Proyecto

Carlos Morales Aguilera
David Criado Ramón

9 de junio de 2017

Contents

1 DESCRIPCIÓN DEL PROBLEMA Y LECTURA DE DATOS

El problema que hemos estudiado es *Ocuppancy Detection*. En él intentamos averiguar si un hogar está ocupado o no (clasificación binaria) basándonos en características como la humedad o la luminosidad.

Para empezar creamos una función en R que nos permite leer el archivo CSV que contiene las muestras. Tras leer este archivo, quitamos cualquier muestra cuyo vector de características tenga algún valor perdido y, además, hemos considerado que la fecha y hora a la que se realiza la muestra no son relevantes a la hora de predecir los datos, por lo que lo eliminamos dicha característica del vector. Además, puesto que la muestra de entrenamiento se encuentra desbalanceada (hay aproximadamente el doble de casos negativos que positivos), utilizamos el método *Synthetic Minority Over-sampling TEchnique* para obtener nuevos casos negativos (con etiqueta negativa) artificiales y reducir el número de casos positivos.

```
# Función que lee un archivo del problema a tratar
leer = function(archivo, balancear = F) {
  # Leemos el archivo CSV
  df = read.csv(archivo,
                quote="\"", comment.char="", stringsAsFactors=FALSE)
  # Quitamos muestras con valores perdidos
  df = na.omit(df)
  # Quitamos la fecha y hora
  df = df[,-1]

  # Pasamos las etiquetas a un factor
  df$Occupancy = factor(df$Occupancy, levels = c(1,0), labels = c("SI", "NO"))

  if (balancear) {
    # Ajustamos posibles desbalances de las clases
    df = SMOTE(Occupancy ~ ., df)
  }
```

```
  # Lo pasamos a matriz
  x = apply(as.matrix.noquote(df[1:5]),2,as.numeric)

  # Separamos las etiquetas del resto de datos
  y = df$Occupancy

  # Devolvemos los datos en data frame y forma matricial separada
  list(df = df, x = x, y = y)
}

# Leemos los datos de training
training = leer("./datos/datatraining.txt", balancear = T)
test = leer("./datos/datatest2.txt")
```

2 PREPROCESAMIENTO

Una vez los datos están leídos, procedemos a preprocesarlos. Puesto que tenemos pocas variables hemos decidido no eliminar variables con alto grado de correlación lineal ni aplicar PCA, ya que obtendríamos datos no interpretables de forma innecesaria. No obstante, si que aplicamos la transformación de *Box y Cox* para corregir la asimetría que pueda existir en nuestros datos, y centramos y escalamos los datos basándonos en la muestra de training.

Podríamos pensar que al no quitar variables con alto grado de correlación obtendremos malos resultados debido a las dependencias entre variables, pero hemos tomado esta decisión ya que obteniendo menos variables aún, habría realmente pocas diferencias entre las muestras por escasez de variables con las que definirlas y separarlas posteriormente.

```
# Calculamos la transformación del preprocesamiento basándonos en training
preProc = preProcess(x = training$x,
                     method = c("BoxCox", "center", "scale"),
                     outcome = training$y,
                     )
# Función para aplicar el preprocesamiento a un subconjunto
aplicarPreprocesamiento = function(subconjunto, preProc) {
  x = predict(preProc, subconjunto$x)
  df = data.frame(x)
  df$Occupancy = subconjunto$y
  list(df = df, x = subconjunto$x, y = subconjunto$y)
}

# Aplicamos el preprocesamiento a la muestra de training
training = aplicarPreprocesamiento(training, preProc)
# Aplicamos el preprocesamiento a la muestra de test
test = aplicarPreprocesamiento(test, preProc)
```

Para ajustar todos los modelos vamos a utilizar la función **train** del paquete *caret*. Todas las muestras de entrenamiento van a realizar un proceso de validación especificado en el siguiente *trainControl*. Vamos a realizar validación cruzada con 5-folds (hemos realizado pruebas con 10-folds y el resultado es similar, por lo que reducimos para obtener mayor eficiencia en tiempo y ya que no tenemos una cantidad considerable de datos). Además guardaremos los resultados obtenidos en estas validaciones para poder estimar el error y mostrar la matriz de confusión y la curva ROC en validación.

Los parámetros pasados al *trainControl* son:

- **method="cv"** Representa que durante el entrenamiento utilizamos validación cruzada con folds para estimar el error fuera de la muestra.
- **number=5** Representa el número de folds a usar, en nuestro caso 5.
- **summaryFunction = twoClassSummary** Representa el tipo de tabla que va a mostrar el ajuste del modelo realizado. Nuestro tipo en particular incluye valor ROC, sensibilidad, especificidad y las desviaciones típicas de dichos valores.
- **classProbs=T** Con valor verdadero indica que se calcule la probabilidad de que pertenezca a una clase o a la otra.
- **savePredictions=T** Con valor verdadero indica que se guarden las predicciones de etiquetas durante la validación.

```
trControl = trainControl(method="cv", number = 5,
                         summaryFunction = twoClassSummary,
                         classProbs = T, savePredictions = T)
```

3 SUPPORT VECTOR MACHINES

El primer modelo que vamos a evaluar es **Support Vector Machines**. Esta técnica dispone de dos parámetros: *sigma*, el parámetro libre que tenemos que evaluar para encontrar la mejor solución con una precisión de dos cifras, y C el valor asociado a la regularización. Un valor alto de C implica dejar menos margen pero ajustar lo mejor posbile mientras que un valor pequeño implica dejar mucho margen aunque algunas muestras queden mal clasificadas.

Este modelo busca el hiperplano separador con un margen que puede ser óptimo o no (dependiendo del parámetro C) haciendo uso de un problema de programación cuadrática. Además, este problema de programación cuadrática, puede recibir los datos de entrada con una tranformación no-lineal gracias a los núcleos. En este caso, hemos utilizado un núcleo RBF-Gaussiano, es decir, hemos aplicado la transformación $K(x, x') = \frac{1}{2\sigma^2}||x - x'||^2$.

