columna ya que es un mero identificador de cada ejemplo y hemos cambiado la clase a valores numéricos, para que luego se pueda hacer una predicción sobre la misma.

```
set.seed(3)
suppressWarnings(suppressMessages(library("caret"))) # preProcess
suppressWarnings(suppressMessages(library("ROCR"))) # prediction
suppressWarnings(suppressMessages(library("pracma"))) # trapz

wdbc = read.csv("./Datos/wdbc.data", header = F, sep = ",", stringsAsFactors=FALSE)
# Eliminamos el identificador
wdbc = wdbc[, -1]
# Cambiamos el nombre de las columnas (por comodidad)
names(wdbc) = c("clase", 1:(dim(wdbc)[2]-1))
# Cambiamos la clase, Maligno por 1 y benigno por 0
clase = mapply(function(x) ifelse(x=="M", 1, 0), wdbc[,1], USE.NAMES = F)
wdbc = cbind(clase, wdbc[,-1])

# Comprobamos si hay valores NA
apply(wdbc, 2, function(x) sum(is.na(x)))
```

```
## clase     1     2     3     4     5     6     7     8     9    10    11
##     0     0     0     0     0     0     0     0     0     0     0     0
##    12    13    14    15    16    17    18    19    20    21    22    23
##     0     0     0     0     0     0     0     0     0     0     0     0
##    24    25    26    27    28    29    30
##     0     0     0     0     0     0     0
```

Como hemos visto, no hay ningún atributo perdido (NA) en nuestra base de datos, por lo que no debemos realizar ningún tratamiento especial de los mismos. Ahora vamos a realizar las distintas particiones. Para ello, vamos a crear cinco conjuntos con el 20% de datos cada uno de ellos y vamos a aplicar la técnica *k-fold* con k = 5.

```
reducir = function(lista, size, indice){
  if(length(unlist(lista)) > size){
    aux = unlist(lista, use.names = F)
    guardar = aux[length(aux)]
```

```
    aux = aux[-length(aux)]
    list(aux, guardar, indice)
  }
}

sobrantes = function(lista, size){
  sobrantes = c()
  for(i in 1:5){
    resultado = reducir(lista[i], size, i)
    if(!is.null(resultado)){
      lista[[resultado[[3]]]] = resultado[[1]]
      sobrantes = c(sobrantes, resultado[[2]])
    }
  }
  list(lista, sobrantes)
}

# Creamos los conjuntos de datos de manera balanceada
# y haciendo k-fold con k = 5 (80% train y 20% test)
pos = sample(which(wdbc[,1] == 1))
neg = sample(which(wdbc[,1] == 0))

# Separamos las tiquetas positivas y negativas
# para tener unos conjuntos balanceados
particiones_pos = split(pos, cut(seq_along(pos), 5, labels = F))
particiones_neg = split(neg, cut(seq_along(neg), 5, labels = F))
#Cogemos los datos de los indices según las particiones de los indices
wdbc.pos <- lapply(particiones_pos, function(part) wdbc[part,])
wdbc.neg <- lapply(particiones_neg, function(part) wdbc[part,])

#Debemos comprobar que sean todos del mismo tamaño, así que cogemos el mínimo
pos.size = min(unlist(lapply(particiones_pos, length)))
neg.size = min(unlist(lapply(particiones_neg, length)))

# Guardamos las asignaciones sobrantes
resultado = sobrantes(particiones_pos, pos.size)
particiones_pos = resultado[[1]]
pos.extras = as.vector(resultado[[2]])
resultado = sobrantes(particiones_neg, neg.size)
particiones_neg = resultado[[1]]
neg.extras = as.vector(resultado[[2]])
extras = c(pos.extras, neg.extras)

#Cogemos los datos de los indices según las particiones de los indices
wdbc.pos <- lapply(particiones_pos, function(part) wdbc[part,])
wdbc.neg <- lapply(particiones_neg, function(part) wdbc[part,])

#Generamos los cinco conjuntos
conjunto_1 = rbind(wdbc.pos[[1]], wdbc.neg[[1]])
conjunto_2 = rbind(wdbc.pos[[2]], wdbc.neg[[2]])
conjunto_3 = rbind(wdbc.pos[[3]], wdbc.neg[[3]])
conjunto_4 = rbind(wdbc.pos[[4]], wdbc.neg[[4]])
conjunto_5 = rbind(wdbc.pos[[5]], wdbc.neg[[5]])
```

```r
#Añadimos lo que ha sobrado
l = list(conjunto_1, conjunto_2, conjunto_3, conjunto_4, conjunto_5)
for(i in 1:length(extras)){
  l[[i]] = rbind(l[[i]], wdbc[extras[i],])
}

# Ya tenemos los conjuntos creados (cada uno tiene el 20% de los datos)
conjunto_1 = l[[1]]
conjunto_2 = l[[2]]
conjunto_3 = l[[3]]
conjunto_4 = l[[4]]
conjunto_5 = l[[5]]

#Repartimos los conjuntos
test_1 = conjunto_1
train_1 = rbind(conjunto_2, conjunto_3, conjunto_4, conjunto_5)
test_2 = conjunto_2
train_2 = rbind(conjunto_1, conjunto_3, conjunto_4, conjunto_5)
test_3 = conjunto_3
train_3 = rbind(conjunto_1, conjunto_2, conjunto_4, conjunto_5)
test_4 = conjunto_4
train_4 = rbind(conjunto_1, conjunto_2, conjunto_3, conjunto_5)
test_5 = conjunto_5
train_5 = rbind(conjunto_1, conjunto_2, conjunto_3, conjunto_4)
```

Explicación del procedimiento

Para obtener unos resultados más fiables hemos usado la técnica de validación cruzada con *k-fold*, en la cual vamos a hacer 5 repeticiones del procedimiento explicado a continuación. El procedimiento seguido para cada modelo es el mismo:

1. Cogemos como test el 20% de los datos (correspondiente a la partición i-ésima donde i representa la iteración) y como train el 80% restante de los datos.
2. A continuación los preprocesarmos. Para ello usamos la función *preProcess*. El preprocesamiento que hacemos es un centrado y un escalado, ya que los datos no dependen de máximos, ni de mínimos, ni dependen de la varianza de los mismos. Este preprocesamiento se hace sobre el conjunto de train y el resultado obtenido se aplica al conjunto de test. Esto se hace para que las modificaciones que se hacen en el train no se vean influidas por el conjunto de test.
3. Una vez tenemos los datos preprocesados ajustamos el modelo correspondiente. Los modelos reciben como argumento una fórmula que describe el modelo que se va a ajustar. Todos nuestros modelos van a tener la siguiente fórmula: *clase* $\sim$ *resto de características*. Esto quiere decir que nuestro modelo va a predecir la clase en función del resto de características de nuestro conjunto de datos.
4. Con el modelo ajustado, predecimos las clases del conjunto de test.
5. Con las clases predecidas y con las clases reales de los ejemplos de test calculamos la curva ROC y la pintamos. También calculamos el área debajo de la curva ROC y la guardamos. La curva ROC representa el ratio entre verdaderos positivos frente al ratio de falsos positivos según el umbral de discriminación, valor a partir del cual decidimos que un caso es positivo. Nosotros vamos a usar la curva ROC para calcular el area bajo la misma. A mayor área, mejor es el modelo.

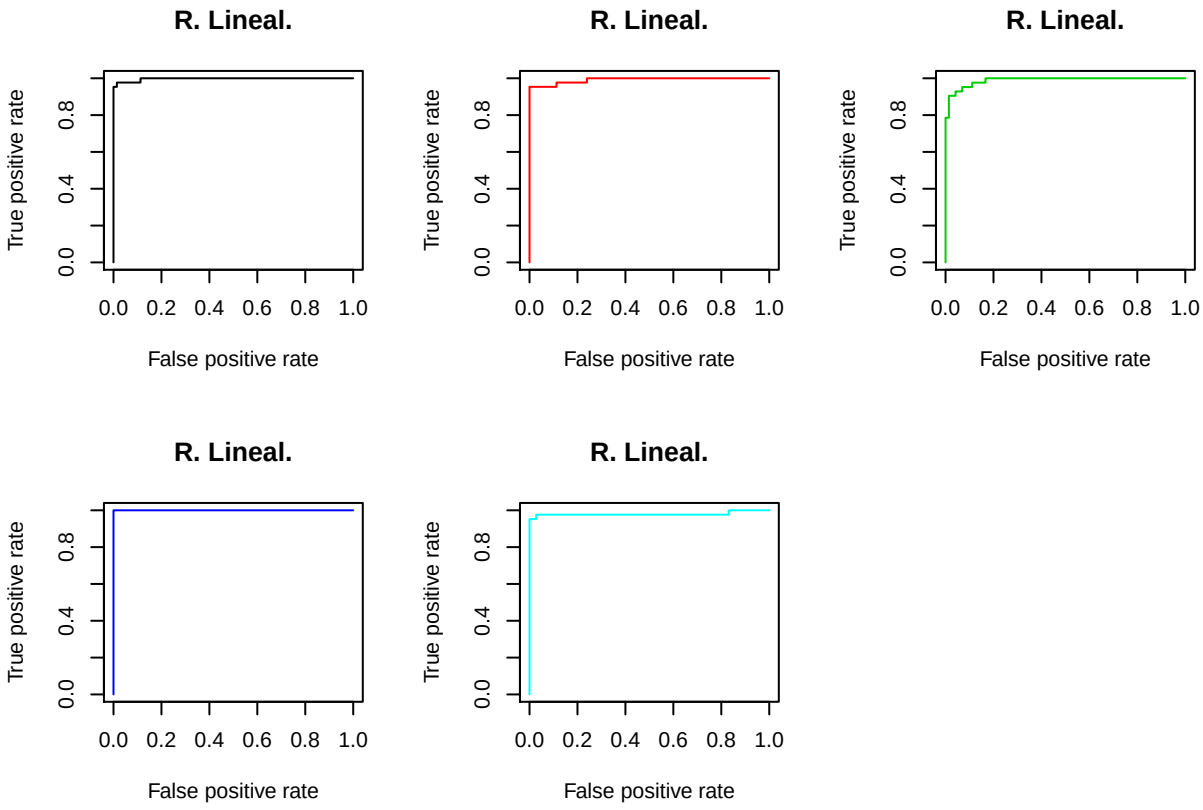
Una vez hemos acabado el proceso de validación cruzada, nuestro resultado es la media de las areas obtenidas.

Modelo lineal: Regresión Lineal

Para el modelo líneal vamos a usar Regresión Lineal y Regresión Logística para compararlas. Ahora vamos a usar Regresión Lineal y posteriormente Logística. Vamos a usar la función *glm*, esta función recibe como argumentos el conjunto de datos, *train* en nuestro caso, y la fórmula que describe el modelo que se va a ajustar.

```
RegresionLineal = function(plot = F) {
  set.seed(3)
  par(mfrow=c(2,3))
  areas = NULL

  for(i in 1:5){
    # Preprocesamos los datos
    train = eval(as.name(paste0("train_", i)))
    test = eval(as.name(paste0("test_", i)))
    clase = train[,1]
    train = train[,-1]
    # No hacemos YeoJohnson porque da un warning por problemas de convergencia
    # "Convergence failure: return code = 52"
    obj = preProcess(train, method = c("center", "scale", "pca"), thresh=0.95)
    train = predict(obj, train)
    train = as.data.frame(cbind(clase, train))
    clase = test[,1]
    test = test[,-1]
    test = predict(obj, test)
    test = as.data.frame(cbind(clase, test))

    # Estimamos el modelo
    rlModel = glm(train$clase ~ ., data = train)

    # Obtenemos la curca ROC
    predict = predict(rlModel, as.data.frame(test[,-1]), type="response")
    prediction = prediction(predict, test$clase)
    rend = performance(prediction,"tpr","fpr")
    # Y el area debajo de la curva ROC
    area = trapz(unlist(rend@x.values), unlist(rend@y.values))
    # Guardamos el area
    areas[i] = area

    if(plot == T){
      plot(rend, col = i, main = "R. Lineal.")
    }
  }
  areas
}

areaRLineal = RegresionLineal(plot = T)
cat("Area media bajo la curva ROC para un modelo lineal (regresión lineal):",
    mean(areaRLineal), "\n")
```

```
## Area media bajo la curva ROC para un modelo lineal (regresión lineal): 0.9914989
```



Modelo lineal: Regresión Logística

Ahora vamos a usar Regresión Logística. Esta función recibe como argumentos el conjunto de datos, *train* en nuestro caso, la fórmula que describe el modelo que se va a ajustar y debemos indicarle en el argumento *family* que use como familia de funciones *binomial(logit)*, lo cual nos permite hacer regresión logística.

