
UNIVERSIDAD DE GRANADA

Aprendizaje Automático

---

# *Proyecto final*

---

*Laura Calle Caraballo*
*Javier León Palomares*

1 de junio de 2017

# Índice

# 1. Introducción

Para la realización de este trabajo se ha empleado la base de datos *Tennis Major Tournament Match Statistics*, que recoge información acerca de los partidos disputados en los *Grand Slam* de 2013 (excepto por el Abierto de Australia, que por alguna razón es el de 2014). Los campos aportados por cada partido son los siguientes:

- Nombres de los participantes.
- Resultado (desde el punto de vista del primer participante).
- Ronda.
- Porcentajes relativos al primer servicio de cada jugador: jugados y ganados.
- Porcentajes relativos al segundo servicio de cada jugador: jugados y ganados.
- Número de *aces* por cada jugador.
- Número de dobles faltas cometidas por cada jugador.
- Número de puntos ganadores por cada jugador.
- Número de errores no forzados cometidos por cada jugador.
- Número de puntos de *break* creados y ganados por cada jugador.
- Número de puntos en la red intentados y ganados por cada jugador.
- Puntos totales ganados por cada jugador.
- Juegos por set de cada jugador.
- Número total de sets ganados por cada jugador.

Esta base de datos contiene valores perdidos. No obstante, todos ellos representan cantidades que se pueden sustituir por cero de forma segura ya que denotan, por ejemplo, ausencia de errores no forzados, ausencia de *aces* o sets no jugados (por abandono o por no ser necesarios).

Debido a que este conjunto de datos no tiene una variable de respuesta definida explícitamente, vamos a elegir una: el resultado del partido. Para que la predicción no sea trivial, es necesario eliminar algunas características que permiten determinar unívocamente el resultado, como el número de sets ganados por cada jugador y la puntuación dentro de los mismos. Además, por la naturaleza del dominio del problema, aprovecharemos que se nos proporciona implícitamente el tipo de pista en el que se han jugado los partidos para considerarlo como una característica más; ya que es un factor con tres niveles, se añadirán dos variables *dummy* para contemplarlo (la tercera clase es una combinación lineal de las otras dos). Una vez realizado todo esto, pasaremos de 40 variables a 28 y comprobaremos así si es factible utilizar este nuevo conjunto de predictores con el mismo propósito.

Un aspecto a tener en cuenta es el sesgo del propio conjunto de datos, provocado por la forma de construir las observaciones. Al ser la asignación de *Jugador 1* y *Jugador 2* arbitraria, de forma implícita se está favoreciendo que el mismo estadístico para un jugador tenga más importancia que para otro. Nosotros, por conocimiento general de este problema, sabemos que no importa en qué orden los nombremos; sin embargo, los modelos no tienen esta información y pueden llegar a conclusiones sesgadas basadas en este orden. Por ejemplo, para un mismo partido, la observación en la que el primer jugador comete 10 dobles faltas y el segundo comete 5 es equivalente a la observación en la que el primer jugador comete 5 dobles faltas y el segundo 10, ya que los jugadores son intercambiables. Un modelo no tiene esta información intuitiva y, por tanto, podría dar importancias distintas a variables en teoría simétricas.

En cuanto a la división en conjuntos de entrenamiento y test, no se nos proporciona un criterio de antemano, por lo que en principio la realizaremos nosotros tomando observaciones aleatorias según la proporción 70 %-30 %.

## 2. Preprocesamiento de datos.

Para realizar el preprocesamiento de datos hemos empleado la función `preProcess` del paquete `caret`; esta función permite aplicar distintos métodos de transformación para adecuar las características iniciales a los modelos que vamos a emplear.