Puesto que el método es considerablemente lento, sólo hemos seleccionado ciertas combinaciones de sigmas (0.01,0.05, 0.1, 0.5, 1, 5, 10, 50, 500) y C (0.6,0.8,1) dándonos lugar a 45 combinaciones diferentes. El código en R para realizar dicha prueba es el siguiente:

```
grid = expand.grid(sigma = seq(0.01, 1, 0.05), C = c(0.6, 0.8, 1))
# Inicializamos la semilla de pseudoaleatorios
set.seed(7)

model_svm = train(Occupancy ~ ., training$df, method="svmRadial", trControl = trControl,
              tuneGrid = grid, metric="ROC")
```

Tras ejecutar el algoritmo obtenemos los siguientes resultados en validación. No obstante, hemos de tener en cuenta que teníamos un gran desbalance en las clases que estamos estudiando y, aunque obtengamos muy buenos resultados en training, existe la posibilidad de que se produzca algún ligero sobreajuste en la clase minoritaria a pesar del uso de la técnica *SMOTE* si se nos proporciona una nueva muestra.

```
pintar_roc = function(modelo, indices, caption, confusion = T) {
  # Mostramos la matriz de confusión
  if (confusion) {
    print(confusionMatrix(modelo))
  }

  # Pintamos la curva ROC
  pred = prediction(as.numeric(modelo$pred$pred[indices]),
```

```
                    as.numeric(modelo$pred$obs[indices]))
  perf = performance(pred, "tpr", "fpr")

  plot(perf, colorize=T, lwd=2, colorkey ="none",
     main=paste('Curva ROC para CV 5-fold de',
                caption))
}
# Mostramos la tabla con los resultados para cada combinación de parámetros
kable(x = model_svm$results[,1:5],
      caption = "Resultados de los ajustes por validación cruzada de SVM",
      col.names = c("sigma", "C", "ROC", "Sensitividad","Especificidad"))
```

Table 1: Resultados de los ajustes por validación cruzada de SVM

sigma	C	ROC	Sensitividad	Especificidad
0.01	0.6	0.9934071	0.9992291	0.9851069
0.01	0.8	0.9934806	0.9992291	0.9851069
0.01	1.0	0.9935141	0.9992291	0.9851069
0.06	0.6	0.9945986	0.9992289	0.9858298
0.06	0.8	0.9953438	0.9992289	0.9858298
0.06	1.0	0.9956223	0.9992289	0.9858298
0.11	0.6	0.9959907	0.9990361	0.9858298
0.11	0.8	0.9966335	0.9992289	0.9858298
0.11	1.0	0.9969752	0.9992289	0.9858298
0.16	0.6	0.9972113	0.9988434	0.9858298
0.16	0.8	0.9979106	0.9990361	0.9858298
0.16	1.0	0.9981940	0.9990361	0.9858298
0.21	0.6	0.9981317	0.9988434	0.9858298
0.21	0.8	0.9983443	0.9990361	0.9858298
0.21	1.0	0.9984835	0.9988434	0.9858298
0.26	0.6	0.9983362	0.9988434	0.9858298
0.26	0.8	0.9985887	0.9988434	0.9858298
0.26	1.0	0.9986717	0.9988434	0.9858298
0.31	0.6	0.9985397	0.9988434	0.9858298
0.31	0.8	0.9986441	0.9988434	0.9858298
0.31	1.0	0.9987176	0.9988434	0.9858298
0.36	0.6	0.9985549	0.9988434	0.9858298
0.36	0.8	0.9986522	0.9988434	0.9858298
0.36	1.0	0.9986993	0.9988434	0.9859744
0.41	0.6	0.9984914	0.9990361	0.9859744
0.41	0.8	0.9986210	0.9990361	0.9859744
0.41	1.0	0.9987633	0.9990361	0.9859744
0.46	0.6	0.9984332	0.9990361	0.9859744
0.46	0.8	0.9986451	0.9988434	0.9859744
0.46	1.0	0.9987374	0.9990362	0.9859744
0.51	0.6	0.9985062	0.9990361	0.9859744
0.51	0.8	0.9986448	0.9990361	0.9859744
0.51	1.0	0.9987619	0.9992289	0.9859744
0.56	0.6	0.9985353	0.9992289	0.9859744
0.56	0.8	0.9987046	0.9992289	0.9859744
0.56	1.0	0.9988267	0.9990362	0.9859744
0.61	0.6	0.9986125	0.9990362	0.9859744
0.61	0.8	0.9988282	0.9990362	0.9859744

