Proyecto Aprendizaje Automático: Ajuste de Modelos No-Lineales

Germán González Almagro, Antonio Manuel Milán Jiménez

6 de junio de 2017

Índice

1. Descripción del problema

Nuestro problema consiste en encontrar el mejor predictor (lineal o no-lineal) para la base de datos "Breast Cancer Wisconsin (Diagnostic)". Se trata de un problema de clasificación en el que, a partir de datos como el radio de la célula o su simetría, tendremos que clasificar a la célula en maligna o benigna. Contamos con un total de 569 instancias (células). Para cada una de ellas tenemos 10 características, repetidas 3 veces pues varían en función de la perspectiva, así que realmente serán 30 características. Además, tenemos un identificador de la célula y la variable de respuesta que será el diagnóstico para dicha célula.

2. Tratado de los datos

Como acabamos de comentar, el primer campo de nuestros datos es el identificador de la célula. Dado que este identificador no nos es relevante para encontrar el predictor, decidimos quitarle de los datos para un tratado de éstos más sencillo.

Ahora tenemos como primer campo la variable de respuesta, el diagnóstico de la célula. Este campo contiene una "B" si la célula es benigna y una "M" si la célula es maligna, así que decidimos convertir este campos a unos y ceros respectivamente para poder tratar los datos de una manera más sencilla. Tambíen separamos este campo de los datos para poder realizar correctamente el preprocesado aunque despues volveremos a incorporarselo.

```
#Cargamos la base de datos
BCW <- read.table("Data/wdbc.data", sep=',')

#Quitamos la primera columna
BCW <- BCW[,-1]

#Guardamos las etiquetas
BCW_labels <- as.character(BCW[,1])

#Transformamos las etiquetas para que sea un problema de clasificacion binaria. Benigno=1 y Maligno=0
BCW_labels[BCW_labels=='B'] =1
BCW_labels[BCW_labels=='M'] =0
BCW_labels <- as.numeric(BCW_labels)
#Quitamos las etiquetas de nuestros datos
BCW <- BCW[,-1]
```

3. Conjuntos de datos de entrenamiento y test.

Dado que la base de datos proporcionada no está dividida en conjuntos de entrenamiento y test, decidimos nosotros mismos dividirla. Así, tomamos aleatoriamente un 70 % de la base de datos como conjunto de entrenamiento, siendo el restante 30 % el conjunto de test.

```
#Escogemos el conjunto de entrenamiento
train_set = sample(1:nrow(BCW), nrow(BCW)*0.7, replace = F)
```

4. Preprocesado de los datos.

Antes de obtener un clasificador, se hace necesario aplicar transformaciones en los datos para obtener de estos la mayor cantidad de información útil posible, dado que, en caso contrario, el clasificador que obtendremos

podría estar considerando irregularidades en los datos que hagan que éste no se comporte de la manera esperada. En esta ocasión aplicaremos a los datos las siguientes transformaciones:

- **Transformación de Yeo-Johnson:** de manera muy similar a la transformación de Box-Cox, esta transformación actúa sobre la asimetría de los datos de forma que la disminuye para permitir un trato homogéneo de todos los atributos, con la salvedad de que la transformación de Yeo-Johnson es capaz de considerar atributos con media 0 o inferior, al contrario que la transformación de Box-Cox. En esta ocasión, aunque no hay atributos que presenten media negativa, sí hay atributos cuya media es 0 o muy cercana a 0.
- **Centrado:** consiste en calcular la media de los valores de cada atributo y restarla a cada uno de ellos en particular, de forma que la nueva media obtenida tras la transformación será 0. El objetivo de esta transformación es reducir la distancia entre los diferentes atributos.
- **Escalado:** consiste en dividir los valores de los atributos por la desviación típica de los mismos, de forma que se uniformiza el rango de los atributos para evitar favorecer los atributos con mayor rango en los modelos que tienen esta característica en cuenta. De esta forma trataremos los atributos independientemente de la escala en la que estos fueron medidos.
- **Análisis de componentes principales:** consiste en seleccionar, de entre todos los atributos, aquellos que representan la mayor varianza de los datos, es decir, aquellos que proporcionan más información sobre el conjunto de datos. Deberemos tener en cuenta que este proceso es de tipo no supervisado, es decir, no tiene en cuenta la variable de respuesta, y por tanto, es posible que perdamos información relevante para la predicción de la misma.

Cabe destacar que, tras aplicar a los datos las transformaciones descritas, perdemos cualquier tipo de interpretación que pudiéramos hacer sobre los mismos.

Para aplicar estas transformaciones haremos uso del paquete `caret`, concretamente de la función `preProcess(...)`, que permite aplicar a los datos una serie de transformaciones dadas como argumento. Como hemos comentado, el análisis de componentes principales `PCA` es susceptible de eliminar atributos significativos para el aprendizaje del modelo, por ello, obtendremos dos conjuntos de datos, de forma que, sólo a uno se le ha aplicado `PCA`.

```
#Preprocesamiento de los datos: Método de Yeo Johnson, centrado, escalado y análisis de
#componentes principales.
PreProccesPCA = preProcess(BCW[train_set, ], thres=0.9,
                           method = c("YeoJohnson", "center", "scale", "pca"))
#Preprocesamiento de los datos: Método de Yeo Johnson, centrado, y escalado.
PreProccesNoPCA = preProcess(BCW[train_set, ], thres=0.9,
                             method = c("YeoJohnson", "center", "scale"))


#Aplicamos la fórmula obtenida a los datos
BCW_PreProcPCA = predict(PreProccesPCA, BCW)
BCW_PreProcNoPCA = predict(PreProccesNoPCA, BCW)

#Restauramos las etiquetas en los datos para poder especificarlas como
#variable de respuesta en la regresion lineal
BCW_PreProcPCA = cbind(BCW_PreProcPCA, BCW_labels)
BCW_PreProcNoPCA = cbind(BCW_PreProcNoPCA, BCW_labels)
```

5. Selección de la clase de funciones a utilizar.

Dado que no conocemos el mejor modelo que ajusta los datos, es lógico comenzar el estudio empleando los modelos más simples, es decir, los modelos lineales. Así pues, trataremos de obtener un clasificador mediante modelos lineales, utilizando para ello la clase de funciones lineales.

Una vez obtenido el modelo lineal y estudiando su comportamiento, lo utilizaremos como base para comparar modelos no lineales. Los modelos no-lineales que utilizaremos serán: Redes Neuronales, Máquina de Soporte de Vectores, Boosting y Random Forest.