A continuación vamos a explicar los métodos utilizados:

- Transformación de *Yeo-Johnson*: es muy similar a la transformación de *Box-Cox* pero, a diferencia de ésta, permite la existencia de valores negativos ó 0. En nuestro caso, no existen valores negativos pero sí es bastante frecuente encontrar valores iguales a 0.
- Centrado: se realiza la media de los valores de cada característica y se le resta a cada valor particular, siendo la nueva media igual a 0. Esto reduce las distancias entre las distintas características.
- Escalado: este método divide los valores por su desviación típica. El objetivo de esta transformación es tener características con rangos uniformes para evitar problemas con modelos que consideran distancias entre valores (lo cual favorecería las características con valores más separados entre sí); además, la independencia respecto a la escala en la que se realizaron las medidas también es importante.
- Análisis de componentes principales (*PCA*): se buscan nuevos ejes de coordenadas de forma que la varianza de algunas características sea lo suficientemente pequeña como para descartarlas. Como resultado se consigue una reducción de dimensionalidad donde las nuevas características, que no guardan relación de significado con las originales, representarán la parte más significativa de la varianza de los datos.

Otro método que se podría haber utilizado para reducir la dimensionalidad de los datos es *Near Zero Variance*, que elimina predictores con varianza cercana a 0, ya que esto suele indicar que su valor es casi constante a lo largo de la muestra y apenas aportan información. Sin embargo, hemos comprobado empíricamente que no afecta a nuestro conjunto de datos en particular.

Una vez explicadas las transformaciones, veamos cómo se traduce en la práctica a código:

```
# Lectura de datos y obtención del conjunto de entrenamiento
datos = leer_datos_partidos()
etiquetas = datos$etiquetas
indices_train = datos$indices_train
datos = datos$datos

preprocesar_datos = function(datos,indices_train,metodos,umbral_varianza=0.9){
    preprocess_obj = preProcess(datos[indices_train,],method=metodos,umbral_varianza)
    nuevosDatos = predict(preprocess_obj,datos)
}

datos[is.na(datos)] = 0 # Sustituimos los valores perdidos por 0
datos = subset(datos,select=-c(FNL1,FNL2)) # Número de sets ganados por cada uno
datos = subset(datos,select=-c(ST1.1,ST2.1,ST3.1,ST4.1,ST5.1,ST1.2,ST2.2,ST3.2,ST4.2,ST5.2))
datos_procesados = preprocesar_datos(datos,indices_train,
                                     c("YeoJohnson","center","scale","pca"),0.85)
datos_procesados_sin_pca = preprocesar_datos(datos,indices_train,
                                             c("YeoJohnson","center","scale"))
```

El análisis de componentes principales reduce a 9 los predictores a utilizar. Ya que existe la posibilidad de que este análisis elimine características relevantes, vamos a realizar experimentos separados preprocesando los datos con y sin esta técnica.

# 3. Modelos a estudiar

En principio, podemos comenzar comprobando la calidad de un modelo lineal; tras ello, fuese o no necesario (debido a la naturaleza de este proyecto), probaremos modelos no lineales como *Random Forest*, *Boosting*, *Support Vector Machines* o redes neuronales.

## 3.1. Modelos lineales

Ya que el objetivo de este proyecto es analizar la eficacia de modelos no lineales, elegiremos un único modelo lineal de forma no tan exhaustiva como en el trabajo 3.

En primer lugar, vamos a formarnos una idea de la utilidad en este contexto de ambos conjuntos de datos preprocesados guiándonos por la tasa de error que produce una regresión logística con todas sus características:

```
reg_log = evalua_glm(etiquetas~.,datos_procesados,indices_train)
reg_log_sin_pca = evalua_glm(etiquetas~.,datos_procesados_sin_pca,indices_train)
```

El primer ajuste da un error de 8.83 %, frente al segundo, que da un 5.3 %. Según este criterio, elegiremos el segundo modelo; no obstante, ya que utiliza todas las características a su disposición, vamos a tratar de encontrar un subconjunto de ellas con la función `regsubsets` que mantenga la calidad a la vez que ofrece una reducción de dimensionalidad.