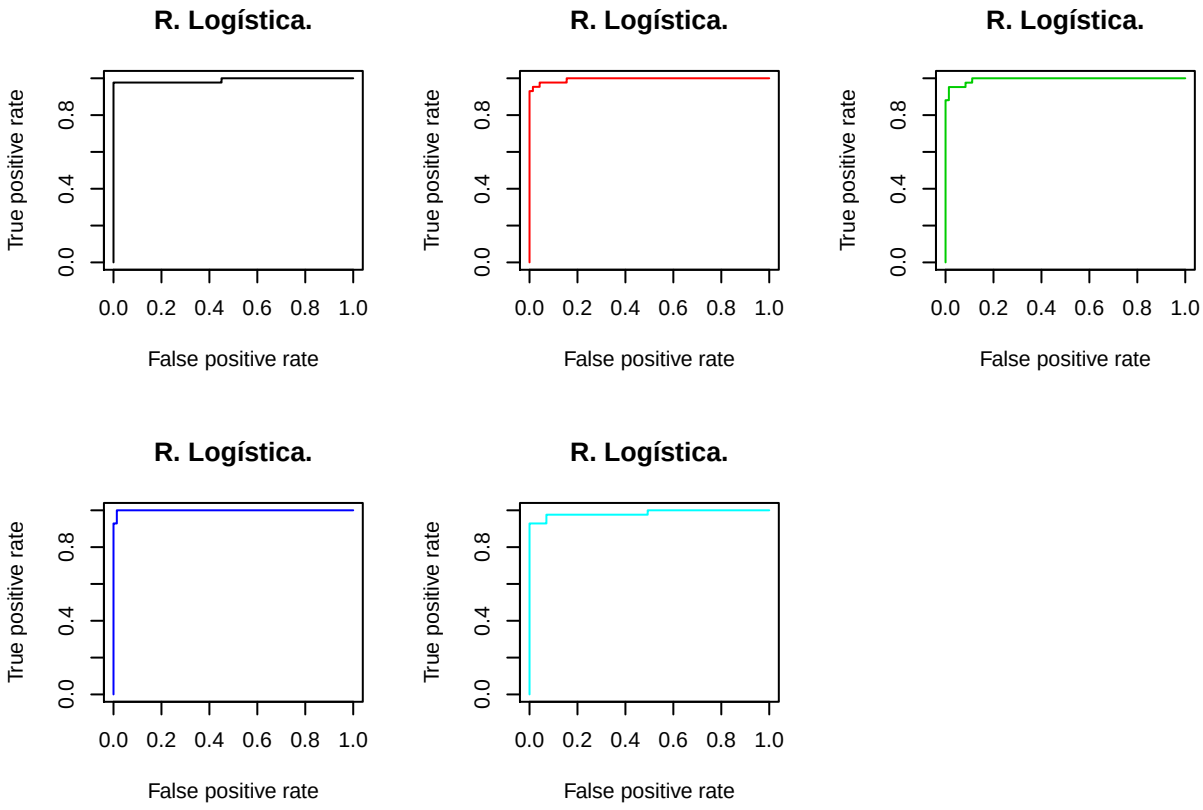
```
RegresionLogistica = function(plot = F) {
  set.seed(3)
  par(mfrow=c(2,3))
  areas = NULL

  for(i in 1:5){
    # Preprocesamos los datos
    train = eval(as.name(paste0("train_", i)))
    test = eval(as.name(paste0("test_", i)))
    clase = train[,1]
    train = train[,-1]
    # No hacemos YeoJohnson porque da un warning por problemas de convergencia
    # "Convergence failure: return code = 52"
    obj = preProcess(train, method = c("center", "scale", "pca"), thresh=0.95)
    train = predict(obj, train)
    train = as.data.frame(cbind(clase, train))
    clase = test[,1]
    test = test[,-1]
    test = predict(obj, test)
    test = as.data.frame(cbind(clase, test))
```

```
    # Estimamos el modelo
    rlModel = glm(train$clase ~ ., data = train, family = binomial(logit))

    # Obtenemos la curca ROC
    predict = predict(rlModel, as.data.frame(test[,-1]), type="response")
    prediction = prediction(predict, test$clase)
    rend = performance(prediction,"tpr","fpr")
    # Y el area debajo de la curva ROC
    area = trapz(unlist(rend@x.values), unlist(rend@y.values))

    # Guardamos el area
    areas[i] = area

    if(plot == T){
      plot(rend, col = i, main = "R. Logística.")
    }
  }
  areas
}

areaRlogistica = RegresionLogistica(plot = T)
```

```
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
cat("\nArea media bajo la curva ROC para un modelo lineal (regresión logística):",
    mean(areaRlogistica), "\n")
```

```
##
## Area media bajo la curva ROC para un modelo lineal (regresión logística): 0.9925802
```



Como podemos ver, obtenemos muy buenos resultados usando modelos lineales. De hecho, al usar regresión logística obtenemos un warning que nos indica que han ocurrido probabilidades 0 ó 1. Esto quiere decir que un modelo líneal es bastante potente como para poder obtener unos buenos resultados. Esto nos lleva a pensar que los datos son linealmente separables. Hemos hecho regresión lineal y regresión logística por comparar ambos modelos, aunque por la teoría sabemos que la regresión logística obtiene unos mejores resultados en problemas de clasificación binaria, ya que da una probabilidad de que una muestra pertenezca a una clase en vez de dar la clase directamente como sucede en regresión líneal.

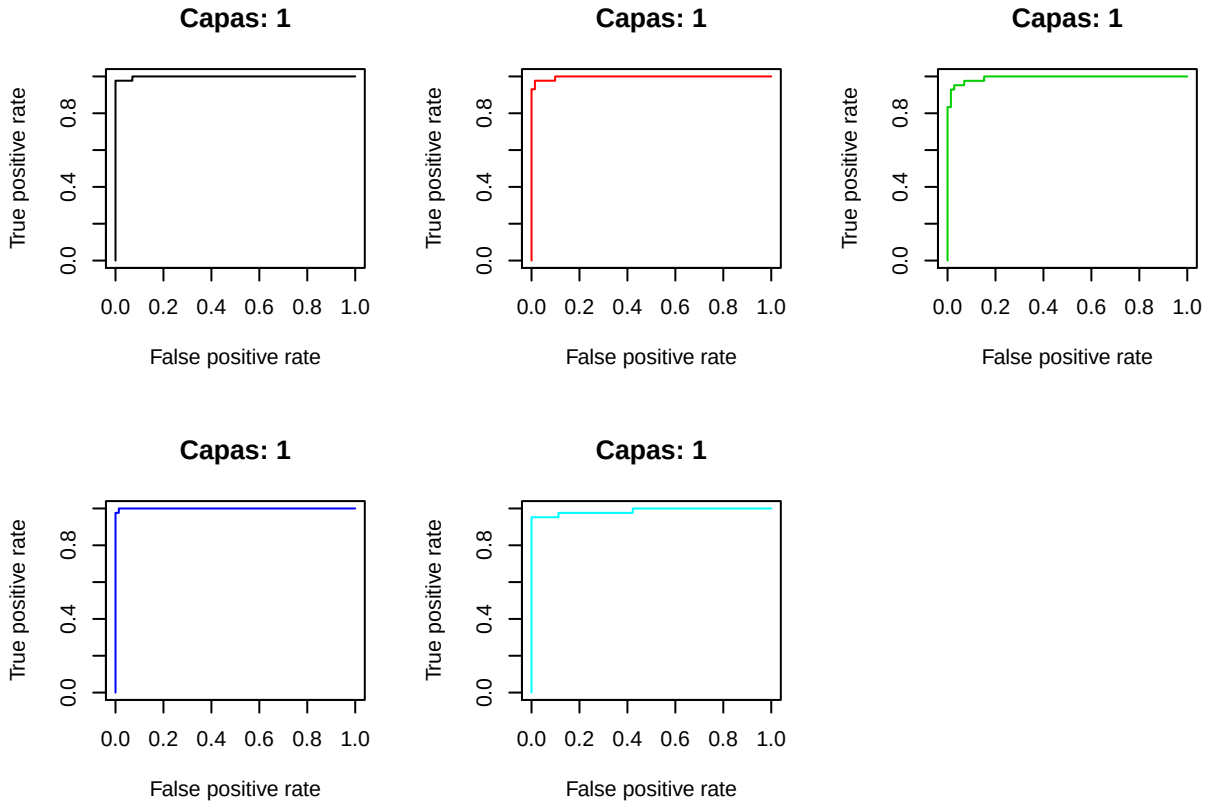
Red Neuronal

Para redes neuronales vamos a usar la función *neuralnet*. Esta función recibe como argumentos la fórmula del modelo que queremos ajustar, el conjunto de datos para aprender el modelo, un tercer argumento denominado *hidden* y un cuarto *linear.output*. El argumento *hidden* es un vector donde el tamaño del mismo indica el número de capas ocultas y el valor del indice i indica el número de unidades ocultas de la capa i. Por ejemplo, si el vector que le pasamos es (1, 2, 10) tendríamos una red neuronal con 3 capas ocultas donde la primera tiene 1 unidad, la segunda tiene 2 y la tercera tiene 10. El argumento *linear.output* está puesto a false para active la función de activación de las neuronas e indique que estamos ante una tarea de clasificación y no de regresión. Como aclaración, la función *neuralnet* no permite que introduzcamos la fórmula con la sintáxis tradicional. Por ello está la sentencia $f = as.formulas(paste(\dots))$ que permite crear de manera automática la fórmula que requiere nuestro modelo. La fórmulas que se crea es de la forma $clase \sim caracteristica\ 1 + \dots + caracteristica\ n$ para cada característica del conjunto de datos que sea distinta *clase*.

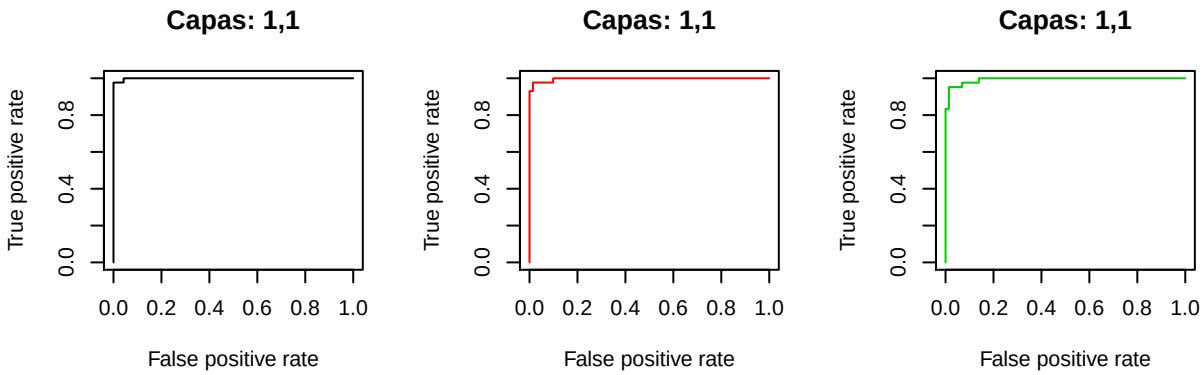
```
RedNeuronal = function(capas_ocultas = c(1), plot = F) {
  set.seed(3)
  par(mfrow=c(2,3))
  suppressWarnings(suppressMessages(library("neuralnet")))
  areas = NULL
```