6. Aproximación mediante modelos lineales.

Las técnicas que utilizaremos para obtener los modelos lineales son regresión lineal y regresión logística haciendo uso de la función `glm(...)` del paquete `stats`. Esta función recibe como parámetro una fórmula en la que debemos especificar la variable de respuesta y las variables que se tendrán en cuenta para obtener una clasificador para la misma. Además, deberemos especificar la familia de funciones que se empleará para obtener el error a minimizar, de forma que especificando como familia las gaussianas estaremos calculando regresión lineal y especificando la familia de binomiales estaremos calculando regresión logística.

Para la selección del subconjunto de variables que consideraremos en el aprendizaje emplearemos la función `regsubsets(...)` del paquete `leaps`, que nos proporciona información sobre los mejores subconjuntos de caraterísticas a utilizar dependiendo del número de ellas que consideremos. Esta función necesita como argumento, entre otros, un método de exploración del conjunto de atributos para proporcionar información sobre ellos; en este caso utilizaremos el método `forward` para los conjuntos de datos a los que no se les ha aplicado `PCA`, dado que el número de caracteristicas consideradas no permite realizar una búsqueda exhaustiva sobre los atributos en tiempo razonable. Al contrario sucede con los conjuntos de datos preprocesados con `PCA`, sobre los que aplicaremos búsqueda exhaustiva para la selección de subconjuntos de características, para ello especificaremos el método `exhaustive`.

Además, puesto que sabemos que al aumentar el número de variables empleadas para la obtención del clasificador, el error proporcionado por el mismo tiende a 0, y sabemos que no siempre escoger el mayor número de variables es la mejor opción, emplearemos un modelo que permita seleccionar el conjunto óptimo de variables teniendo en cuenta los dos factores anteriormente descritos. Para ello emplearemos el método *Akaike information criterion*, o C_p, que introduce una penalización en el error según el número de características empleadas para la obtención del modelo como sigue:

$$C_p = \frac{1}{n}(RSS + 2d\widehat{\sigma}^2)$$

Donde n es el número de variables asociadas a cada predictor, $\widehat{\sigma}$ es un estimador del error asociado a las variables de respuesta, d es el número total de predictores y RSS es la suma residual de errores cuadrados, que es una medida similar a la del error cuadrático, con la salvedad de que no se calcula la media del error total: $RSS = (e_1^2 + e_2^2 + +e_n^2)$. De esta forma tendremos en cuenta el número de características empleadas para la obtención del predictor a la hora de elegir el subconjunto de las mismas.

Para automatizar el proceso que elige el mejor conjunto de características con las que hallar el predictor, así como la obtención del mismo, implementamos las funciones `GetBestModel(...)` y `ProcessClassifModel(...)`, que procesan conjuntos de características y fórmulas para obtener un modelo de clasificación respectivamente:

```
ProcessClassifModel <- function(form, Data, Train_index, RVM_Index,
                                Family, f_valley, CalcErr){

  #Obtenemos el modelo en base a la fórmula dada como argumento
  model = glm(as.formula(form) , data = Data, subset = Train_index, family = Family)
```

```
  #Obtenemos las predicciones para los conjuntos de train y test
  model_predict_test = predict(model, Data[-Train_index, -RVM_Index], type = "response")
  model_predict_train = predict(model, Data[Train_index, -RVM_Index], type = "response")
  #Calculamos las etiquetas en base a la predicción
  calculated_labels_test = rep(0, length(model_predict_test))
  calculated_labels_test[model_predict_test >= 0.5] = 1
  calculated_labels_train = rep(0, length(model_predict_train))
  calculated_labels_train[model_predict_train >= 0.5] = 1
  #Calculamos el error mediante la matriz de confusión
  error_test = CalcErr(calculated_labels_test, Data[-Train_index, RVM_Index])
  error_train = CalcErr(calculated_labels_train, Data[Train_index, RVM_Index])

  #Obtenemos la curva de ROC
  pred = prediction(model_predict_test, Data[-Train_index, RVM_Index])
  perfArea = performance(pred, "auc")
  perfCurva = performance(pred, "tpr", "fpr")
  #Devolvemos los errores y los parámetros utilizados para obtenerlos,
  #además de la curva de ROC y el valor del área bajo ella
  list(E_train = error_train, E_test = error_test, formula = form, modelo = model,
       ROC = perfCurva, AreaROC = perfArea@y.values, FValley = f_valley)

} #Fin de ProcessClassifModel

GetBestModel <- function(ResponseVarName, Data, Train_index, Method,
                         Nvmax, ProcessModel, Family, CalcErr = calcularErrorCuadratico){

  if(class(ProcessModel) != "function"){
    print("Error: ProcessModel debe ser una funcion que procese un modelo")
    return(-1)
  }

  #Obtenemos la fórmula que incuye la variable de respuesta
  # y las demás variables como predictores.
  RVM_Formula = paste(ResponseVarName, "~.")
  #Obtenemos la columna en la que se encuentra la variable de respuesta
  RVM_Index = which(colnames(Data) == ResponseVarName)
  #Obtenemos lo subconjuntos de mejores características
  #dentro del conjunto de datos de entrada
  subsets = regsubsets(as.formula(RVM_Formula), data = Data[Train_index, ],
                       method = Method, nvmax = Nvmax, really.big = T)

  summ = summary(subsets)
  matrix_summ = summary(subsets)$which[ , -1]
  #Obtenemos el primer valle de la lista proporcionada por C_p
  valleys = which(diff(sign(diff(summ$cp)), na.pad = FALSE) > 0)
  first_valley = ifelse(length(valleys) == 0, length(summ$cp), valleys[1] + 1)
  var_set = matrix_summ[first_valley, ]
  #Obtenemos los resultados utilizando como subconjunto de variables
  #el indicado por el primer valle de la lista proporcionada por C_p
  no_format_set = which(var_set)
  names_list = names(no_format_set)
  form = paste(names_list, collapse = '+')
  form = paste(ResponseVarName, "~",form, collapse = '')
```

```
  #Obtenemos y procesamos el modelo mediante la función dada como argumento
  ProcessModel(form, Data, Train_index, RVM_Index, Family, first_valley, CalcErr)
}
```

6.1. Discutir la necesidad de regularización y, en su caso, la función empleada para ello.

Estamos utilizando la clase de funciones más simples, las lineales, por lo que no parece lógico utilizar regularización, pues no es posible reducir la complejidad del modelo y únicamente nos limitaremos a modificar el modelo lineal obtenido sin la aplicación de regularización.