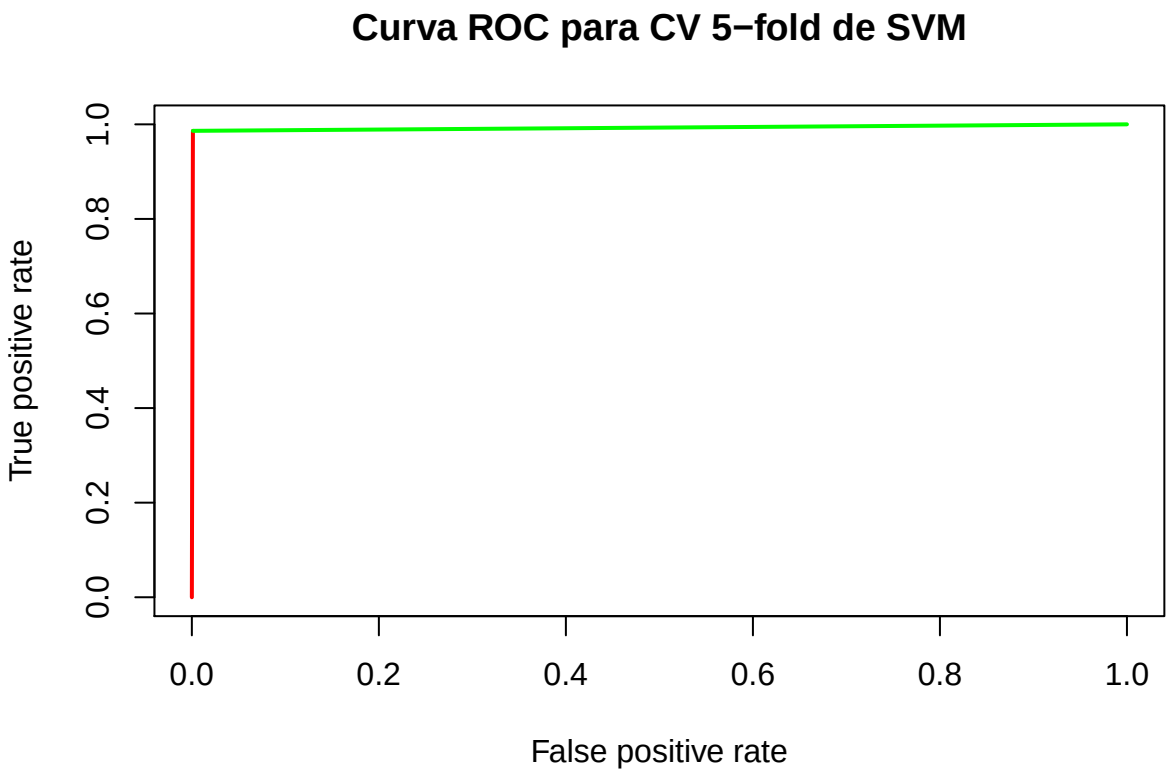
sigma	C	ROC	Sensitividad	Especificidad
0.61	1.0	0.9989038	0.9990362	0.9859744
0.66	0.6	0.9987618	0.9990362	0.9859744
0.66	0.8	0.9989439	0.9990362	0.9859744
0.66	1.0	0.9989938	0.9990362	0.9859744
0.71	0.6	0.9988524	0.9990362	0.9859744
0.71	0.8	0.9990015	0.9990362	0.9859744
0.71	1.0	0.9990821	0.9990362	0.9859744
0.76	0.6	0.9989125	0.9990362	0.9859744
0.76	0.8	0.9990592	0.9990362	0.9859744
0.76	1.0	0.9991499	0.9990362	0.9859744
0.81	0.6	0.9989786	0.9990362	0.9859744
0.81	0.8	0.9991128	0.9988436	0.9859744
0.81	1.0	0.9992017	0.9988436	0.9861189
0.86	0.6	0.9990436	0.9990362	0.9859744
0.86	0.8	0.9991584	0.9988436	0.9859744
0.86	1.0	0.9992210	0.9988436	0.9861189
0.91	0.6	0.9991116	0.9990362	0.9859744
0.91	0.8	0.9991801	0.9988436	0.9859744
0.91	1.0	0.9992257	0.9988436	0.9862634
0.96	0.6	0.9991459	0.9990362	0.9859744
0.96	0.8	0.9992104	0.9988436	0.9859744
0.96	1.0	0.9992190	0.9988436	0.9865526

```
cat("El mejor modelo usa sigma:", model_svm$bestTune$sigma, "y C:",
    model_svm$bestTune$C, "\n")
```

```
## El mejor modelo usa sigma: 0.91 y C: 1
```

```
# Cogemos los valores del mejor ajuste
selectedIndex1 = model_svm$pred$C == model_svm$bestTune$C &
                 model_svm$pred$sigma == model_svm$bestTune$sigma

pintar_roc(model_svm,selectedIndex1, "SVM")
```

```
## Cross-Validated (5 fold) Confusion Matrix
##
## (entries are percentual average cell counts across resamples)
##
##           Reference
## Prediction   SI   NO
##         SI 42.8  0.8
##         NO  0.0 56.4
##
##  Accuracy (average) : 0.9917
```



Podemos observar que el ajuste ofrecido por SVM es casi perfecto. No obstante, el problema de asignación cuadrática al que se enfrenta es considerablemente complejo, lo que hace el entrenamiento algo más complejo y largo debido a que el tamaño de la muestra de entrenamiento es muy grande. El mejor modelo que hemos obtenido tiene un sigma muy cercano a 1, es decir estamos realizando una transformación exponencial de aproximadamente la mitad de la distancia euclídea entre dos puntos del espacio de nuestro problema. Además, el valor 0.8 de regularización, no es excesivamente bajo ni tampoco excesivamente alto por lo que ciertamente limita el margen y nos permite tener un pequeño ruido en el entrenamiento a cambio de no producir un sobreajuste.

4 ADABOOST

El siguiente modelo que vamos a evaluar es **AdaBoost**. Este método se basa en el uso de múltiples clasificadores simples, en este caso, árboles de decisión débiles (*decision stumps*), los cuales son árboles de decisión de 1 único nivel de profundidad y que clasifican los datos en dos conjuntos diferentes. El método que hemos escogido en particular de **AdaBoost** es **Real AdaBoost**.

Real Adaboost inicializa un vector para la distribución a *1/tamaño de la muestra* y con el mismo tamaño que la muestra y en cada iteración actualiza dicha distribución, basándose en el paso anterior tomando una transformación exponencial de α, que da más importancia a las muestras mal clasificadas y menos importancia a las bien clasificadas y donde el valor α es calculado en cada iteración como $\alpha(x) = \frac{1}{2} ln\left(\frac{x}{1-x}\right)$, y encuentra un clasificador débil con un error lo suficientemente pequeño.