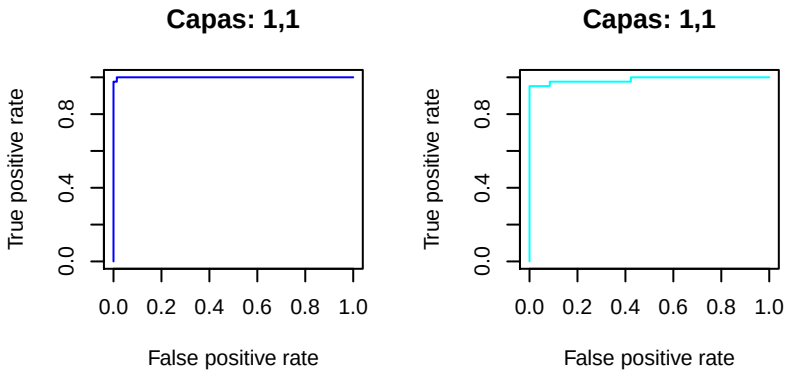
```r
for(i in 1:5){
  # Preprocesamos los datos
  train = eval(as.name(paste0("train_", i)))
  test = eval(as.name(paste0("test_", i)))
  clase = train[,1]
  train = train[,-1]
  # No hacemos YeoJohnson porque da un warning por problemas de convergencia
  # "Convergence failure: return code = 52"
  obj = preProcess(train, method = c("center", "scale", "pca"), thresh=0.95)
  train = predict(obj, train)
  train = as.data.frame(cbind(clase, train))
  clase = test[,1]
  test = test[,-1]
  test = predict(obj, test)
  test = as.data.frame(cbind(clase, test))

  # Usamos una fórmula hecha a mano ya que la función neuralnet no admite
  # la sintaxis usada hasta ahora (y ~ x)
  f = as.formula(paste("clase ~", paste(names(train)[!names(train) %in% "clase"],
                                        collapse = " + ")))

  # Estimamos el modelo
  # El argumento linear.output = F indica que queremos que se use la
  # función de activación y que no queremos hacer regresión sino clasificación.
  nnModel = neuralnet(f, data=train, hidden=capas_ocultas, linear.output=F)

  # Obtenemos la curca ROC
  predict = compute(nnModel, test[,-1])
  # La función prediction está en ROCR y en neuralnet
  # así que da error tener los dos paquetes cargados a
  # la vez, así que tenemos que desactivar los dos y cargar
  # ROCR de nuevo
  detach("package:neuralnet", unload=TRUE)

  suppressWarnings(suppressMessages(library("ROCR")))
  prediction = prediction(predict$net.result, test$clase)
  rend = performance(prediction,"tpr","fpr")
  detach("package:ROCR", unload=TRUE)
  suppressWarnings(suppressMessages(library("neuralnet")))

  # Y el area debajo de la curva ROC
  area = trapz(unlist(rend@x.values), unlist(rend@y.values))
  # Guardamos el area
  areas[i] = area

  if(plot == T){
    plot(unlist(rend@x.values), unlist(rend@y.values), type = "l",
         xlab = "False positive rate", ylab = "True positive rate", col = i,
         main = paste("Capas:", paste(capas_ocultas, collapse = ",")))
  }
}

areas
```

```
}

area_1_1 = RedNeuronal(capas_ocultas = c(1), plot = T)
area_2_1 = RedNeuronal(capas_ocultas = c(1, 1), plot = T)
```

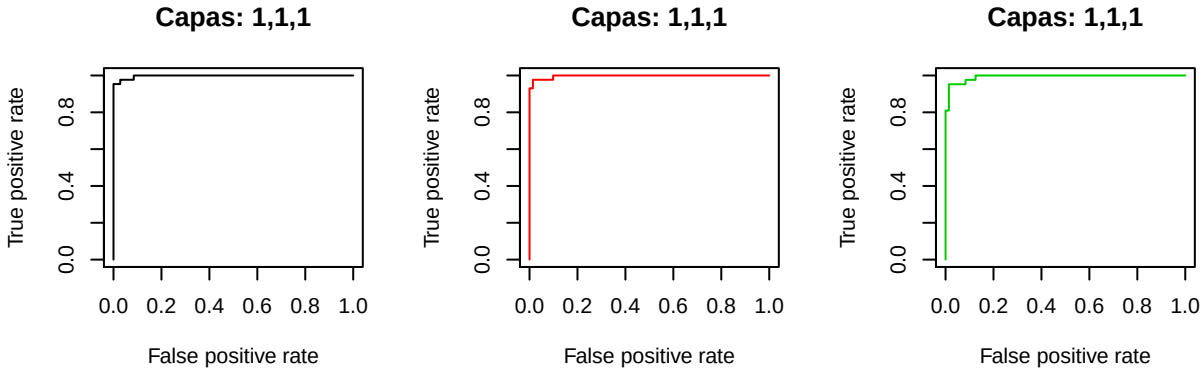


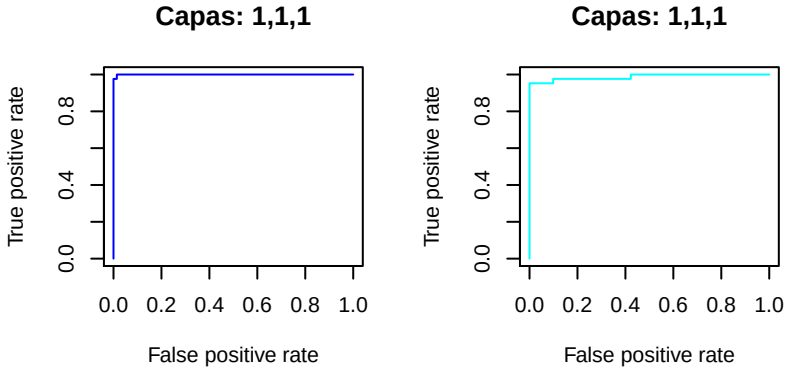

```
area_3_1 = RedNeuronal(capas_ocultas = c(1, 1, 1), plot = T)
```



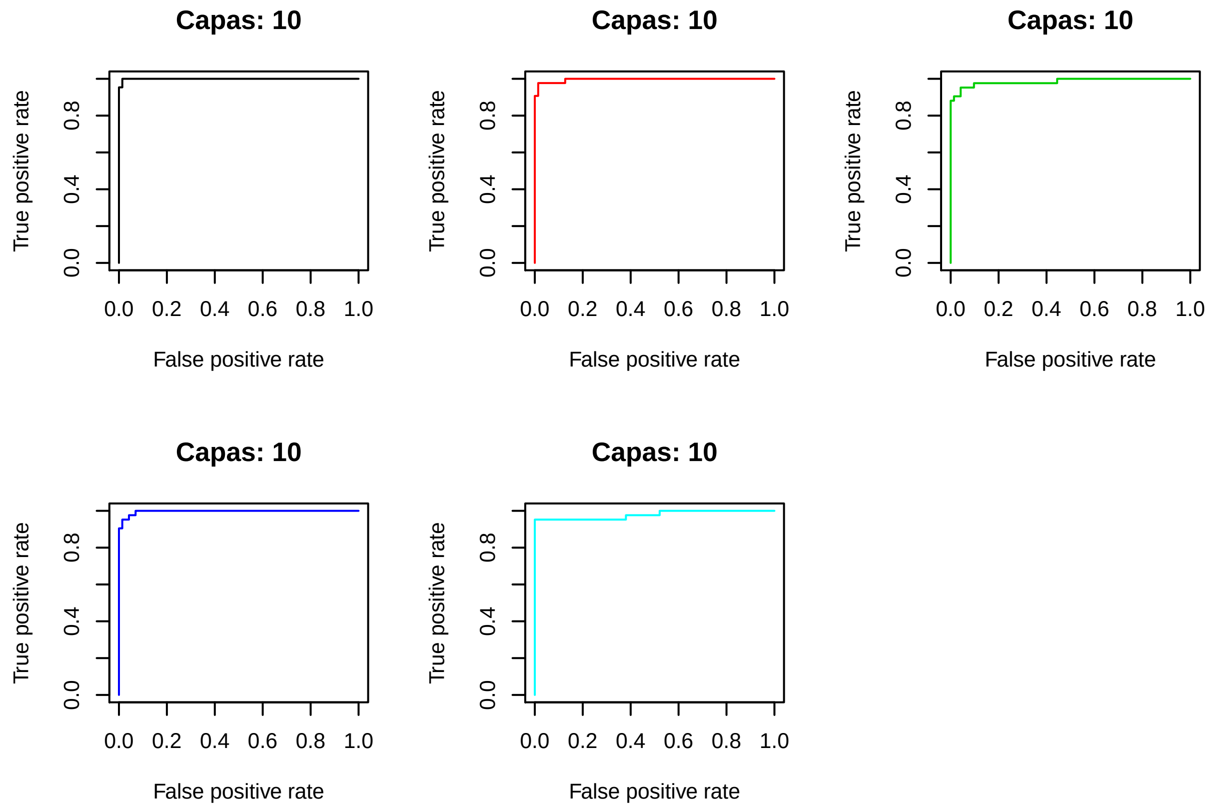



```
area_1_10 = RedNeuronal(capas_ocultas = c(10), plot = T)
```

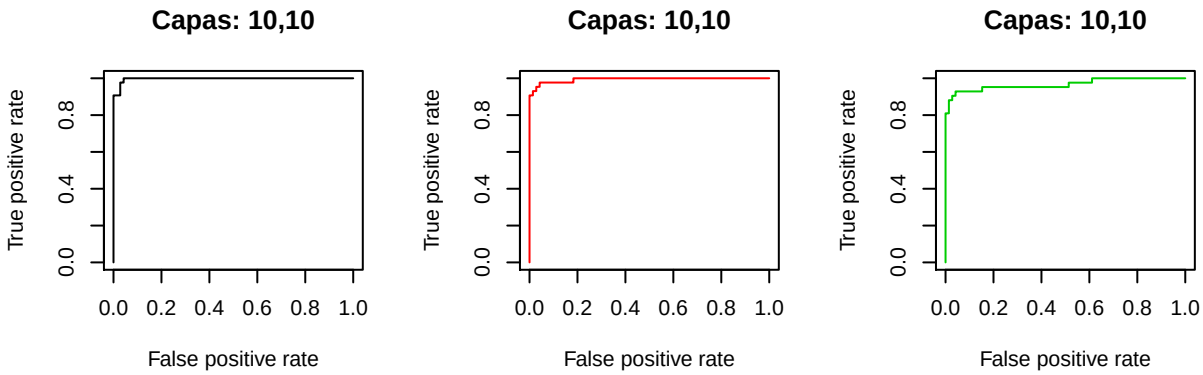


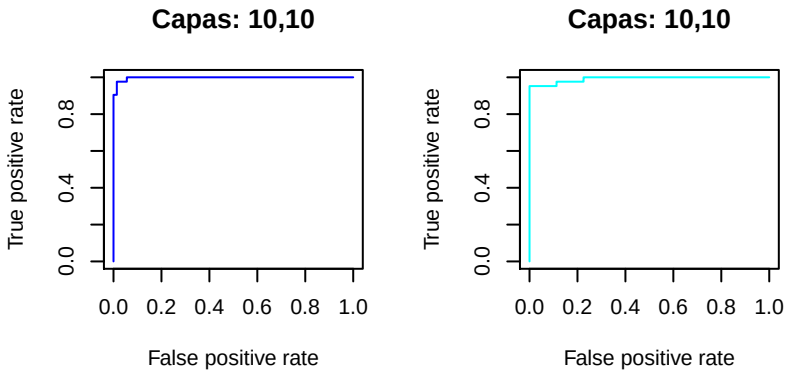


```r
area_2_10 = RedNeuronal(capas_ocultas = c(10, 10), plot = T)
```