6.2. Selección del modelo lineal

Tal y como hemos visto, PCA es un proceso no supervisado, por tanto, existe la posibilidad de que desechemos características relevantes para nuestro problema, consideraremos 4 modelos a comparar, dos obtenidos con regresión logística y dos obtenidos con regresión lineal, incluyendo cada pareja un modelo a cuyos datos se les aplicó `PCA` previamente y otro al que no se le aplicó.

Para calcular el error asociado a nuestros modelos utilizamos la matriz de confusión, de cuyo análisis obtendremos el error necesario para poder las curvas de ROC:

```
ConfMatrixError <- function(calculated_labels, original_labels){
  #Obtenemos la matriz de confusión
  matrix_c = table(calculated_labels, original_labels)
  #Calculamos los aciertos
  p_aciertos = (matrix_c[1,1] + matrix_c[2,2])/sum(matrix_c)
  #Devolvemos los fallos
  (1 - p_aciertos)*100
}
```

Utilizamos la función `GetBestModel()` que acabamos de detallar para obtener los 4 modelos lineales:

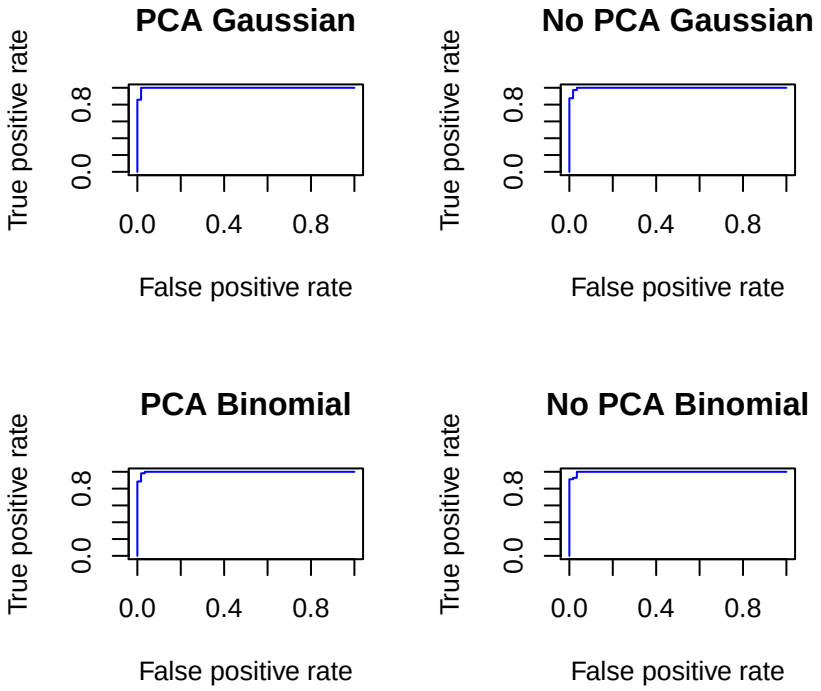
```
glmPCAGaussian = GetBestModel("BCW_labels", BCW_PreProcPCA, train_set,
                              "exhaustive", 0.8*ncol(BCW_PreProcPCA),
                              ProcessModel = ProcessClassifModel, Family = "gaussian",
                              CalcErr = ConfMatrixError)

glmNoPCAGaussian = GetBestModel("BCW_labels", BCW_PreProcNoPCA, train_set,
                                "exhaustive", as.integer(0.8*ncol(BCW_PreProcNoPCA)),
                                ProcessModel = ProcessClassifModel, Family = "gaussian",
                                CalcErr = ConfMatrixError)

glmPCABinomial = GetBestModel("BCW_labels", BCW_PreProcPCA, train_set,
                              "exhaustive", 0.8*ncol(BCW_PreProcPCA),
                              ProcessModel = ProcessClassifModel, Family = "binomial",
                              CalcErr = ConfMatrixError)

glmNoPCABinomial = GetBestModel("BCW_labels", BCW_PreProcNoPCA, train_set,
                                "exhaustive", as.integer(0.8*ncol(BCW_PreProcNoPCA)),
                                ProcessModel = ProcessClassifModel, Family = "binomial",
                                CalcErr = ConfMatrixError)
```

Utilizaremos el área bajo la curva de ROC para comparar los 4 modelos obtenidos:

	PCA Gaussian	No PCA Gaussian	PCA Binomial	No PCA Binomial
Área ROC	0.9975587	0.9974062	0.9977113	0.9972536
Número Predictores	5	10	5	10

Podemos representar las curvas de ROC obtenidas con cada modelo:

```
par(mfrow = c(2,2))
plot(glmPCAGaussian$ROC, main = "PCA Gaussian", col = "blue")
plot(glmNoPCAGaussian$ROC, main = "No PCA Gaussian", col = "blue")
plot(glmPCABinomial$ROC, main = "PCA Binomial", col = "blue")
plot(glmNoPCABinomial$ROC, main = "No PCA Binomial", col = "blue")
```



Los modelos lineales han conseguido un area bajo la curva de ROC muy cercana a 1, indicativo de que los datos siguen una distribución que permite separarlos de forma lineal sin apenas fallos.

Compararemos ahora estos modelos con los no-lineales que hemos mencionado anteriormente para saber si consiguen obtener mejores resultados o de lo contrario, un sencillo modelo lineal es suficiente para clasificar estos datos. Como representante de los modelos lineales utilizaremos el de la familia binomial con datos preprocesados con PCA, ya que es con la que mejores resultados se obtienen.

7. Aproximación mediante modelos no lineales.

Emplearemos las técnicas de Random Forest, Boosting, Support Vector Machines y Neural Networks para obtener clasificadores no lineales para nuestro conjunto de datos. Todos los modelos nombrados requieren parámetros que han de ser estimados de forma empírica, para la estimación de estos hiperparámetros emplearemos validación cruzada.