El clasificador final obtenido es el proporcionado en la última iteración. Para realizar esto hacemos uso del paquete *fastAdaboost*. Para este debemos especificar e número de iteraciones a usar y nosotros las vamos a desde 1 hasta 151 avanzando de 10 en 10.

```r
# Inicializamos la semilla de pseudoaleatorios
set.seed(7)

grid = expand.grid(method="Real adaboost", nIter = seq(1,151,10))
model_adaboost = train(x = training$x, y = training$y, method="adaboost",
                       trControl = trControl, metric ="ROC",
                       tuneGrid = grid, maxdepth = 1)
```

Veamos los resultados producidos por el ajuste:

```r
# Mostramos la tabla con los resultados para cada combinación de parámetros
kable(x = model_adaboost$results[,2:5],
      caption = "Resultados de los ajustes por validación cruzada de AdaBoost",
      col.names = c("Nº Iteraciones", "ROC",
                    "Sensitividad","Especificidad"))
```

Table 2: Resultados de los ajustes por validación cruzada de AdaBoost

Nº Iteraciones	ROC	Sensitividad	Especificidad
1	0.6291978	0.9926736	0.9963854
11	0.9872369	0.9965297	0.9976866
21	0.9545729	0.9971083	0.9972529
31	0.9516792	0.9974939	0.9973975
41	0.9539255	0.9973012	0.9975421
51	0.9549528	0.9971085	0.9975421
61	0.9655894	0.9971083	0.9976867
71	0.9543691	0.9969158	0.9976867
81	0.9455558	0.9969157	0.9975421
91	0.9540643	0.9969157	0.9976867
101	0.9535632	0.9969157	0.9979759
111	0.9543052	0.9973010	0.9979759
121	0.9560017	0.9969157	0.9979759
131	0.9569680	0.9971083	0.9979759
141	0.9572742	0.9969155	0.9979759
151	0.9559897	0.9971082	0.9979759

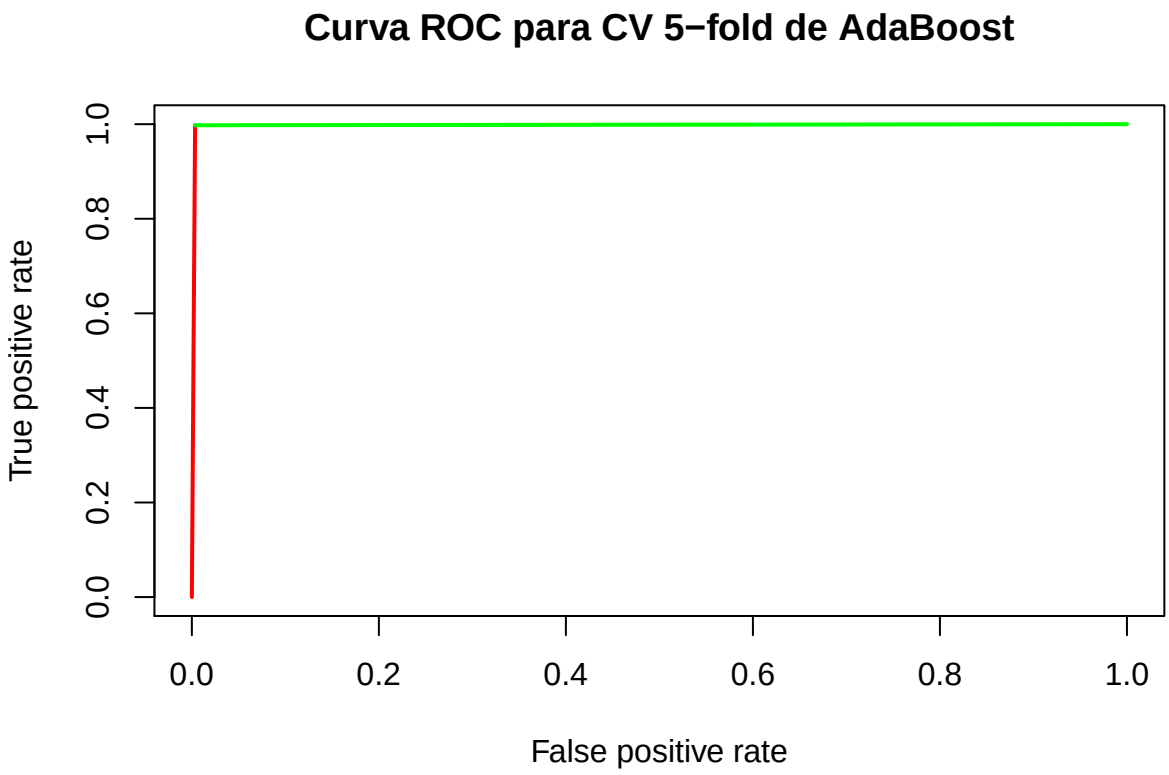
```r
cat("El mejor modelo usa nIter:", model_adaboost$bestTune$nIter, "\n")
```

```
## El mejor modelo usa nIter: 11
```

```r
# Cogemos los valores del mejor ajuste
selectedIndex2 = model_adaboost$pred$method == model_adaboost$bestTune$method &
                 model_adaboost$pred$nIter == model_adaboost$bestTune$nIter

pintar_roc(model_adaboost,selectedIndex2, "AdaBoost")
```

```
## Cross-Validated (5 fold) Confusion Matrix
##
## (entries are percentual average cell counts across resamples)
##
##           Reference
## Prediction   SI   NO
##         SI 42.7  0.1
```

```
##          NO  0.1 57.0
##
##  Accuracy (average) : 0.9972
```



Podemos observar que tras probar con distintos números de iteraciones (nIter) hemos obtenido muy buenos ajustes, y observamos que AdaBoost es un algoritmo fácil de ajustar ya que el único parámetro que obtenemos es el mencionado número de iteraciones. Para observar el funcionamiento hemos comprobado con valores desde 1 iteración hasta 151 iteraciones (de 10 en 10), y el mejor resultado obtenido es tan solo con 11 iteraciones, por lo que deducimos que al haber más iteraciones se produce un ligero sobreajuste, ya que posee mayor error, pero no es un error muy significativo. Esto podría deberse a que al añadir un nuevo árbol, la muestra actual se clasifique bien con la distribución actual pero reduzca el margen necesario para que en otra muestra de validación posterior no salga un error.