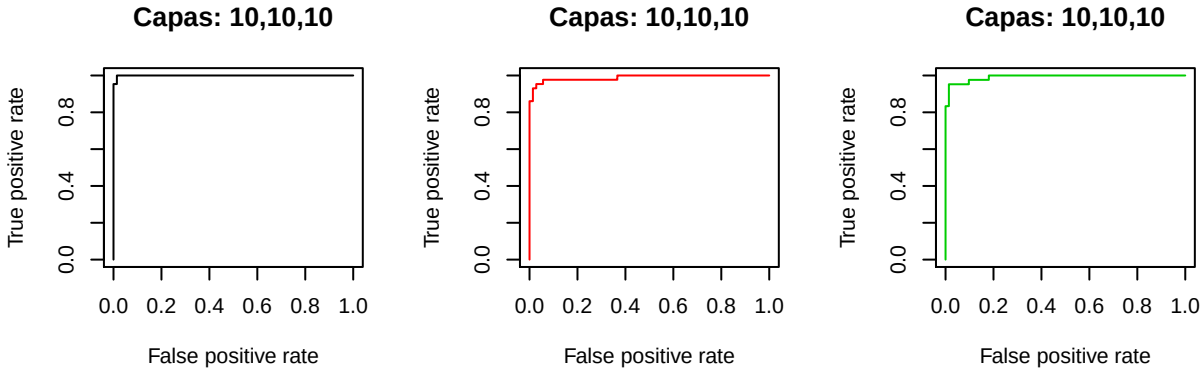


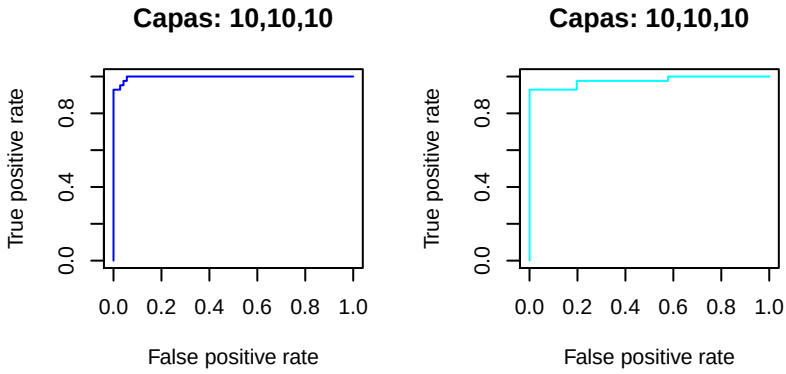

```r
area_3_10 = RedNeuronal(capas_ocultas = c(10, 10, 10), plot = T)
```



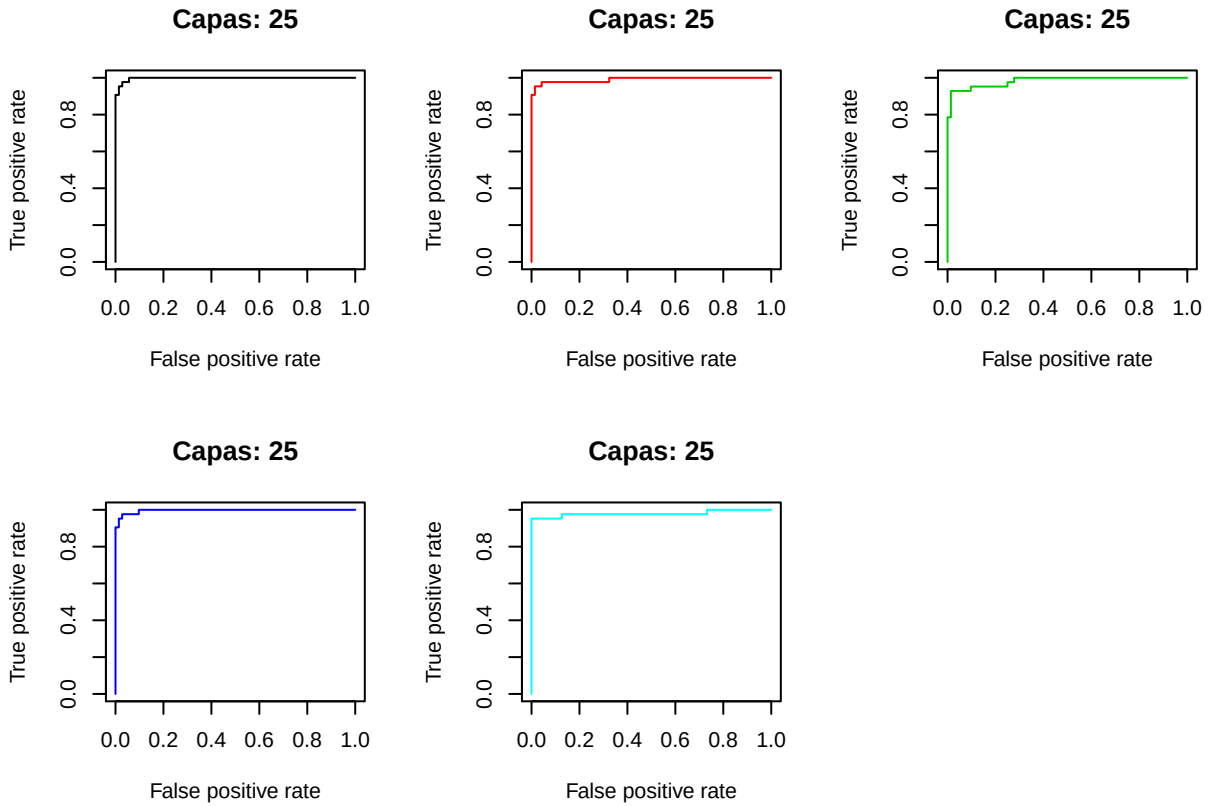



```
area_1_25 = RedNeuronal(capas_ocultas = c(25), plot = T)
```

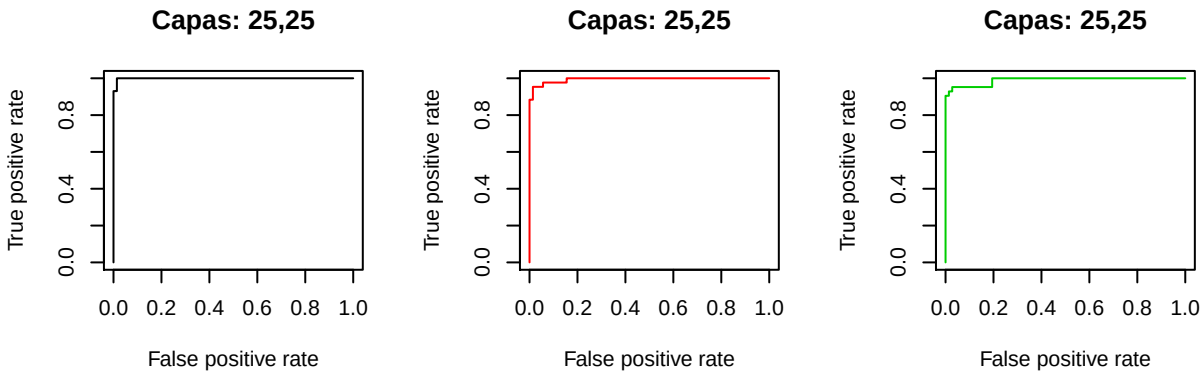


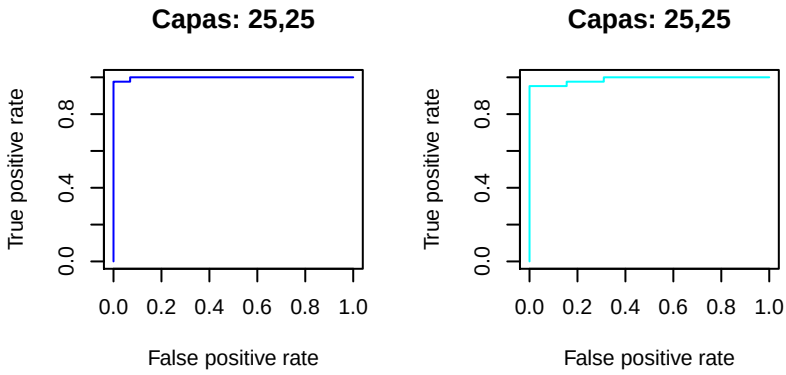


```
area_2_25 = RedNeuronal(capas_ocultas = c(25, 25), plot = T)
```

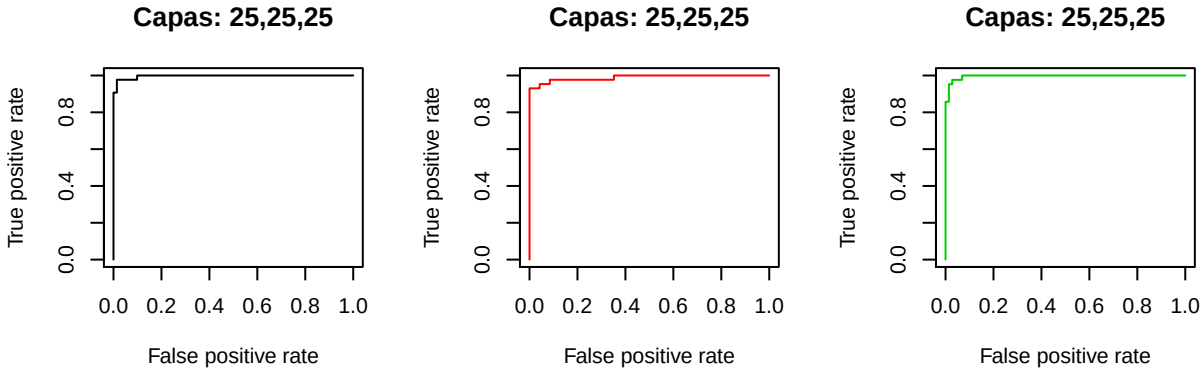


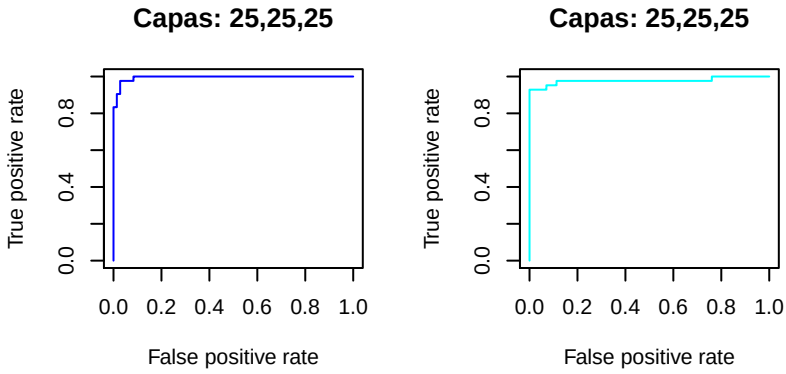

```
area_3_25 = RedNeuronal(capas_ocultas = c(25, 25, 25), plot = T)
```



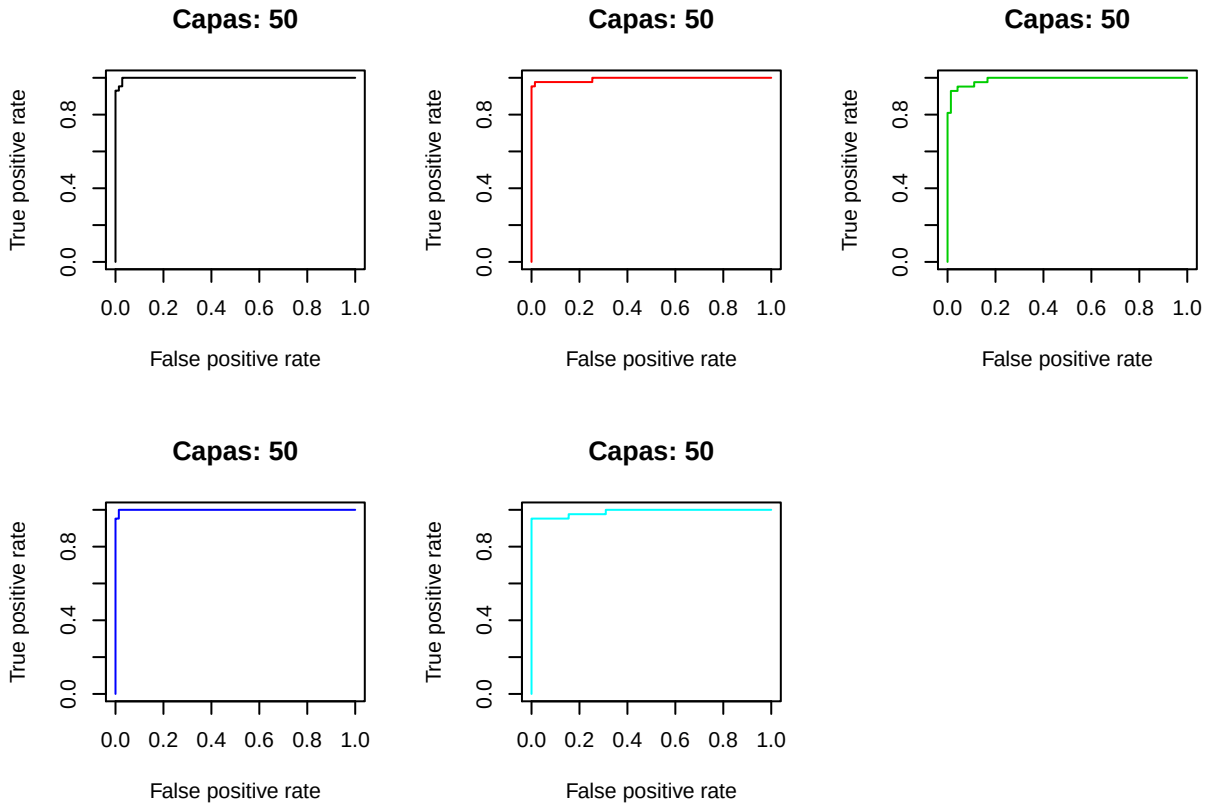



```
area_1_50 = RedNeuronal(capas_ocultas = c(50), plot = T)
```