Dicha función es empleada de la siguiente forma para obtener los subconjuntos de variables con los que vamos a calcular las distintas regresiones:

```
subconjuntos_formulas = function(datos,max_tam,metodo="exhaustive"){
    # Obtenemos los subconjuntos de variables
    subsets = regsubsets(etiquetas~.,data=datos,method=metodo,nvmax=max_tam)
    # Obtenemos la matriz de características seleccionadas por grupos de tamaño [1,nvmax]
    matriz_subsets = summary(subsets)$which[,-1]
    # Guardamos, para cada fila, las columnas cuyas variables han sido seleccionadas.
    seleccionados = apply(matriz_subsets,1,which)
    # Obtenemos los nombres de esas columnas (para utilizarlos en la regresión)
    seleccionados = lapply(seleccionados,names)
    # Construimos la suma de las variables que usaremos en la regresión lineal
    seleccionados = mapply(paste,seleccionados,MoreArgs=list(collapse="+"))
    # Construimos strings equivalentes a las fórmulas que usaremos en la regresión lineal
    formulas = mapply(paste,rep("etiquetas~",max_tam),seleccionados,USE.NAMES = FALSE)
    # Construimos objetos fórmula
    formulas = apply(matrix(formulas,nrow=length(formulas)), 1, as.formula)
    list(formulas=formulas,cp=summary(subsets)$cp,bic=summary(subsets)$bic)
}
```

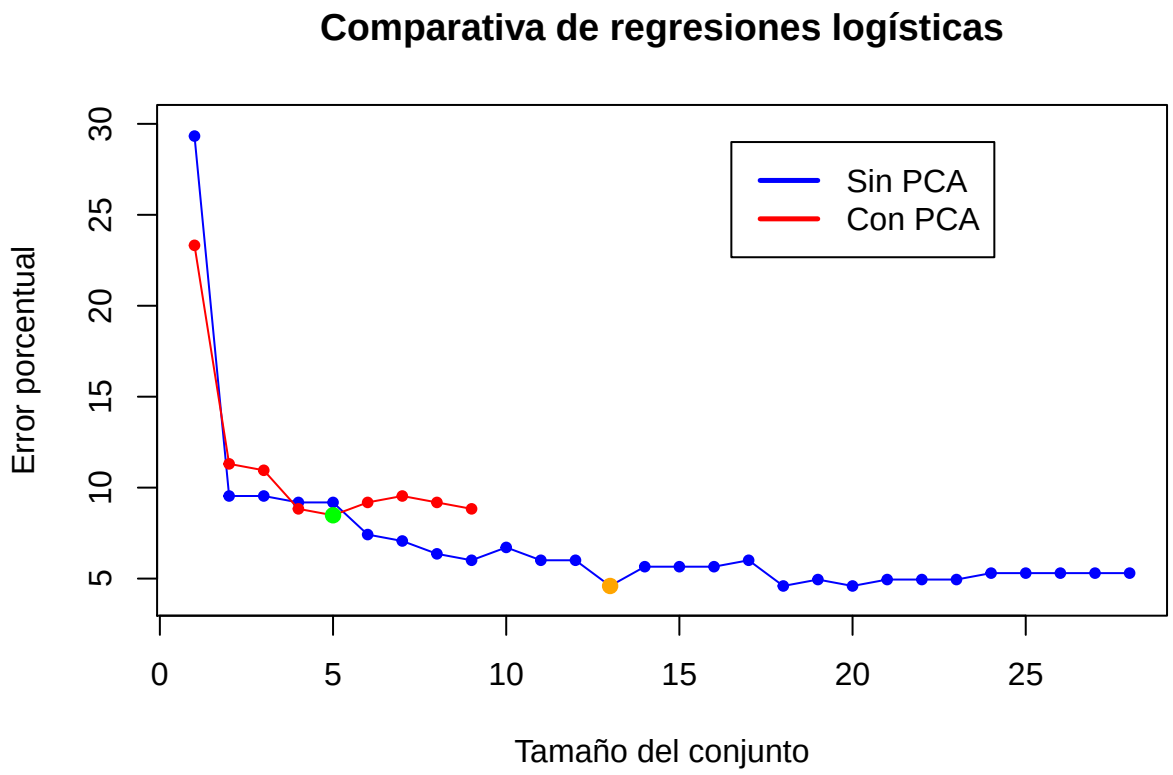
Un ejemplo de uso es:

```
# Seleccionamos subconjuntos de características para los datos con PCA
max_caracteristicas = ncol(datos_procesados)-1
seleccion_caracteristicas = subconjuntos_formulas(datos_procesados[indices_train,],
                                                  max_caracteristicas,metodo="exhaustive")
formulas = seleccion_caracteristicas$formulas
```