Para comparar modelos emplearemos el área bajo la curva de ROC, automatizamos el caúlo de la misma mediante la función `rocplot(...)`:

```
rocplot =function (pred , truth , plot=T, ...){

  predob = ROCR::prediction (pred , truth)
  perf = performance (predob , "tpr", "fpr")
  if(plot){plot(perf ,...)}
  as.numeric(performance(predob, "auc")@y.values)

}
```

Además, debemos crear los objetos de control para las funciones `train(...)` y `tune(...)`, que emplearemos para indicar el método a seguir par ala estimación de parámetros, a saber, validación cruzada, así como los parametros para la misma:

```
#Objeto de control para la función train: número de particiones de para vc = 10
#porcentaje de datos empleados para validar: 75%

trainControl = trainControl(method = "cv", number = 10, p = 0.75)

#Objeto de control para la función tune: número de particiones de para vc = 10
#porcentaje de datos empleados para validar: 75% (por defecto)

tuneControl = tune.control(sampling = "cross", cross = 10)
```

7.1. Random Forest

La técnica de Random Forest, similar a bagging, consiste en construir un conjunto de árboles de clasificación que no contemplan todas las carácterísticas de los ejemplos proporcionados, de esta forma los árboles resultantes no estarán correlacionados. La salida del modelo, en el caso de regresión, será la media de la etiqueta porporcionada por cada arbol para un nuevo ejemplo, y en clasificación la salida será el resultado de una votación.

En esta ocasión emplearemos $m = \sqrt{p}$, donde m es el número de predictores empleados en cada arbol y p el número de predictores totales. Estimaremos el número de árboles mediante validación cruzada, para ello emplearemos la función `tune(...)`, que recibe, entre otros, un función que calcula un modelo y una lista de posibles parámetros para dicho modelo; devolverá aquel parámetros de entre los dados como argumento que resulte en un mejor modelo.

```
#Construimos la lista de posibles parámetros par randomForest
number_trees_list = c(seq(10,100,10),seq(200,2000,100))

#Estimación de parámetros
RFTune = tune(randomForest, as.factor(BCW_labels)~.,
              data = BCW_PreProcNoPCA[train_set, ],
              mtry = sqrt(ncol(BCW_PreProcNoPCA)-1),
              # xtest = BCW_PreProcNoPCA[train_set, -ncol(BCW_PreProcNoPCA)],
              # ytest = as.factor(BCW_PreProcNoPCA[train_set, ncol(BCW_PreProcNoPCA)]),
              ranges = list(ntree=number_trees_list),
              importance = T, keep.forest = T,
              tunecontrol = tuneControl)
```

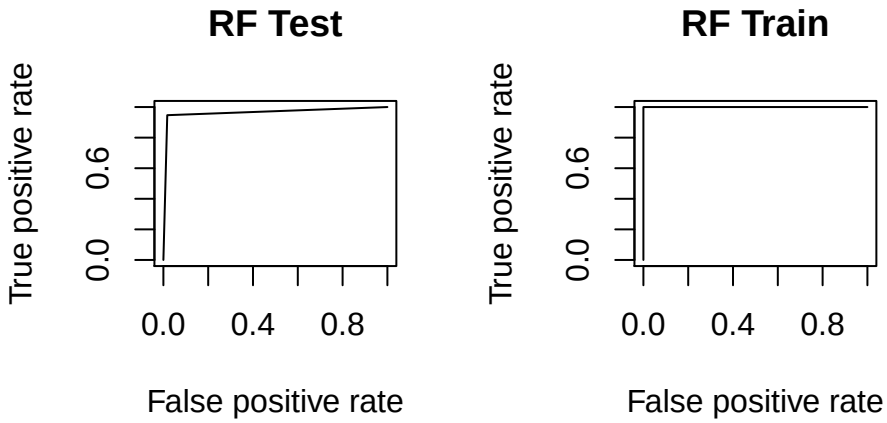
Estimado el número de árboles en 50 , obtenemos el modelo mediante la función "randomForest", a la que damos como argumento la estimación obtenida mendiante validación cruzada por `tune()`:

```
#Obtención del modelo
RFModel = randomForest(as.factor(BCW_labels)~.,
```

```
                        data = BCW_PreProcNoPCA,
                        ntree = as.numeric(RFTune$best.parameters[1]),
                        mtry = sqrt(ncol(BCW_PreProcNoPCA)-1),
                        subset = train_set, keep.forest = T, importance = T)
```

Una vez obtenido el modelo predecimos las etiquetas mediante la función `predict()` y calculamos el área bajo la curva de ROC:

```
#Estimación de las etiquetas
RF_test_labels = predict(RFModel, BCW_PreProcNoPCA[-train_set, -ncol(BCW_PreProcNoPCA)])
RF_train_labels = predict(RFModel, BCW_PreProcNoPCA[train_set, -ncol(BCW_PreProcNoPCA)])

#Obtención de resultados
par(mfrow = c(1,2))
areaRocRF_test = rocplot(as.numeric(RF_test_labels),
                         BCW_PreProcNoPCA[-train_set, ncol(BCW_PreProcNoPCA)],
                         main = "RF Test")

areaRocRF_train = rocplot(as.numeric(RF_train_labels),
                         BCW_PreProcNoPCA[train_set, ncol(BCW_PreProcNoPCA)],
                         main = "RF Train")
```



Mediante el modelo obtenemos un ajuste que resulta en un área bajo la curva de ROC para el conjunto de test de 0.9648306, mientras que la obtenida para el conjunto de train es 1.

7.2. Boosting

El método de Boosting consiste en construir un clasificador complejo a partir de clasificadores más simples de forma iterativa, asignando en cada iteración un nuevo peso a los ejemplos mal clasificados para forzar a que estos se tengan en cuenta en las sucesivas iteraciones.

En esta ocasión emplearemos el método "stumps" para construir los clasificadores simples, que consiste en costruir árboles de porfundidad una teniendo en cuenta una única característica.