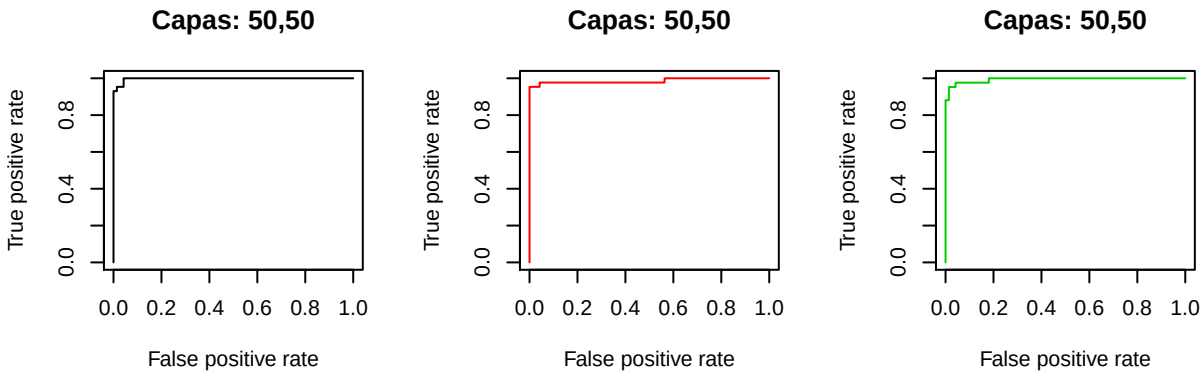


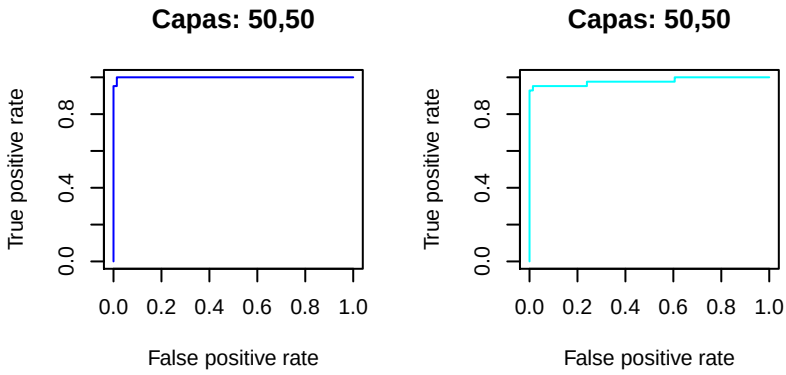


```
area_2_50 = RedNeuronal(capas_ocultas = c(50, 50), plot = T)
```

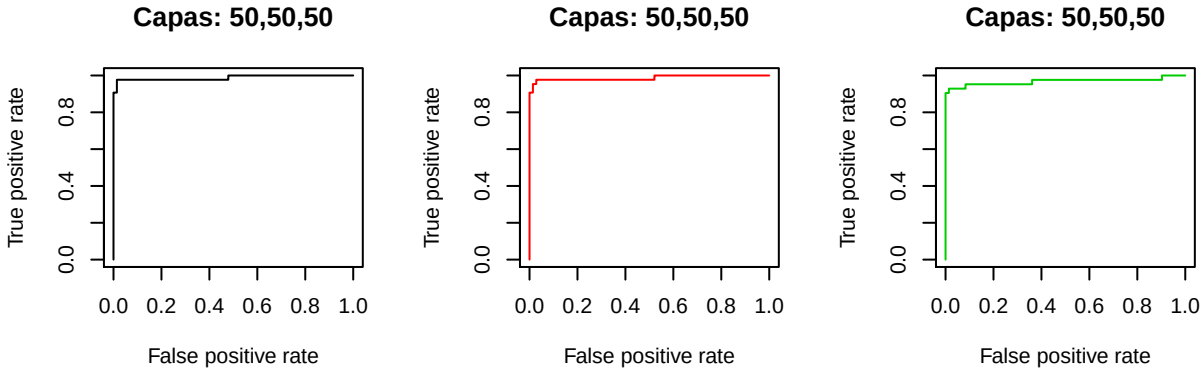


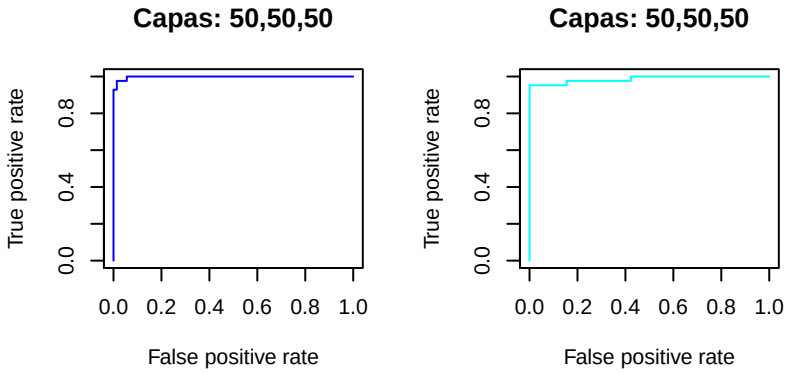

```
area_3_50 = RedNeuronal(capas_ocultas = c(50, 50, 50), plot = T)
```



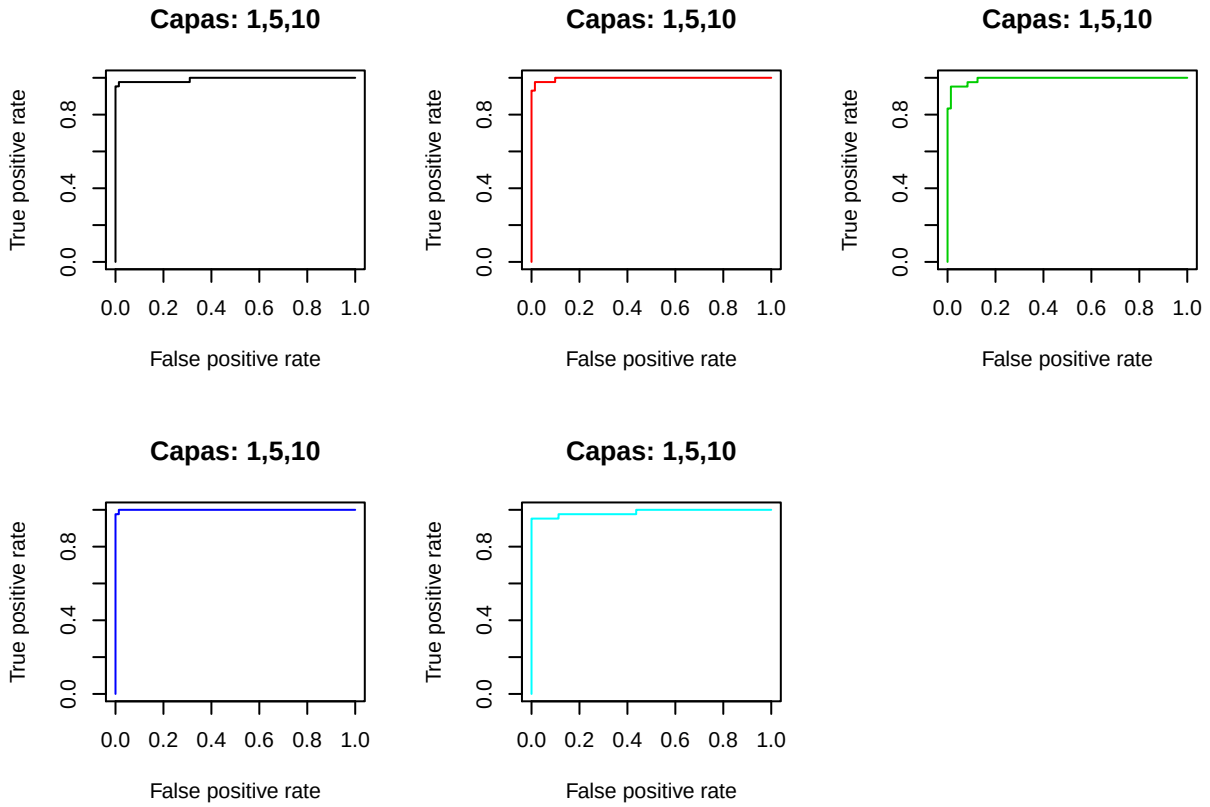



```
area_3_1_5_10 = RedNeuronal(capas_ocultas = c(1, 5, 10), plot = T)
```

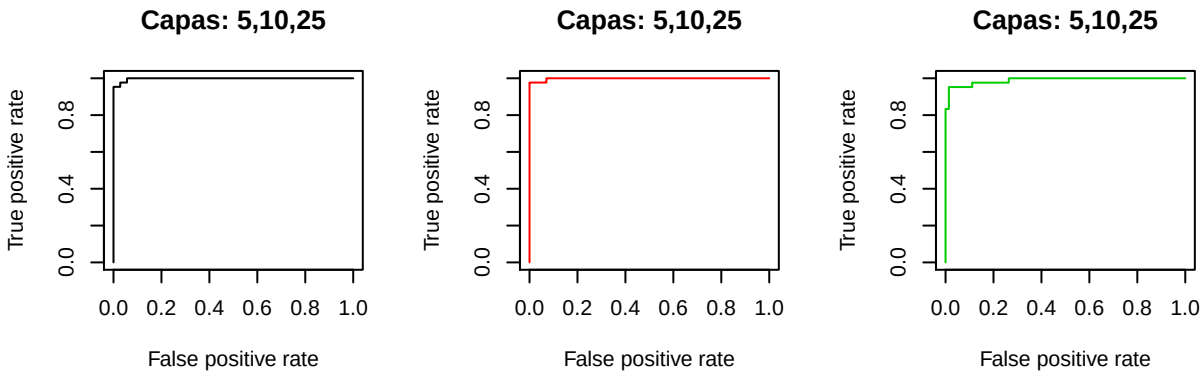


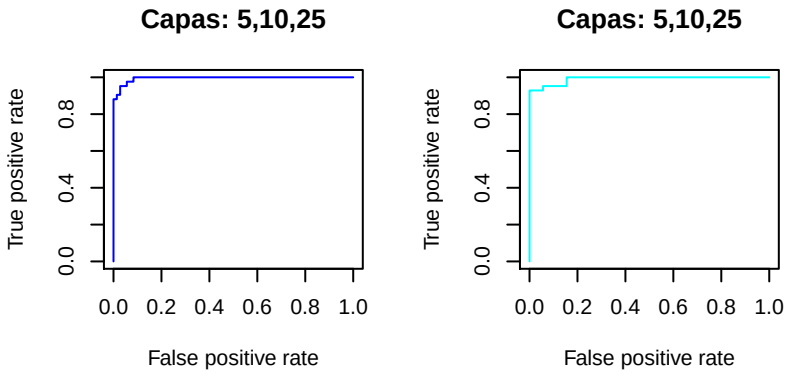


```
area_3_5_10_25 = RedNeuronal(capas_ocultas = c(5, 10, 25), plot = T)
```