Ahora realizaremos ajustes para todos los subconjuntos de características obtenidos mediante el código anterior:

```
ajustes_glm = mapply(evalua_glm, formulas,
                     MoreArgs = list(datos = datos_procesados,
                                     subconjunto = indices_train))
ajustes_glm_sin_pca = mapply(evalua_glm, formulas_sin_pca,
                             MoreArgs = list(datos = datos_procesados_sin_pca,
                                             subconjunto = indices_train))
```

Para comparar más fácilmente los porcentajes de error obtenidos por los modelos, vamos a representarlos en la siguiente gráfica:




El punto señalado en verde nos indica el conjunto de predictores cuyo error ha sido menor (8.48 %) para los datos con análisis de componentes principales. Análogamente, el punto naranja señala el conjunto de características que ha dado menor error (4.59 %) para los datos sin PCA.

Ya que hemos conseguido no sólo reducir el número de predictores necesarios (de 28 a 13) sino también el error, vamos a elegir el modelo que proporciona un 4.59 % de error para futuras comparaciones con modelos no lineales.

Cabe mencionar que de ahora en adelante las transformaciones de preprocesamiento de datos serán sin análisis de componentes principales puesto que, como ya hemos visto, no sólo obtienen mejores resultados sino que facilitan la interpretación de los mismos.

## 3.2. Modelos no lineales

Procediendo de forma similar, analizaremos varios modelos no lineales utilizando los dos conjuntos de características obtenidos en la sección de preprocesamiento de datos. Antes de empezar, es preciso apuntar que los ajustes lineales anteriores han obtenido errores bastante bajos, por lo que la complejidad adicional introducida por las técnicas que veremos a continuación deberá tener asociado un error aún más bajo que la justifique.

Para automatizar la optimización de hiperparámetros mediante validación cruzada utilizaremos las funcionalidades proporcionadas por el paquete `caret`; en concreto, la función `train`. Esto se explorará con mayor profundidad en los apartados dedicados a cada tipo de modelo.

### 3.2.1. *Random Forest*

El primer modelo no lineal que vamos a estudiar será *Random Forest*. Esta técnica consiste en entrenar un cierto número de árboles usando subconjuntos aleatorios de características para después promediar sus resultados.

Partiendo del concepto de *bagging*, *Random Forest* muestrea con reemplazo el conjunto de $n$ datos original para obtener $p$ conjuntos de un cierto tamaño $n'$ con los que ajustar $p$ árboles. Lo que diferencia estas dos técnicas es la elección de subconjuntos aleatorios de variables utilizadas en el ajuste de cada árbol que realiza *Random Forest*.

Como hemos comentado, vamos a utilizar `train` para encontrar, en este caso, el número de árboles (`ntree`) más adecuado para una cantidad de variables por árbol fija.