Uno de los parámetros que debemos estimar para obtener el clasificador es el número de iteraciones, para ello emplearemos la función `train(...)`, que dado un método, un conjunto de ejemplos para el método y una lista de posibles valores para el parámetro a estimar, realiza validación cruzada para obtener el mejor valor del hiperparámetro de entre los proporcionados como argumento:

```
#Estimación de parámetros
SeqBy50 = seq(0,500,50); SeqBy50[1] = 1
BoostTrainGrid = matrix(c(SeqBy50, rep(1,length(SeqBy50)), rep(1,length(SeqBy50))),
```

```
                        nrow = length(SeqBy50), ncol = 3)
colnames(BoostTrainGrid) = c("iter", "maxdepth","nu")

BoostTrain = train(x=data.frame(BCW_PreProcNoPCA[train_set, -ncol(BCW_PreProcNoPCA)]),
                   y=as.factor(BCW_PreProcNoPCA[train_set, ncol(BCW_PreProcNoPCA)]),
                   method="ada",
                   tuneGrid = data.frame(BoostTrainGrid),
                   trControl = trainControl,
                   control = rpart.control(maxdepth=1, cp=-1, minsplit=0, xval=0),
                   type = "real")
```

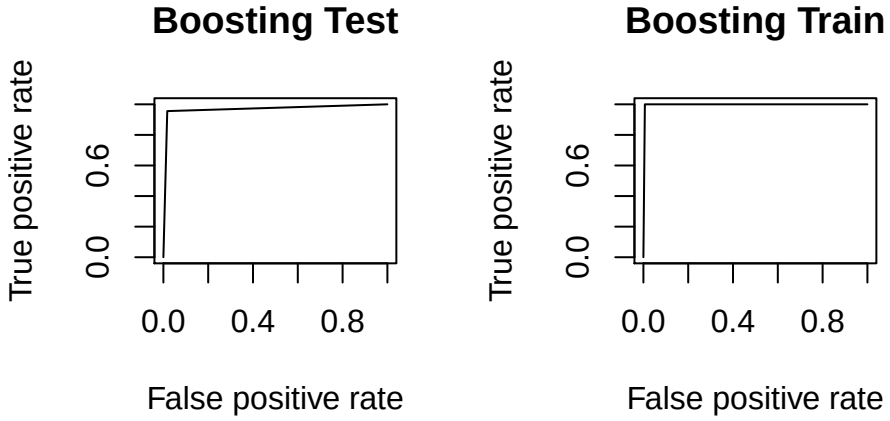
Estimado el número de iteraciones en 350 , obtenemos el modelo mediante la función `ada(...)`, que emplea la variante de boosting "AdaBoost" para obtener el clasificador, usando para ello la estimación proporcionada por `train(...)`:

```
#Obtención del modelo
BoostModel = ada(as.factor(BCW_labels)~.,
                 data = BCW_PreProcNoPCA[train_set,],
                 iter = as.numeric(BoostTrain$bestTune[1]),
                 control = rpart.control(maxdepth=1,cp=-1,minsplit=0,xval=0),
                 type = "real")
```

Con el modelo podemos predecir las etiquetas y obtener resultados sobre la bondad del ajuste calculando el área bajo la curva de ROC proporcionada por el modelo:

```
par(mfrow = c(1,2))
```

```
#Estimación de las etiquetas
Boost_test_labels = predict(BoostModel, BCW_PreProcNoPCA[-train_set, -ncol(BCW_PreProcNoPCA)])
Boost_train_labels = predict(BoostModel, BCW_PreProcNoPCA[train_set, -ncol(BCW_PreProcNoPCA)])

#Obtención de mediciones
par(mfrow = c(1,2))
areaRocBoost_test = rocplot(as.numeric(Boost_test_labels),
                            BCW_PreProcNoPCA[-train_set, ncol(BCW_PreProcNoPCA)],
                            main = "Boosting Test")

areaRocBoost_train = rocplot(as.numeric(Boost_train_labels),
                             BCW_PreProcNoPCA[train_set, ncol(BCW_PreProcNoPCA)],
                             main = "Boosting Train")
```



Mediante el modelo obtenemos un ajuste que resulta en un área de ROC para el conjunto de test de 0.9692554

y para el conjunto de train obtenemos un valor de 0.9692554.

7.3. Support Vector Machines

Las máquinas de vectores de soporte son un conjunto de algoritmos encargados de construir un hiperplano que separe a nuestro conjunto de datos con el mayor margen posible, es decir, que separando los datos, esté a la mayor distancia de ellos. Los datos que están delimitando estos márgenes se les llama vector soporte. La SVM que vamos a contruir tiene un kernel radial Gaussiano, es decir, que el hiperplano que construye sigue una función radial Gaussiana.

Una de las ventajas del núcleo RBF-Gaussiano para SVM es que son pocos los parámetros que debemos estimar, sa saber $\gamma \in [0, 1]$ y el coste de clasificar mal un ejemplo C. Estimaremos γ con una precisión de dos decimales y seleccionaremos C de entre el conjunto $\{01, 1, 10, 100, 1000\}$. Para ello emplearemos de nuevo la función `train(...)`, descrita en secciones anteriores de este documento.

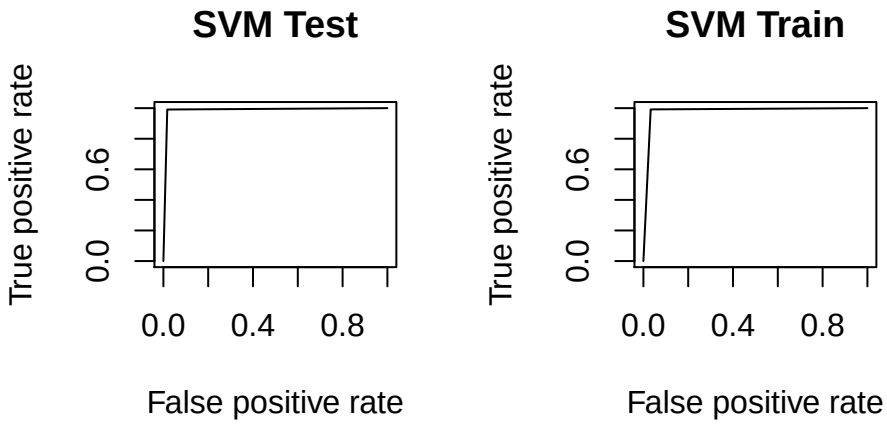
```
SVMTrainGrid = expand.grid(c(0.1,1,10,100,1000), seq(0,1,0.01))
colnames(SVMTrainGrid) = c("C", "sigma")

SVMTrain = train( x=data.frame(BCW_PreProcNoPCA[train_set, -ncol(BCW_PreProcNoPCA)]),
                  y=as.factor(BCW_PreProcNoPCA[train_set, ncol(BCW_PreProcNoPCA)]),
                  method="svmRadialSigma",
                  tuneGrid = data.frame(SVMTrainGrid),
                  trControl = trainControl)
```

Una vez obtenida la estimación para γ, a saber 0.01, y para C, a saber 1, obtenemos el modelo mediante la función `svm(...)`, especificando como parámetros los obenidos mediante validación cruzada, así como indicando "radial" como valor para el parámetro `kernel`, asociado al núcleo RBF-Gaussiano:

```
#Obtención del modelo
SVMModel = svm(as.factor(BCW_labels)~.,
               data = BCW_PreProcNoPCA[train_set, ],
               gamma = as.numeric(SVMTrain$bestTune[1]),
               cost = as.numeric(SVMTrain$bestTune[2]),
               kernel = "radial")
```