Por otro lado, al tratarse del mejor resultado con 11 iteraciones, deducimos que se trata de un muy buen resultado para tratarse de tan pocas iteraciones, por lo que concluimos con que AdaBoost es un muy buen algoritmo, pese a su costoso tiempo de ejecución.

5 RANDOM FOREST

El siguiente modelo que vamos a probar es **Random Forest**. Este método construye un número de árboles (especificado por el parámetro *ntree*) sobre muestras obtenidas por *bootstrapping* pero evitando el alto grado de correlación entre los árboles haciendo una selección aleatoria de las variables que actuarán como predictores (especificado en el parámetro *mtry*). Es habitual que el mejor valor ronde la raíz cuadrado del número de variables de las que disponemos. Además, conforme mayor sea el valor de mtry, puesto que habrá menos combinaciones aleatorias posibles, mayor será el grado de correlación entre los distintos árboles. En el caso en el que *mtry* sea exactamente el número de predictores totales estamos en realidad realizando *bagging*.

```r
# Inicializamos la semilla de pseudoaleatorios
set.seed(7)
grid = expand.grid(mtry = 1:5)
model_randomforest = train(x = training$x, y = training$y, method="rf",
                           trControl = trControl, metric ="ROC",
                           tuneGrid = grid,
                           ntree = 82)

  mejor_error = length(test$y)
  num_arboles = 1
  for(ntree in seq(1,200,1)){
    set.seed(7)
    modelo_actual = randomForest(Occupancy ~ .,
                                 data = test$df, replace = T, ntree = ntree,
                                 mtry = 1)

    # Predicciones
    etiquetas_nuevas = predict(modelo_actual, test$df[-6])

    # Fallo en %
    errores_prediccion = sum(etiquetas_nuevas!=test$y)/length(test$y)*100

    # cat("Errores prediccion con" , ntree, "árboles:", errores_prediccion, "\n")

    if(errores_prediccion < mejor_error){
      mejor_error = errores_prediccion
      num_arboles = ntree
    }

  }
    cat("Numero de árboles óptimo",num_arboles, "\n")
```

```
## Numero de árboles óptimo 51
```

```r
    cat("Error minimo conseguido",mejor_error, "\n")
```

```
## Error minimo conseguido 0.04101723
```

Como podemos observar, una vez había suficientes árboles, realmente no existen grandes diferencias en los resultados salvo el tiempo de ejecución así que en el training realizado previamente hemos tomado 82 árboles para realizar el ajuste.

Evaluamos los resultados:

```r
# Mostramos la tabla con los resultados para cada combinación de parámetros
kable(x = model_randomforest$results[,1:4],
      caption = "Resultados de los ajustes por validación cruzada de Random Forest",
      col.names = c("Nº Predictores aleatorios", "ROC", "Sensitividad","Especificidad"))
```

Table 3: Resultados de los ajustes por validación cruzada de Random Forest

Nº Predictores aleatorios	ROC	Sensitividad	Especificidad
1	0.9998789	0.9959517	0.9976867
2	0.9998776	0.9961446	0.9973975
3	0.9997851	0.9953731	0.9975421

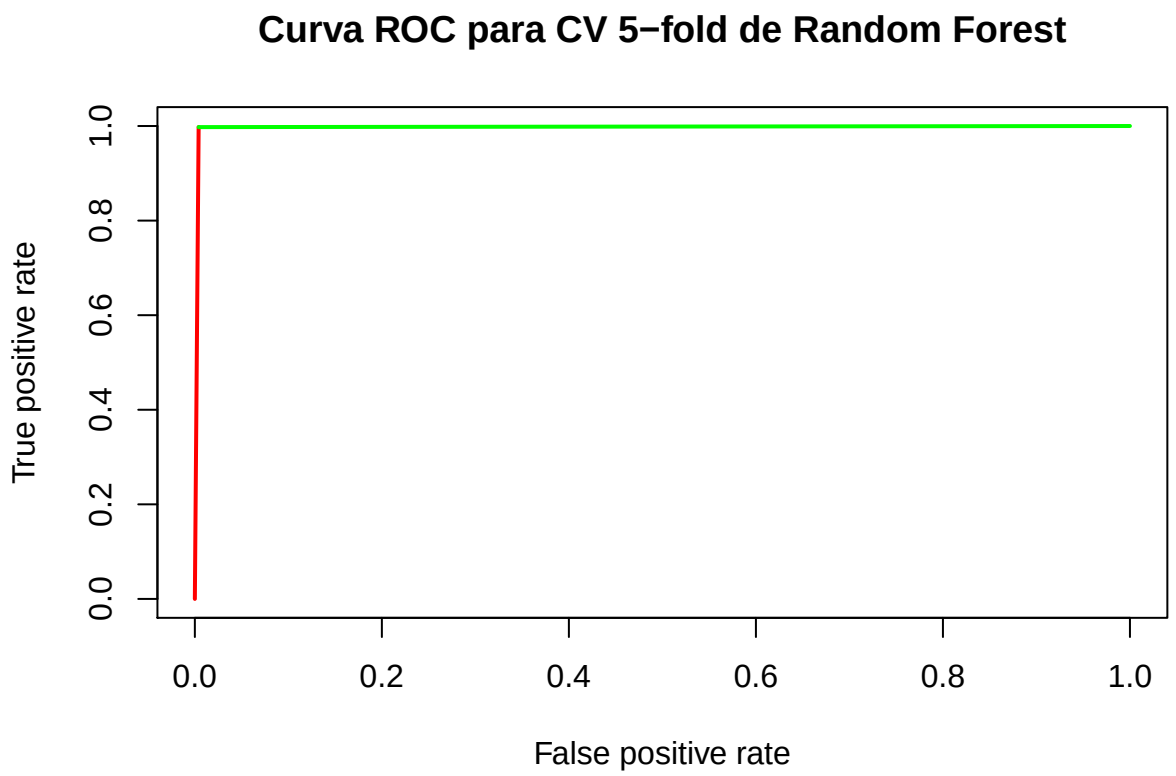
Nº Predictores aleatorios	ROC	Sensitividad	Especificidad
4	0.9997762	0.9951804	0.9976867
5	0.9993097	0.9949876	0.9978313

```
cat("El mejor modelo usa número de predictores:", model_randomforest$bestTune$mtry,
    "\n")
```

```
## El mejor modelo usa número de predictores: 1
```

```
# Cogemos los valores del mejor ajuste
selectedIndex3 = model_randomforest$pred$mtry ==
                 model_randomforest$bestTune$mtry

pintar_roc(model_randomforest,selectedIndex3, "Random Forest")
```

```
## Cross-Validated (5 fold) Confusion Matrix
##
## (entries are percentual average cell counts across resamples)
##
##           Reference
## Prediction   SI   NO
##         SI 42.7  0.1
##         NO  0.2 57.0
##
##  Accuracy (average) : 0.9969
```