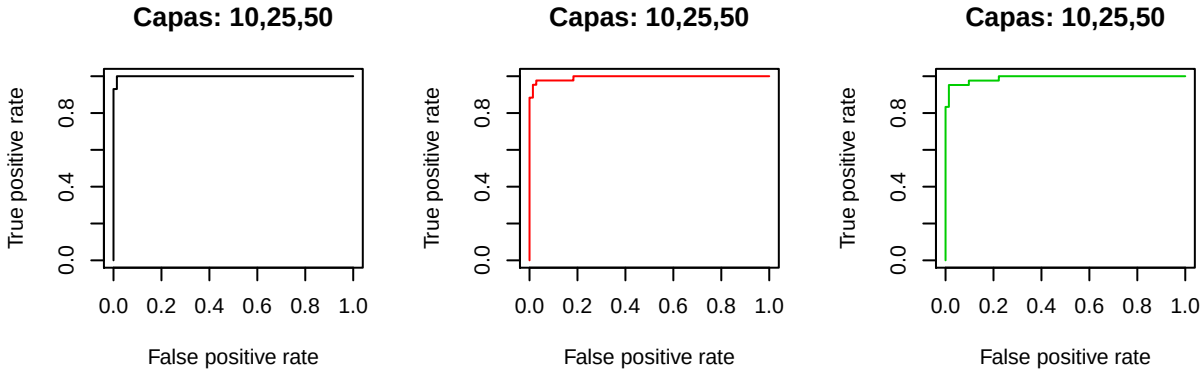


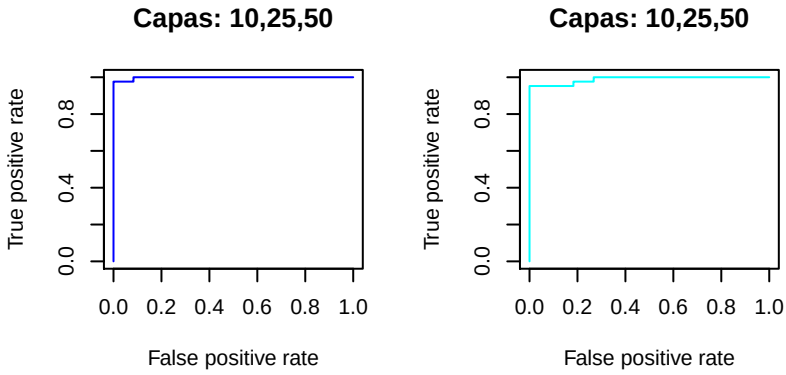

```
area_3_10_25_50 = RedNeuronal(capas_ocultas = c(10, 25, 50), plot = T)
```






```
detach("package:neuralnet", unload=TRUE)
suppressWarnings(suppressMessages(library("ROCR")))
```





Hemos visto que se puede inicializar el vector de pesos y que en las transparencias de teoría se nos indica como hacerlo. Lo hemos probado, generando un valor de la distribución gausiana para cada característica. En nuestro caso, tras aplicar PCA a nuestros datos nos quedamos con 10 características más la clase, así que hemos generado 11 valores. Al pasarle este vector a la función *neuralnet* en el argumento *startweights* salta un warning indicando que faltan pesos y que se generaran aleatoriamente. Tras usar la función *browser()* de R e inspeccionar el código de la función *neuralnet* hemos visto que dicha función está esperando un vector de 13 pesos (dos más que el número de columnas de nuestro dataset). Esto nos genera el problema de que no sabemos como inicializar los dos pesos restantes y, por lo tanto, hemos decidido dejar que *neuralnet* inicialice todos los pesos de manera aleatoria.

Para probar nuestro algoritmo hemos realizado distintos experimentos. Hemos probados redes neuronales con 1, 2 y 3 capas ocultas y con 1, 10, 25 y 50 unidades por capa. Los resultados los podemos ver en la siguiente tabla:

Capas	Unidades por capa	Area media bajo curva ROC
1	1	0.9950130802
2	1	0.9954105122
3	1	0.9949497588
1	10	0.9910867444
2	10	0.9894793611
3	10	0.9906997108
1	25	0.9894499209
2	25	0.9940300719
3	25	0.9909043841
1	50	0.9942303693
2	50	0.9912444304
3	50	0.9852621327
3	1, 5, y 10	0.9938991182
3	5, 10 y 25	0.9944271356
3	10, 25 y 50	0.9942294161

El mejor resultado lo obtenemos con una Red Neuronal con 2 capas ocultas y 1 unidad por cada.

Máquina de Soporte de Vectores (SVM)

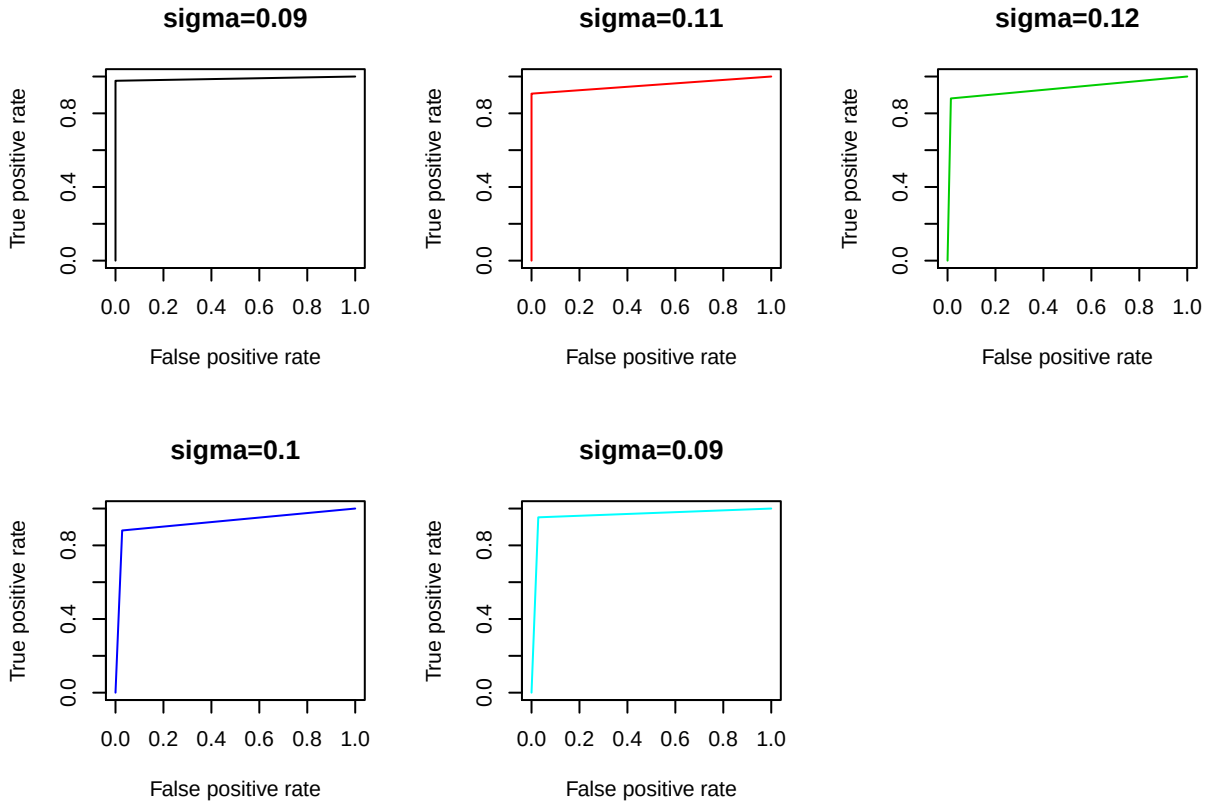
Para la máquina de soporte de vectores (SVM) vamos a usar la función *ksvm*. Esta función recibe el conjunto de datos, la fórmula a ajustar en el modelo, el núcleo usado, que en este caso es RBF-Gaussiano y un argumento llamado *kpar*, el cual sirve para pasar los parámetros que necesita el núcleo. En nuestro caso, la fórmula que usa el núcleo RBF-Gaussiano incluye un *sigma*, el cual debemos indicar. Para no condicionar los resultados usando un valor incorrecto de *sigma* hemos hecho un preproceso de estimación de parámetros. Para la estimación de parámetros hemos usado validación cruzada, por eso creamos un objeto llamaco *ctrl* con el método *trainControl*. En este objeto indicamos que queremos que se use validación cruzada y mediante los argumentos *summaryFunction* y *classProbs* indicamos que, cuando llamemos a la función *train* queremos usar ROC como la métrica para decidir cual es el mejor valor de los parámetros. Con la función *train* es con la que hacemos realmente la estimación de parámetros. Esta función recibe los datos separados (la X y la Y), el método que se desea usar, en nuestro caso *gausssprRadial* para indicar el núcleo RBF-Gaussiano, la métrica para comparar los resultados, *ROC* en nuestro caso y el objeto de control, *ctrl*, que hemos creado anteriormente. Una vez obtenemos el resultado de la función *tune*, cogemos el valor de sigma del mejor ajuste obtenido y este es el que le pasamos a la función *ksvm* en el argumento *kpar*.

```
SVM = function(plot = F) {
  set.seed(3)
```

```
  par(mfrow=c(2,3))
  suppressWarnings(suppressMessages(library("caret")))
  suppressWarnings(suppressMessages(library("kernlab")))
  areas = NULL

  for(i in 1:5){
    # Preprocesamos los datos
    train = eval(as.name(paste0("train_", i)))
    test = eval(as.name(paste0("test_", i)))
    clase = as.factor(train[,1])
    train = train[,-1]
    # No hacemos YeoJohnson porque da un warning por problemas de convergencia
    # "Convergence failure: return code = 52"
    obj = preProcess(train, method = c("center", "scale", "pca"), thresh=0.95)
    train = predict(obj, train)
    train = as.data.frame(cbind(clase, train))
    clase = as.factor(test[,1])
    test = test[,-1]
    test = predict(obj, test)
    test = as.data.frame(cbind(clase, test))

    # Estimamos los parámetros usando validación cruzada
    ctrl = trainControl(method="cv", summaryFunction=twoClassSummary, classProbs=T)
    tune = train(x=train[,-1], y = make.names(as.factor(train$clase)),
                 method = "gausssprRadial", metric="ROC", trControl=ctrl)
    bestSigma = round(tune$bestTune[1,1], 2)

    # Estimamos el modelo
    # rbfdot: núcleo RBF-Gaussiano
    svmModelo = ksvm(train$clase ~ ., data=train, kernel="rbfdot",
                     kpar=list(sigma=bestSigma))

    # Obtenemos la curca ROC
    predict = predict(svmModelo, as.data.frame(test[,-1]), type="response")
    prediction = prediction(as.numeric(predict), as.numeric(test$clase))
    rend = performance(prediction,"tpr","fpr")

    # Y el area debajo de la curva ROC
    area = trapz(unlist(rend@x.values), unlist(rend@y.values))
    # Guardamos el area
    areas[i] = area

    if(plot == T){
      plot(unlist(rend@x.values), unlist(rend@y.values), type = "l",
           xlab = "False positive rate", ylab = "True positive rate",
           col = i, main = paste0("sigma=", bestSigma))
    }
  }

  areas
}

areaSVM = SVM(plot = T)
```

```
cat("Area media bajo la curva ROC usando SVM:", mean(areaSVM), "\n")
```

```
## Area media bajo la curva ROC usando SVM: 0.9528170964
```



Boosting

Para boosting vamos a usar la función *ada*, la cual implementa el algoritmo *AdaBoost*. Esta función recibe el conjunto de datos, la fórmula a ajustar en el modelo, el número de iteraciones a realizar, un parámetro llamado *nu* que representa un concepto similar a la tasa de aprendizaje de técnicas como puede ser el Gradiente Descendente y un objeto de control que comentaremos más adelante. Al igual que hemos hecho con SVM, hemos realizado una estimación de parámetros para encontrar el mejor número de iteraciones y el mejor valor para *nu*. El objeto *ctrl* creado con la función *trainControl* es el mismo y la llamada a la función *train* también, solo se ha cambiado el método a usar para usar Adaboost en vez de usar SVM. Una vez obtenemos el resultado de la función *tune*, cogemos el valor de *nu* y las iteraciones del mejor ajuste obtenido y éste es el que le pasamos a la función *ada*.