Una vez obtenido el modelo podemos predecir las etiquetas y obtener resultados sobre la bondad del ajuste calculando el área bajo la curva de ROC proporcionada por el modelo:

```
#Estimación de las etiquetas
SVM_test_labels = predict(SVMModel, BCW_PreProcNoPCA[-train_set, -ncol(BCW_PreProcNoPCA)])
SVM_train_labels = predict(SVMModel, BCW_PreProcNoPCA[train_set, -ncol(BCW_PreProcNoPCA)])

#Obtención de mediciones
par(mfrow = c(1,2))
areaRocSVM_test = rocplot(as.numeric(SVM_test_labels),
                          BCW_PreProcNoPCA[-train_set, ncol(BCW_PreProcNoPCA)],
                          main = "SVM Test")

areaRocSVM_train = rocplot(as.numeric(SVM_train_labels),
                           BCW_PreProcNoPCA[train_set, ncol(BCW_PreProcNoPCA)],
                           main = "SVM Train")
```



Empleando el clasificador obtenido mediante el modelo logramos un ajuste que resulta en un área de ROC para el conjunto de test de 0.9869545 y para el conjunto de train en un valor de 0.9796679.

7.4. Neural Networks

Las redes neuronales constituyen uno de los modelos mas potentes y flexibles para aprendizaje automático, son capaces de aproximar funciones complejas de manera eficiente, la principal desventaja de este modelo, presente también es muchos otros, es la facilidad que tiene para sobreajustar la muestra y lo sensibles que son respecto a los parámetros libres. Se basan en capas constituidas por neuronas que aportan información a la siguiente capa de neuronas de forma que la capa final esta constituida por una sola neurona que es la que reune toda la información obtenida en capas anteriores para porporcionar una etiqueta.

Del párrafo anterior concluimos que deberemos estimar dos parámetros, el número de capas ocultas y el número de unidades por capa. En esta ocasión consideraremos arquitecturas con 1, 2 y 3 capas ocultas, cada una de ellas con un número de neuronas entre 0 y 50.

De esta forma, obtendremos tres modelos, uno con 1 capa oculta, otro con 2 y un último modelo con 3 capas ocultas, y estimaremos para cada uno de ellos el número de unidades ocultas por capa, sujetos siempre a las limitaciones de capacidad de cómputo.

En cuanto a detalles de implementación comunes, debemos tener en cuenta que la función `neuralnet(...)` obtiene modelos de regresión por defecto, para obtener un modelo de clasificación deberemos especificar `linear.output = FALSE`, además no acepta como vector de respuesta un vector de factores, por tanto debemos porporcionar las etiquetas como enteros y aplicar a las predicciones una transformación para poder compararlas con las etiquetas originales.

7.4.1. Arquitectura NN con 1 capa oculta.

Comenzamos por estimar el número de neuronas para la capa oculta de esta arquitectura, emplearemos para ello la función `train(...)` y consideraremos el rango del número de neuronas de 0 a 50.

```
hiddens1 = matrix(data = c(1:50, rep(0,50), rep(0,50)), nrow = 50, ncol = 3)
colnames(hiddens1) = c("layer1", "layer2", "layer3")

trainNN1 = train(x=data.frame(BCW_PreProcNoPCA[train_set, -ncol(BCW_PreProcNoPCA)]),
                 y = BCW_PreProcNoPCA[train_set, ncol(BCW_PreProcNoPCA)],
                 method="neuralnet",
                 tuneGrid = data.frame(hiddens1),
                 trControl = trainControl,
                 linear.output = FALSE)
```

Una vez estimado el número de neuronas en 50, obtenemos el modelo mediante la función `neuralnet(...)` , dando como parámetro para el número de neuronas el estimado anteriormente:

```
#Construimos la fórmula con todos los predictores
form = paste(colnames(BCW_PreProcNoPCA)[-ncol(BCW_PreProcNoPCA)], collapse = '+')
form = as.formula(paste("BCW_labels", "~",form, collapse = ''))

#Obtenemos el modelo
NNModel1 = neuralnet(as.formula(form),data=BCW_PreProcNoPCA[train_set,],
                         hidden = as.numeric(trainNN1$bestTune[1]),
                         linear.output = FALSE)
```

Una vez obtenido el modelo podemos predecir las etiquetas y obtener resultados sobre la bondad del ajuste calculando el área bajo la curva de ROC:

```
#Estimación de las etiquetas1
Neural_test_labels1 = compute(NNModel1, BCW_PreProcNoPCA[-train_set, -ncol(BCW_PreProcNoPCA)])
Neural_train_labels1 = compute(NNModel1, BCW_PreProcNoPCA[train_set, -ncol(BCW_PreProcNoPCA)])

calculated_labels_test1 = rep(0, length(Neural_test_labels1$net.result))
calculated_labels_test1[Neural_test_labels1$net.result >= 0.5] = 1
calculated_labels_train1 = rep(0, length(Neural_train_labels1$net.result))
calculated_labels_train1[Neural_train_labels1$net.result >= 0.5] = 1

#Obtención de mediciones2
areaRocNeural_train1 = rocplot(calculated_labels_train1,
                               BCW_PreProcNoPCA[train_set, ncol(BCW_PreProcNoPCA)], F)

areaRocNeural_test1 = rocplot(calculated_labels_test1,
                              BCW_PreProcNoPCA[-train_set, ncol(BCW_PreProcNoPCA)], F)
```

7.4.2. Arquitectura NN con 2 capas ocultas.

De igual forma que en el caso anterior, comenzamos por estimar el número de neuronas para cada capa oculta, aunque esta vez, dadas las limitaciones de cómputo, reducimos el número de posibilidades para este parámetro a los múltiplos de 5 entre el 0 y el 50 y comenzando por el 1, de forma que reducimos el número de posbilidades de 2500 a 121.

```
SeqBy5 = seq(0,50,5); SeqBy5[1] = 1
hiddens2 = expand.grid(SeqBy5, SeqBy5); hiddens2 = cbind(hiddens2, rep(0,nrow(hiddens2)))
colnames(hiddens2) = c("layer1", "layer2", "layer3")

trainNN2 = train(x=data.frame(BCW_PreProcNoPCA[train_set, -ncol(BCW_PreProcNoPCA)]),
                     y = BCW_PreProcNoPCA[train_set, ncol(BCW_PreProcNoPCA)],
                     method="neuralnet",
                     tuneGrid = data.frame(hiddens2),
                     trControl = trainControl,
                     linear.output = FALSE)
```