En este caso el mejor valor que hemos obtenido es el que utiliza 2 predictores aleatorio, que, coincide con el valor de la raíz del número de atributos. Cabía esperar que conforme los número fueran más altos saldría un modelo más malo porque se parecería cada vez más a ejecutar bagging aumentando así el grado de correlación entre los árboles creados hasta llegar a bagging con m = 5.

Podemos observar que nos da una solución muy aproximada a los óptimos, con lo que ajusta muy bien la muestra de partida. Además, es el método más rápido de los evaluados hasta el momento y la complejidad relativa del mismo es bastante baja.

6 DECISIÓN DEL MODELO

Prácticamente, todos los modelos nos han dado resultados muy parecidos en la validación cruzada, la princial diferencia de ellos radica en la complejidad y el tiempo que ha tardado en producirse el entrenamiento. De entre ellos uno ha sido el que tanto más rápido ha realizado el entrenamiento y a la vez ha obtenido los mejores valores para la métrica ROC y la métrica de precisión (*Accuracy*) hemos escogido **Random Forest con 2 predictores y 81 árboles como modelo seleccionado**.

Por último, aunque ya hayamos estimado el error fuera de la muestra con la validación cruzada, vamos a evaluar con una muestra completamente nueva de test para ver si, a pesar de los problemas de desbalanceo de las clases, realmente hemos escogido un buen modelo.

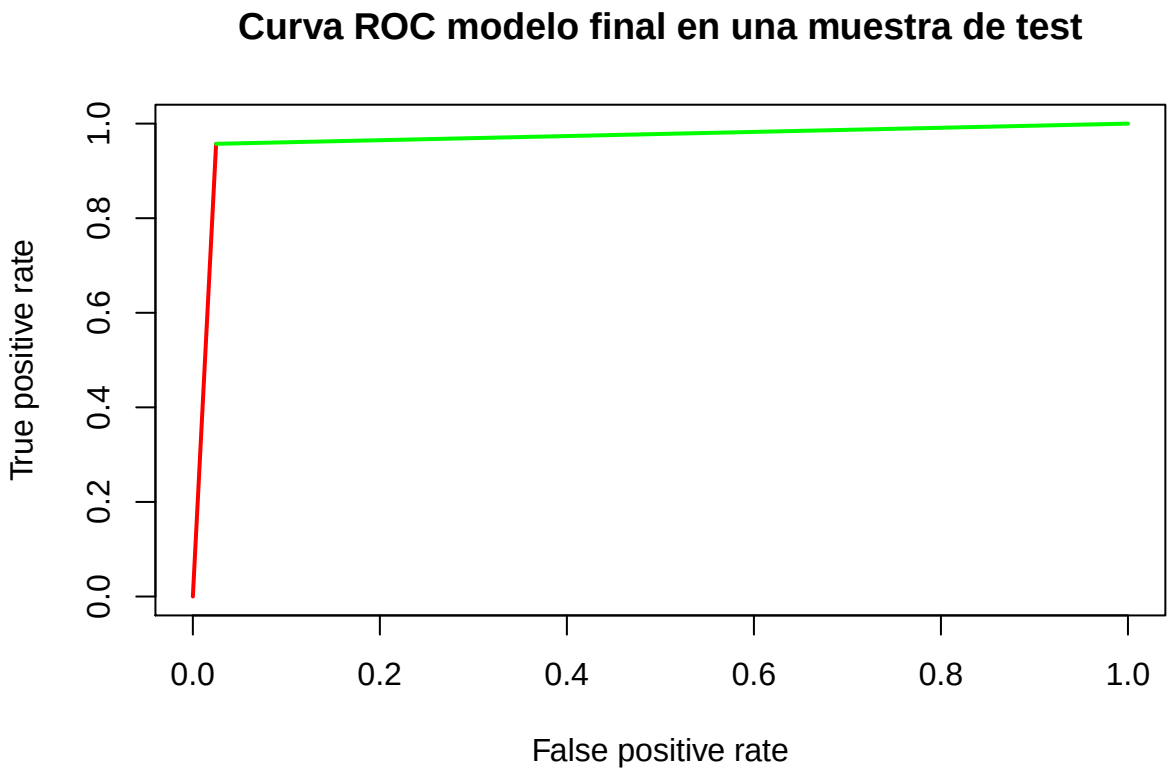
```
# Calculamos las etiquetas en test con el modelo escogido
etiquetas = predict(model_randomforest, test$x)

# Mostramos la matriz de confusión
confusionMatrix(etiquetas, test$y)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   SI   NO
##         SI 1998  329
##         NO   51 7374
##
##                Accuracy : 0.961
##                  95% CI : (0.957, 0.9648)
##     No Information Rate : 0.7899
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.8882
##  Mcnemar's Test P-Value : < 2.2e-16
##
##             Sensitivity : 0.9751
##             Specificity : 0.9573
##          Pos Pred Value : 0.8586
##          Neg Pred Value : 0.9931
##              Prevalence : 0.2101
##          Detection Rate : 0.2049
##    Detection Prevalence : 0.2386
##       Balanced Accuracy : 0.9662
##
##        'Positive' Class : SI
##
```