En cuanto al objeto de control que hemos mencionado con anterioridad sirve para indicar ciertos parámetros para los árboles que va creando AdaBoost. En concreto, como queremos usar stumps, la profundidad máxima de los arboles debe ser 1, ya que con stumps la decisión es fácil; positivo o negativo, y por eso sólo necesitamos un nivel de profundidad. El parámetro *cp* indica el grado de complejidad. Con este parámetro indicamos que si ninguna división mejora los resultados por encima de *cp*, será podada mediante validación cruzada. El parámetro *minsplit* representa el número mínimo de casos que debe haber en un nodo para que se realice una división. El parámetro *xval* indica el número de validaciones cruzadas a realizar. El valor de estos parámetros es el indicado por la documentación de la función *ada* para realizar stumps.

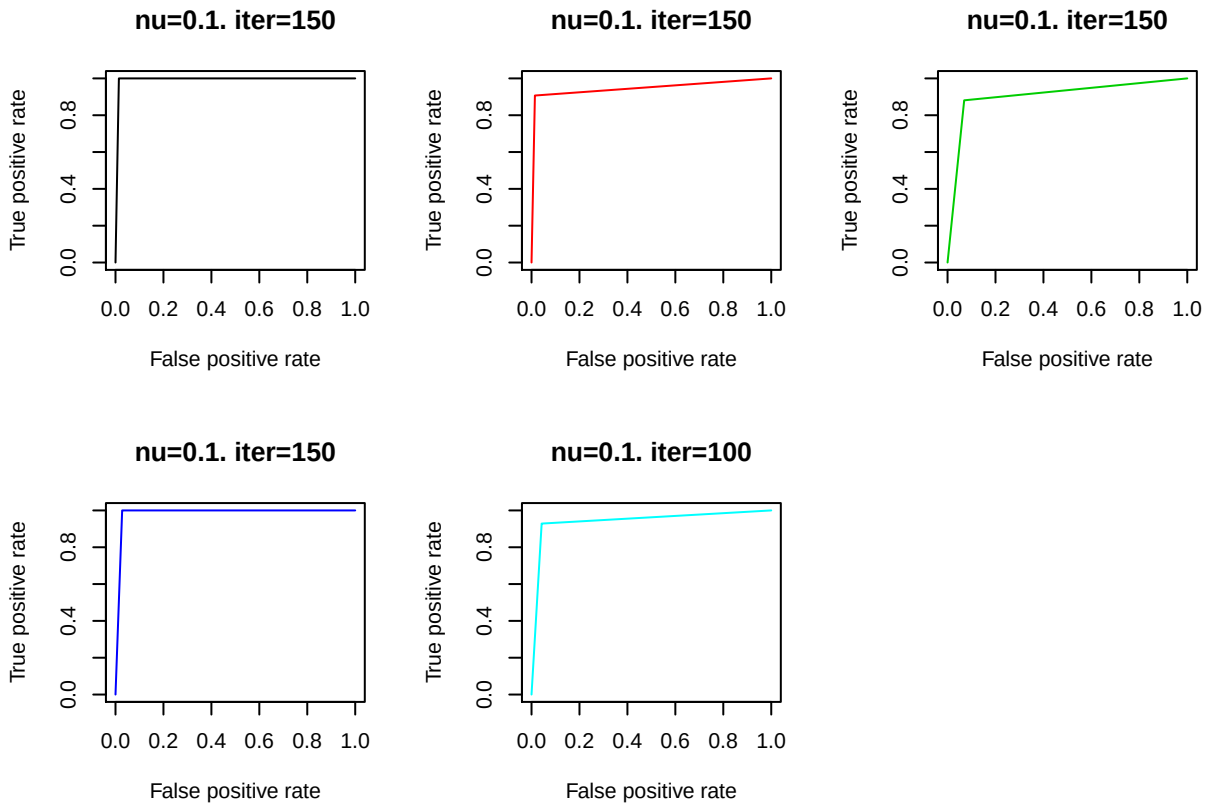
```
Boosting = function(plot = F) {
  set.seed(3)
```

```
  par(mfrow=c(2,3))
  suppressWarnings(suppressMessages(library("ada")))
  areas = NULL

  for(i in 1:5){
    # Preprocesamos los datos
    train = eval(as.name(paste0("train_", i)))
    test = eval(as.name(paste0("test_", i)))
    clase = as.factor(train[,1])
    train = train[,-1]
    # No hacemos YeoJohnson porque da un warning por problemas de convergencia
    # "Convergence failure: return code = 52"
    obj = preProcess(train, method = c("center", "scale", "pca"), thresh=0.95)
    train = predict(obj, train)
    train = as.data.frame(cbind(clase, train))
    clase = as.factor(test[,1])
    test = test[,-1]
    test = predict(obj, test)
    test = as.data.frame(cbind(clase, test))

    # Estimamos los parámetros usando validación cruzada
    ctrl = trainControl(method="cv", summaryFunction=twoClassSummary, classProbs=T)
    tune = train(x=train[,-1], y = make.names(as.factor(train$clase)), method = "ada",
                 metric="ROC", trControl=ctrl)

    # Estimamos el modelo
    boostingModelo = ada(train$clase ~ ., data=train, iter=tune$bestTune$iter,
                         nu=tune$bestTune$nu, type="discrete",
                         control = rpart.control(maxdepth=1, cp=-1,
                                                 minsplit = 0, xval = 0))

    # Obtenemos la curca ROC
    predict = predict(boostingModelo, as.data.frame(test[,-1]))
    prediction = prediction(as.numeric(predict), as.numeric(test$clase))
    rend = performance(prediction,"tpr","fpr")

    # Y el area debajo de la curva ROC
    area = trapz(unlist(rend@x.values), unlist(rend@y.values))
    # Guardamos el area
    areas[i] = area

    if(plot == T){
      plot(unlist(rend@x.values), unlist(rend@y.values), type = "l",
           xlab = "False positive rate", ylab = "True positive rate",
           col = i, main = paste0("nu=", tune$bestTune$nu,
                                  ". iter=", tune$bestTune$iter))
    }
  }

  areas
}

areaBoosting = Boosting(plot = T)
```

```
## Loading required package: plyr
```

```
cat("Area media bajo la curva ROC usando Boosting:", mean(areaBoosting), "\n")
```

```
## Area media bajo la curva ROC usando Boosting: 0.9548855796
```



Random Forest

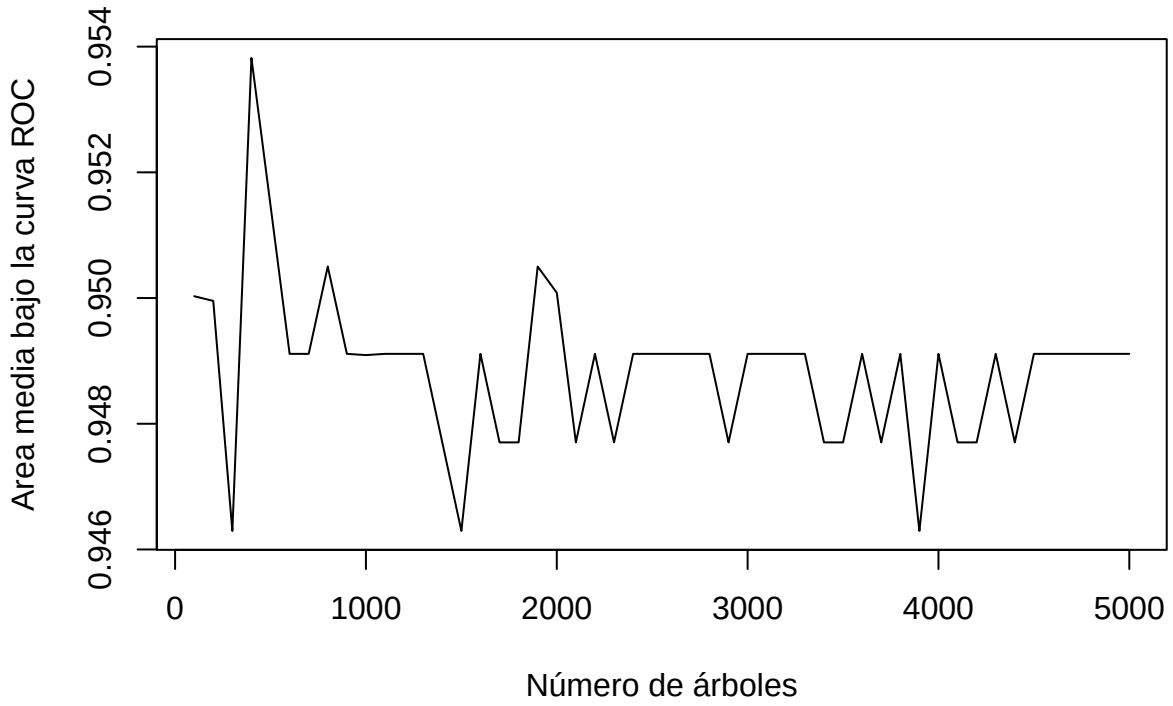
Para Random Forest vamos a usar la función *RandomForest*. Esta función recibe el conjunto de datos, la fórmula a ajustar en el modelo y un parámetro llamado *mtry* que representa el número de características (cogidas de manera aleatoria) que se van a usar en cada división. Tal y como nos dice el enunciado, debemos usar los parámetros vistos en teoría, por lo tanto el número de características a usar en cada división es la raiz cuadrada del número total de características. En cuanto al número de árboles vamos a realizar un experimento a continuación para determinat el mejor núemor de árboles.

```
RandomForest = function(n_trees = 500, plot = F) {
  set.seed(3)
  par(mfrow=c(2,3))
  suppressWarnings(suppressMessages(library("randomForest")))
  areas = NULL

  for(i in 1:5){
    # Preprocesamos los datos
    train = eval(as.name(paste0("train_", i)))
    test = eval(as.name(paste0("test_", i)))
    clase = as.factor(train[,1])
    train = train[,-1]
    # No hacemos YeoJohnson porque da un warning por problemas de convergencia
```

```
    # "Convergence failure: return code = 52"
    obj = preProcess(train, method = c("center", "scale", "pca"), thresh=0.95)
    train = predict(obj, train)
    train = as.data.frame(cbind(clase, train))
    clase = as.factor(test[,1])
    test = test[,-1]
    test = predict(obj, test)
    test = as.data.frame(cbind(clase, test))

    # Estimamos el modelo
    rfModel = randomForest(train$clase ~ ., data = train, mtry =
                             round(sqrt(dim(train)[2]-1)), ntree = n_trees)

    # Obtenemos la curca ROC
    predict = predict(rfModel, as.data.frame(test[,-1]), type="response")
    prediction = prediction(as.numeric(predict), as.numeric(test$clase))
    rend = performance(prediction,"tpr","fpr")

    # Y el area debajo de la curva ROC
    area = trapz(unlist(rend@x.values), unlist(rend@y.values))
    # Guardamos el area
    areas[i] = area

    if(plot == T){
      plot(unlist(rend@x.values), unlist(rend@y.values), type = "l",
           xlab = "False positive rate", ylab = "True positive rate",
           col = i, main = paste0("RF con ntree=", n_trees))
    }
  }

  areas
}
```

Para ver cual es el número de árboles óptimo hemos realizado el siguiente experimento. Hemos llamado a nuestra función *RandomForest* con distintos valores para el número de árboles (100, 200, . . . , 4800, 4900 y 5000) y hemos obtenido el número de árboles que nos da la mayor área media bajo la curva ROC.

```
resultados = matrix(NA, ncol = 2)
for(i in seq(100, 5000, 100)){
  area = RandomForest(n_trees = i, plot = F)
  resultados = rbind(resultados, c(i, mean(area)))
}
resultados = resultados[-1,]

par(mfrow=c(1,1))
plot(x = resultados[,1], y = resultados[,2], type = "l",
     xlab = "Número de árboles", ylab = "Area media bajo la curva ROC")
```




```
mejor = resultados[which(resultados[,2]==max(resultados[,2])),]
```

La mayor área obtenida es 0.9538185833 y la obtenemos usando 400 árboles.