Estimado el número de neuronas para la primera capa oculta en 35 y en 10 para la segunda, obtenemos el modelo de igual forma que en la sección anterior:

```
NNModel2 = neuralnet(as.formula(form),data=BCW_PreProcNoPCA[train_set,],
                         hidden = c(trainNN2$bestTune[1], trainNN2$bestTune[2]),
                         linear.output = FALSE)
```

Con el modelo obtenido podemos predecir las etiquetas y obtener resultados sobre la bondad del ajuste calculando el área bajo la curva de ROC:

```
#Estimación de las etiquetas2
Neural_test_labels2 = compute(NNModel2, BCW_PreProcNoPCA[-train_set, -ncol(BCW_PreProcNoPCA)])
Neural_train_labels2 = compute(NNModel2, BCW_PreProcNoPCA[train_set, -ncol(BCW_PreProcNoPCA)])

calculated_labels_test2 = rep(0, length(Neural_test_labels2$net.result))
calculated_labels_test2[Neural_test_labels2$net.result >= 0.5] = 1
calculated_labels_train2 = rep(0, length(Neural_train_labels2$net.result))
calculated_labels_train2[Neural_train_labels2$net.result >= 0.5] = 1

#Obtención de mediciones2
par(mfrow = c(1,2))
areaRocNeural_train2 = rocplot(calculated_labels_train2,
                               BCW_PreProcNoPCA[train_set, ncol(BCW_PreProcNoPCA)], F)

areaRocNeural_test2 = rocplot(calculated_labels_test2,
                              BCW_PreProcNoPCA[-train_set, ncol(BCW_PreProcNoPCA)], F)
```

7.4.3. Arquitectura NN con 3 capas ocultas.

Tal y como hacíamos en los casos anteriores, comenzamos por estimar el número de neuronas para cada capa oculta, una vez más debemos tener en cuenta las limitaciones de cómputo, de forma que debemos reducir el número de posibilidades para este parámetro a los múltiplos de 10 entre el 0 y el 50 y comenzando por el 1, de forma que reducimos el número de posbilidades de 125000 a 216.

```
SeqBy10 = seq(0,50,10); SeqBy10[1] = 1
hiddens3 = expand.grid(SeqBy10, SeqBy10, SeqBy10)
colnames(hiddens3) = c("layer1", "layer2", "layer3")

trainNN3 = train(x=data.frame(BCW_PreProcNoPCA[train_set, -ncol(BCW_PreProcNoPCA)]),
                     y = BCW_PreProcNoPCA[train_set, ncol(BCW_PreProcNoPCA)],
                     method="neuralnet",
                     tuneGrid = data.frame(hiddens3),
                     trControl = trainControl,
                     linear.output = FALSE)
```

Obtenemos una estimación para el número de neuronas para la primera, segunda y tercera capas ocultas de 50, 10 y 1 respectivamente. En base a esto obtenemos el modelo:

```
NNModel3 = neuralnet(as.formula(form),data=BCW_PreProcNoPCA[train_set,],
                         hidden = as.vector(trainNN3$bestTune),
                         linear.output = FALSE)
```

Obtenido el modelo obtenido podemos predecir las etiquetas y obtener resultados sobre la bondad del ajuste calculando el área bajo la curva de ROC:

```
#Estimación de las etiquetas3
Neural_test_labels3 = compute(NNModel3, BCW_PreProcNoPCA[-train_set, -ncol(BCW_PreProcNoPCA)])
Neural_train_labels3 = compute(NNModel3, BCW_PreProcNoPCA[train_set, -ncol(BCW_PreProcNoPCA)])

calculated_labels_test3 = rep(0, length(Neural_test_labels3$net.result))
calculated_labels_test3[Neural_test_labels3$net.result >= 0.5] = 1
calculated_labels_train3 = rep(0, length(Neural_train_labels3$net.result))
calculated_labels_train3[Neural_train_labels3$net.result >= 0.5] = 1
```

```r
#Obtención de mediciones1
par(mfrow = c(1,2))
areaRocNeural_train3 = rocplot(calculated_labels_train3,
                               BCW_PreProcNoPCA[train_set, ncol(BCW_PreProcNoPCA)], F)

areaRocNeural_test3 = rocplot(calculated_labels_test3,
                              BCW_PreProcNoPCA[-train_set, ncol(BCW_PreProcNoPCA)], F)
```

Vamos a dibujar un ejemplo de una red neuronal que hemos construido para observar su estructura. Dibujaremos una red neuronal con pocos nodos por capa para que sea más sencilla su ilustración:

```r
NNModel_ejemplo = neuralnet(as.formula(form),data=BCW_PreProcNoPCA[train_set,],
                        hidden = c(3,3,3),
                        linear.output = FALSE)

plot(NNModel_ejemplo)
```

7.5. Comparativa de resultados

Una vez obtenidos los tres modelos podemos compararlos representando en una tabla el área bajo la curva de ROC proporcionada para cada modelo para train y test:

	1 Capa Oculta	2 Capas Ocultas	3 Capas Ocultas
Área ROC Test	0.9825297528	0.9736801953	0.9736801953
Área ROC Train	0.9967532468	1	1
Unidades por capa	(50,0,0)	(35,10,0)	(50,10,1)

7.6. Estimación de E_{out}

A continuación, vamos a predecir el comportamiento de los diferentes modelos fuera de la muestra, calculando una cota para E_{out} basada en E_{test}. Utilizaremos la función `ConfMatrixError()` que ya describimos anteriormente para calcular el E_{test} de cada modelo.