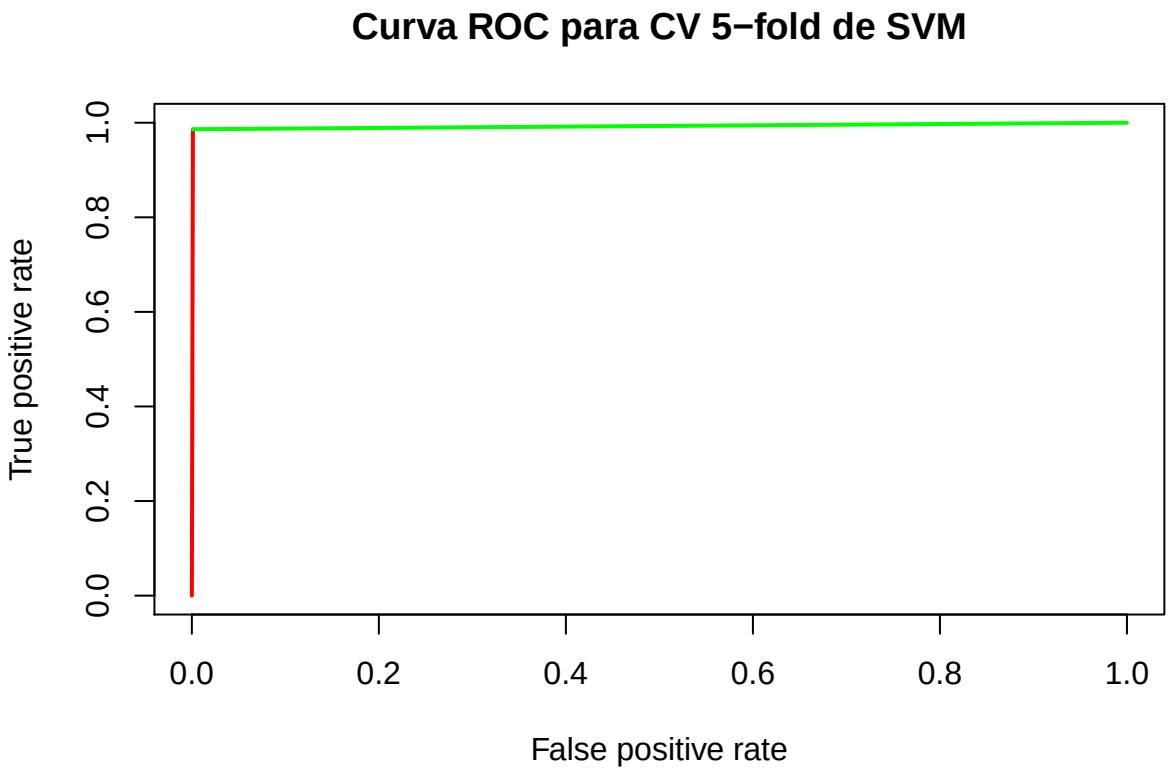
```r
# Pintamos la curva ROC
  pred = prediction(as.numeric(etiquetas),
                    as.numeric(test$y))

  perf = performance(pred, "auc")

  cat("El area debajo de la curva ROC es de:", perf@y.values[[1]])
```

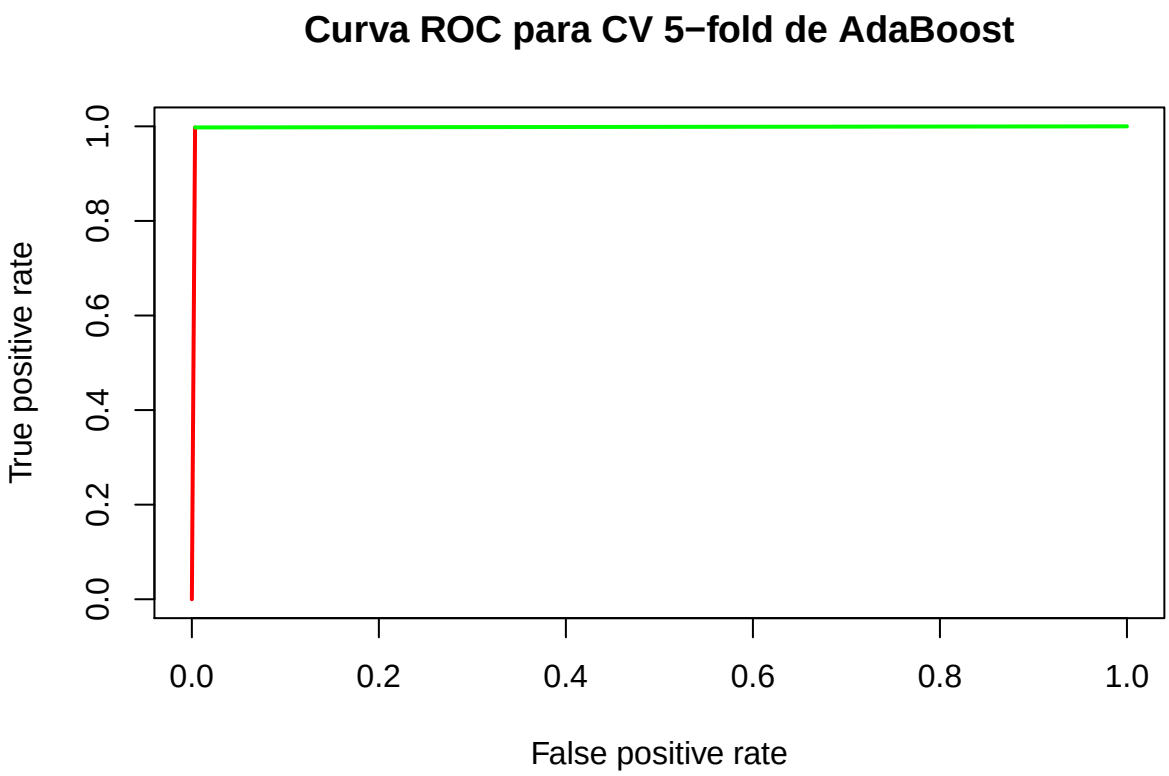
```
## El area debajo de la curva ROC es de: 0.9661996
```