Comparativa de resultados

Modelo	Area media bajo curva ROC
Modelo lineal: Regresión Lineal	0.991498929
Modelo lineal: Regresión Logística	0.9925802013
Red Neuronal: 1 capa/s oculta y 1 unidad/es por capa	0.9950130802
Red Neuronal: 2 capa/s oculta y 1 unidad/es por capa	0.9954105122
Red Neuronal: 3 capa/s oculta y 1 unidad/es por capa	0.9949497588
Red Neuronal: 1 capa/s oculta y 10 unidad/es por capa	0.9910867444
Red Neuronal: 2 capa/s oculta y 10 unidad/es por capa	0.9894793611
Red Neuronal: 3 capa/s oculta y 10 unidad/es por capa	0.9906997108
Red Neuronal: 1 capa/s oculta y 25 unidad/es por capa	0.9894499209
Red Neuronal: 2 capa/s oculta y 25 unidad/es por capa	0.9940300719
Red Neuronal: 3 capa/s oculta y 25 unidad/es por capa	0.9909043841
Red Neuronal: 1 capa/s oculta y 50 unidad/es por capa	0.9942303693
Red Neuronal: 2 capa/s oculta y 50 unidad/es por capa	0.9912444304
Red Neuronal: 3 capa/s oculta y 50 unidad/es por capa	0.9852621327
Red Neuronal: 3 capas con 1, 5, y 10 unidades por capa	0.9938991182
Red Neuronal: 3 capas con 5, 10 y 25 unidades por capa	0.9944271356
Red Neuronal: 3 capas con 10, 25 y 50 unidades por capa	0.9942294161
SVM con núcleo RBF-Gaussiano	0.9528170964
Boosting: AdaBoost con stumps	0.9548855796
Random Forest con 400 árboles	0.9538185833

A la vista de los resultados, vemos que el modelo lineal (tanto usando Regresión Lineal como Regresión Logística) presenta unos resultados muy buenos (0.991498929 y 0.9925802013 respectivamente) ya que prácticamente rozan el 1. Esto nos muestra que los datos son linealmente separables y que un modelo lineal nos permite explotar un ajuste bastante preciso. Como un modelo lineal ya ajusta los datos de manera tan precisa, es razonable pensar que tratar de ajustar los datos con modelos como lo son los modelos no lineales dé como resultado un ajuste menos preciso tal y como podemos ver en la tabla superior. En cuanto a las redes neuronales la mayoría obtienen unos mejores resultados que los modelos lineales, pero no todas lo consiguen. Por otro lado, vemos que los demás modelos no lineales no son capaces de mejorar los resultados de los modelos lineales. Esto es razonable, por ejemplo en el caso de SVM, ya que ajusta siguiendo el núcleo RBF-Gaussiano que ajusta el modelo de forma radial, por lo que ajustar unos datos separables linealmente le será más complicado (ver *figura 1*). El modelo de Boosting ajustará los datos creando un modelo robusto en función de modelos muy simples (ajustes lineales verticales u horizontales) por lo que será dificil que mejore un modelo lineal que divide los datos de manera natural. (ver *figura 2* y *figura 3*). Random Forest al ser un modelo basado en arboles de decisión también es difícil que mejore un modelo lineal cuando los datos son separables linealmente. Nuestro mejor resultado lo obtenemos usando una red neuronal con 2 capas ocultas y una única unidad por capa, lo cual nos hace ver que una red neuronal con una arquitectura bastante pequeña es capaz de entontrar buenos resultados. No obstante, todos estos modelos obtienen unos resultados muy buenos y los modelos lineales solo los superan por centésimas. Esto nos muestra que nuestros datos se ajustan de manera muy sencilla independientemente del modelo. Además estos resultados, también nos demuestran que si se obtiene un buen modelo lineal, dificilmente se va a mejorar ese modelo y de hecho si tenemos en cuenta el coste que supone obtener otros modelos más complejos, no será recomendable probar siquiera con estos.

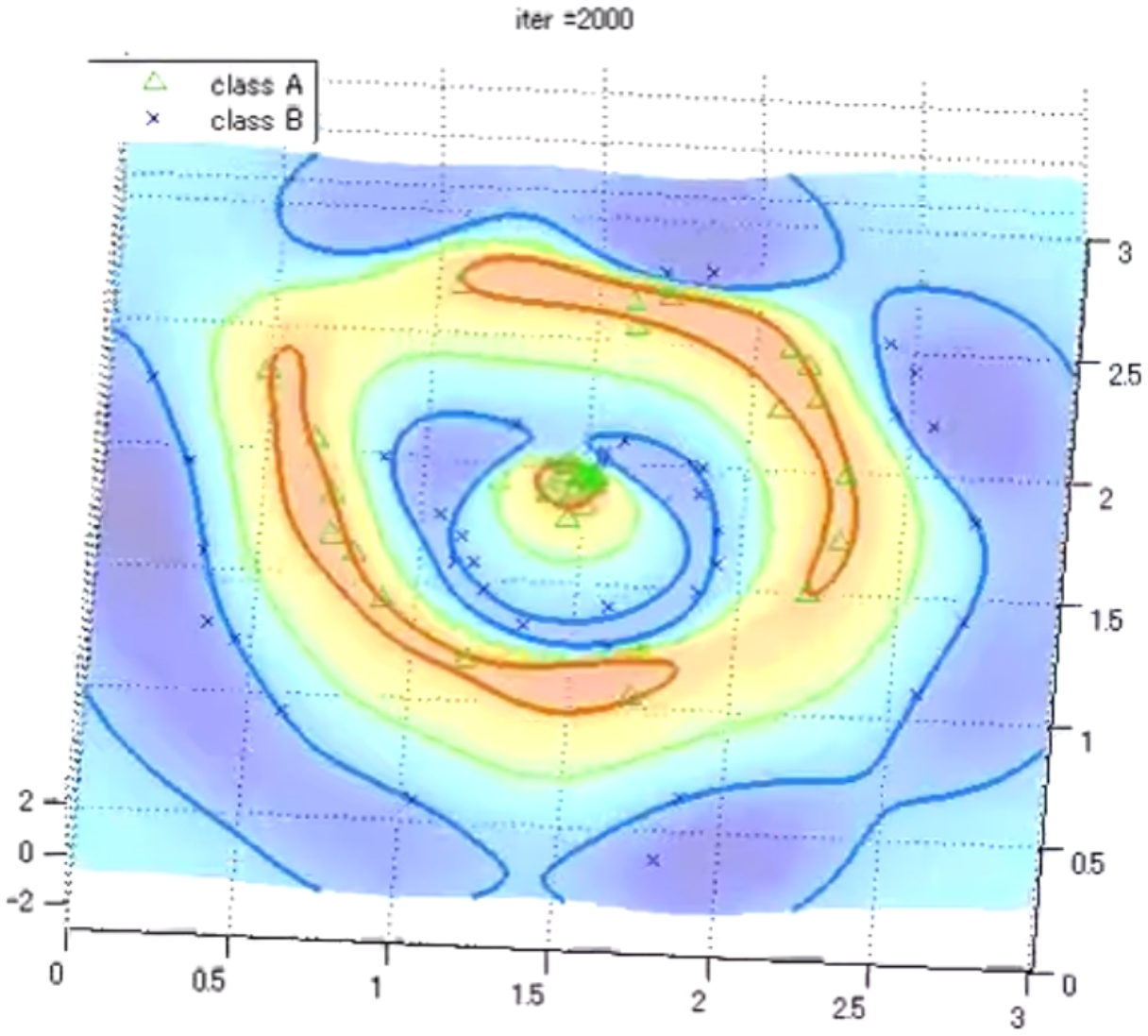


Figure 1: SVM con núcleo RBF-Gaussiano

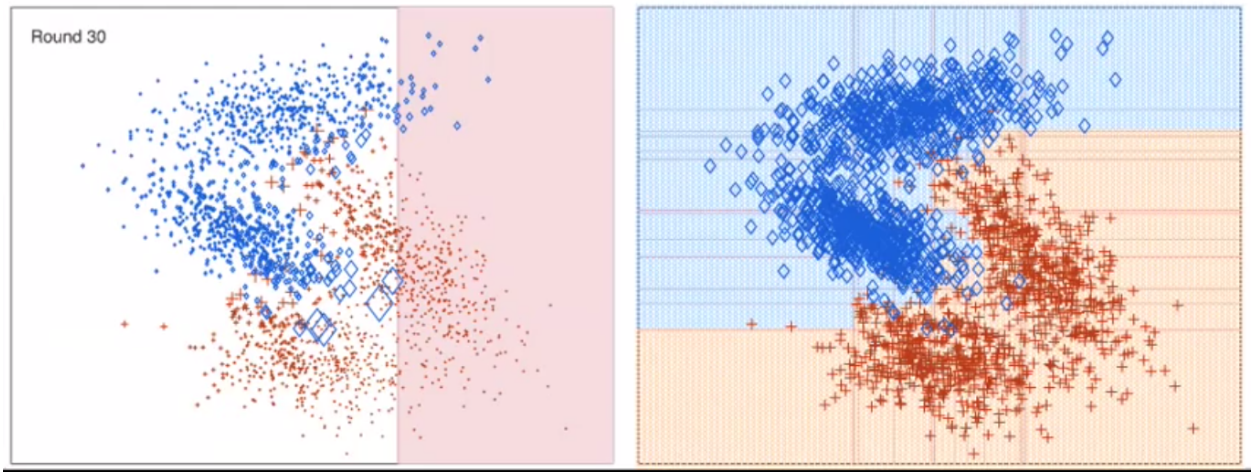


Figure 2: Algoritmo AdaBoost

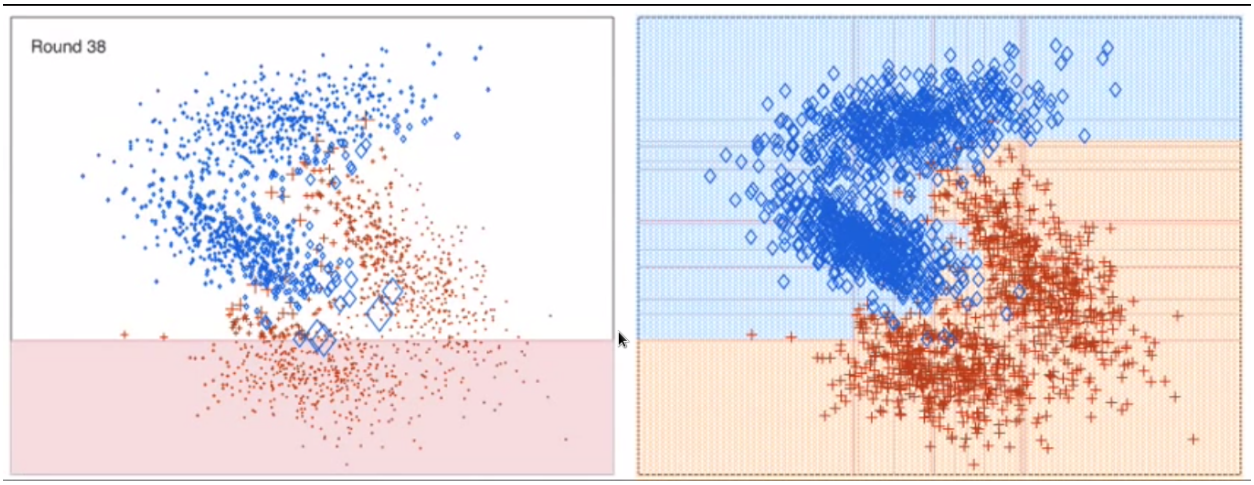


Figure 3: Algoritmo AdaBoost

Bibliografía

En cuanto a bibliografía hemos usado las documentaciones oficiales de los paquetes usados:

caret: https://cran.r-project.org/web/packages/caret/caret.pdf

ROCR: ftp://cran.r-project.org/pub/R/web/packages/ROCR/ROCR.pdf

pracma: https://cran.r-project.org/web/packages/pracma/pracma.pdf

neuralnet: ftp://oss.ustc.edu.cn/CRAN/web/packages/neuralnet/neuralnet.pdf

kernlab: https://cran.r-project.org/web/packages/kernlab/kernlab.pdf

ada: https://cran.r-project.org/web/packages/ada/ada.pdf

randomForest: https://cran.r-project.org/web/packages/randomForest/randomForest.pdf