Calculamos la cota basada en E_{test} aplicando:

$$P(|E_{test}(g) - E_{out}(g)| > \epsilon) \leq 2e^{-2N\epsilon^2}$$

Sabemos que la probabilidad de $P(|E_{test}(g) - E_{out}(g)| > \epsilon) = \delta = 005$, por tanto, debemos encontrar el valor de ϵ para obtener una cota. Si se cumple la anterior desigualdad entonces se cumple que $\delta \leq 2e^{-2N\epsilon^2}$, basta con despejar ϵ para poder obtener un valor para la cota:

$$\delta \leq 2e^{-2N\epsilon^2} \rightarrow ln\left(\frac{\delta}{2}\right) \leq -2N\epsilon^2 \rightarrow \sqrt{\frac{ln\left(\frac{\delta}{2}\right)}{-2N}} \geq \epsilon \rightarrow \sqrt{\frac{ln(2) - ln(\delta)}{2N}} \geq \epsilon$$

Tomaremos ϵ como el primer valor que hace cierta la desigualdad.

```r
options(digits = 2)

#Calculamos los parámetros
N_test = nrow(BCW_PreProcNoPCA) - length(train_set)
delta = 0.05
```

```r
epsilon = sqrt((log(2) - log(delta))/(N_test))

#E_test obtenido mediante el modelo lineal
#Calculamos la cota basada en E_test
BasadaE_test_lineal = glmPCABinomial$E_test + epsilon
print(c("Cota basada en E_test para el modelo lineal: ", BasadaE_test_lineal))
```

```
## [1] "Cota basada en E_test para el modelo lineal: "
## [2] "1.90126140352781"
```

```r
#E_test obtenido mediante el modelo de "random forest""
E_test_RF = ConfMatrixError(as.numeric(RF_test_labels),
                            BCW_PreProcNoPCA[-train_set, ncol(BCW_PreProcNoPCA)])
#Calculamos la cota basada en E_test
BasadaE_test_RF = E_test_RF + epsilon
print(c("Cota basada en E_test para Random Forest: ", BasadaE_test_RF))
```

```
## [1] "Cota basada en E_test para Random Forest: "
## [2] "4.24044269007752"
```

```r
#E_test obtenido mediante el modelo "Boosting"
E_test_Boosting = ConfMatrixError(as.numeric(Boost_test_labels),
                            BCW_PreProcNoPCA[-train_set, ncol(BCW_PreProcNoPCA)])
#Calculamos la cota basada en E_test
BasadaE_test_Boosting = E_test_Boosting + epsilon
print(c("Cota basada en E_test para Boosting: ", BasadaE_test_Boosting))
```

```
## [1] "Cota basada en E_test para Boosting: "
## [2] "3.65564736844009"
```

```r
#E_test obtenido mediante el modelo "SVM"
E_test_SVM = ConfMatrixError(as.numeric(SVM_test_labels),
                           BCW_PreProcNoPCA[-train_set, ncol(BCW_PreProcNoPCA)])
#Calculamos la cota basada en E_test
BasadaE_test_SVM = E_test_SVM + epsilon
print(c("Cota basada en E_test para SVM: ", BasadaE_test_SVM))
```

```
## [1] "Cota basada en E_test para SVM: " "1.31646608189038"
```

```r
#E_test obtenido mediante el modelo NN con 1 capa oculta
E_test_NN1 = ConfMatrixError(calculated_labels_test1,
                             BCW_PreProcNoPCA[-train_set, ncol(BCW_PreProcNoPCA)])
# table(calculated_labels_test1,BCW_PreProcNoPCA[-train_set, ncol(BCW_PreProcNoPCA)])
#Calculamos la cota basada en E_test
BasadaE_test_NN1 = E_test_NN1 + epsilon
print(c("Cota basada en E_test para Neural Network 1 capa oculta: ", BasadaE_test_NN1))
```

```
## [1] "Cota basada en E_test para Neural Network 1 capa oculta: "
## [2] "1.90126140352781"
```

```r
#E_test obtenido mediante el modelo NN con 2 capas ocultas
E_test_NN2 = ConfMatrixError(calculated_labels_test2,
                             BCW_PreProcNoPCA[-train_set, ncol(BCW_PreProcNoPCA)])
# table(calculated_labels_test2,BCW_PreProcNoPCA[-train_set, ncol(BCW_PreProcNoPCA)])
#Calculamos la cota basada en E_test
BasadaE_test_NN2 = E_test_NN2 + epsilon
print(c("Cota basada en E_test para Neural Network 2 capa oculta: ", BasadaE_test_NN2))
```

```
## [1] "Cota basada en E_test para Neural Network 2 capa oculta: "
## [2] "3.07085204680266"
```

```
#E_test obtenido mediante el modelo NN con 3 capas oculta
E_test_NN3 = ConfMatrixError(calculated_labels_test3,
                             BCW_PreProcNoPCA[-train_set, ncol(BCW_PreProcNoPCA)])
# table(calculated_labels_test3,BCW_PreProcNoPCA[-train_set, ncol(BCW_PreProcNoPCA)])
#Calculamos la cota basada en E_test
BasadaE_test_NN3 = E_test_NN3 + epsilon
print(c("Cota basada en E_test para Neural Network 3 capa oculta: ", BasadaE_test_NN3))
```

```
## [1] "Cota basada en E_test para Neural Network 3 capa oculta: "
## [2] "3.07085204680266"
```

```
# print(c("Err RF: ",E_test_RF))
# print(c("Err Boosting: ",E_test_Boosting))
# print(c("Err SVM: ",E_test_SVM))
# print(c("Err NN1: ",E_test_NN1))
# print(c("Err NN2: ",E_test_NN2))
# print(c("Err NN3: ",E_test_NN3))
#
# all.equal(calculated_labels_test2,calculated_labels_test1)
# all.equal(calculated_labels_test2,calculated_labels_test3)
# calculated_labels_test2
# calculated_labels_test3
```

8. Comparación y selección del modelo final

Vamos a comparar las estimaciones de E_{out} y el Área de ROC para todos los modelos:

	Lineal	Random Forest	Boosting	SVM	NN 1 capa	NN 2 capas	NN 3 capas
Error E_{test}	1.75 %	4.09 %	3.51 %	1.17 %	1.75 %	2.92 %	2.92 %
Estimación E_{out}	1.9 %	4.24 %	3.66 %	1.32 %	1.9 %	3.07 %	3.07 %
Área ROC	1	0.96	0.97	0.99	0.98	0.97	0.98

Con esta tabla, vemos que todos los modelos han obtenido unos muy buenos resultado. En todos ellos, el área de ROC es muy cercano a 1.

Sorprende que el modelo más sencillo de todos, el modelo lineal que calculamos al inicio, haya conseguido un área de 1, por encima de todos los demás modelos no lineales, mucho más complejos.

Teniendo en cuenta que el modelo lineal ha conseguido el mejor área de ROC y que por el principio de "la navaja de Ockham" debemos escoger la solución más simple, decidimos escoger como clasificador de nuestro problema el modelo lineal.

9. Referencias