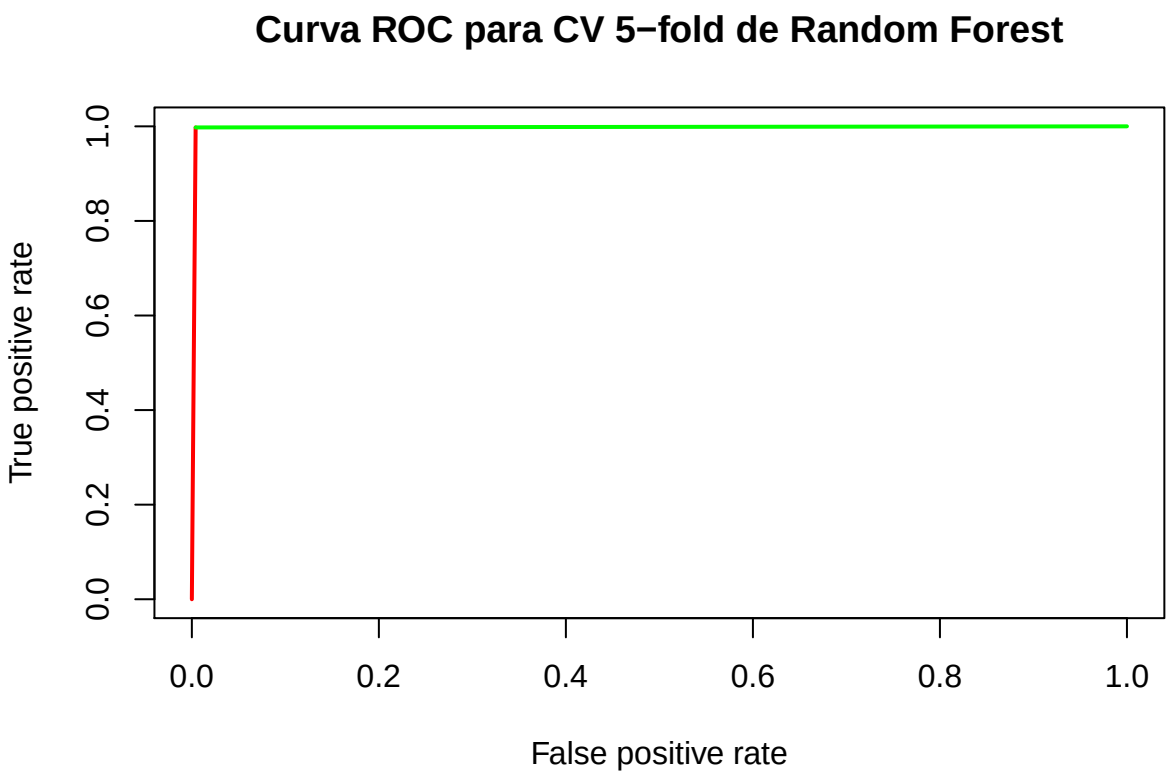
```r
  perf = performance(pred, "tpr", "fpr")

  plot(perf, colorize=T, lwd=2, colorkey ="none",
     main="Curva ROC modelo final en una muestra de test")
```



7 CONCLUSIÓN




Curva ROC para CV 5−fold de AdaBoost
True positive rate
0.0
0.2
0.4
0.6
0.8
1.0
False positive rate
0.0
0.2
0.4
0.6
0.8
1.0



Puesto que la curva ROC es muy similar (todas con un área muy cercano a 1 por debajo de la curva, por tanto, muy bueno en ajuste en test) todos los métodos probados dan muy buenos resultados. No obstante, en todos los casos hemos tomado como criterio principal para la selección del método a usar el tiempo de ejecución y la complejidad del mismo, llevándonos a usar **Random Forest**, que es a su vez, el que daba resultados ligeramente mejores en validación.

Todos los modelos no lineales que hemos probado han demostrado ser lo suficientemente potentes para ajustar con gran precisión la muestra de entrenamiento. Los modelos con más parámetros (SVM) han sido los que han dado los resultados ligeramente más bajos en la validación aunque debido al elevado tiempo que requiere su ejecución sólo hemos seleccionado algunos valores de los posibles y quizás existe algún valor que no hemos probado y da resultados ligeramente mejores que los que hemos obtenido. Aun así, la poca complejidad de los otros dos métodos y el menor coste computacional hizo que descartáramos las redes neuronales (que no hemos llegado a probar en este trabajo) y SVM como nuestra selección. La principal diferencia entre Random Forest y AdaBoost reside en los tiempos de ejecución. Aunque los diferencias no han sido radicales, Random Forest ha sido ligeramente más rápido y, por tanto, nuestra selección.

Como podemos observar, a pesar de los problemas que podría haber supuesto el desbalanceo inicial de las clases a predecir, el cual hemos solucionado añadiendo muestras basándonos en las propiedades de las que ya teníamos (con *SMOTE*), obtenemos una precisión del 96,79 % con Random Forest en test (que es otra vez una muestra desbalanceada) y un área debajo de la curva ROC de 0,9708926 y una precisión del 99,65 % en validación cruzada y un área debajo de la curva ROC de 0,9996171 durante el training, dando lugar a un muy buen ajuste con un modelo no lineal el cual es bastante simple en complejidad.