Para empezar, vamos a especificar que queremos realizar validación cruzada. Esto se hace creando un objeto `trainControl` que posteriormente `train` tomará como argumento.

```
control = trainControl(method = "cv", number = 10) # 10-fold cv
```

El siguiente paso es crear un conjunto de posibles valores de `ntree`:

```
num_arboles = seq(50,500,5)
```

Utilizaremos la función que se muestra a continuación para obtener el error de validación cruzada para cada uno de esos valores:

```
evalua_random_forest_cv = function(datos,etiquetas,control,arboles=100){
    mtry = sqrt(ncol(datos))
    rf_train = train(datos,as.factor(etiquetas),
                     method="rf",ntree=arboles,
                     preProcess=c("YeoJohnson","center","scale"),
                     trControl=control,tuneGrid=expand.grid(.mtry=mtry))
    rf_error = (1-rf_train$results$Accuracy)*100
    list(arboles=arboles,error=rf_error,rf=rf_train)
}

ajustes_rf_cv = mapply(evalua_random_forest_cv,num_arboles,
                       MoreArgs = list(datos=datos[indices_train,],
                                       etiquetas=etiquetas[indices_train],
                                       control=control))
```

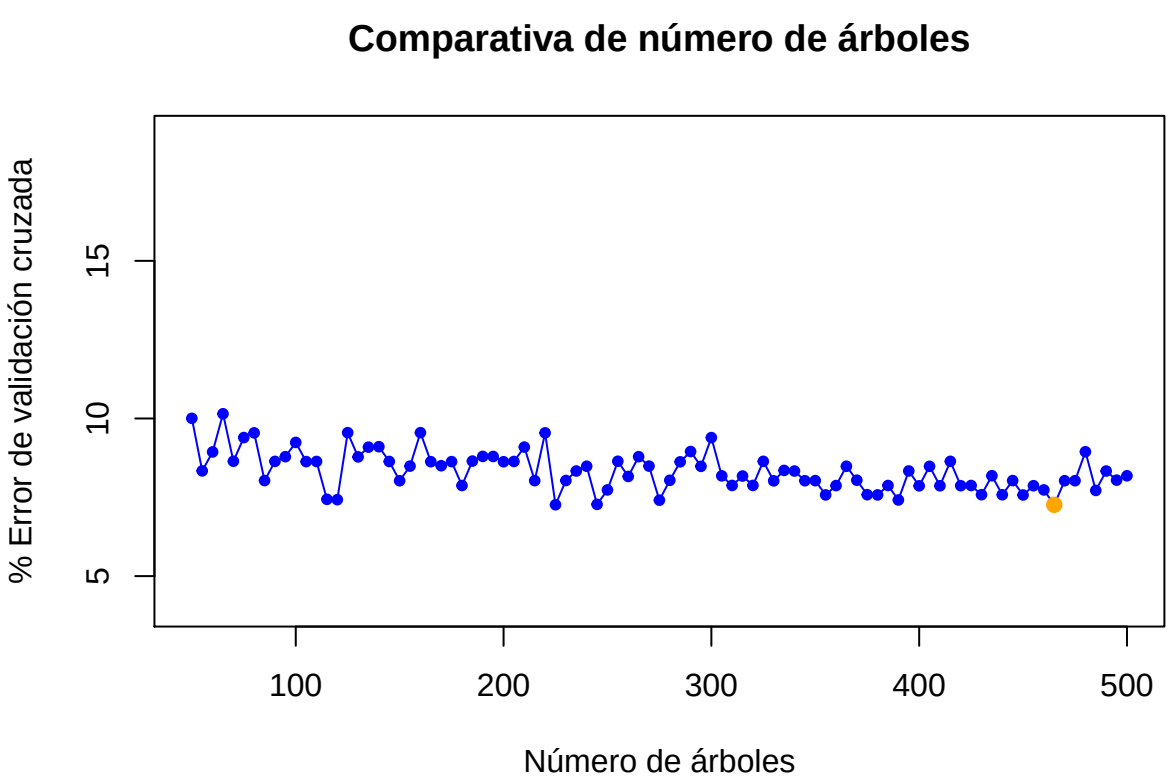
Siguiendo las directrices del proyecto, mantendremos constante el número de variables predictoras a $\sqrt{n}$, donde $n$ es el total de características disponibles.

Los parámetros principales que recibe `train` son, por este orden: los datos originales, las etiquetas, el método (en nuestro caso, *Random Forest*), el número de árboles específico y las transformaciones de preprocesamiento.

Además, hemos de pasarle el objeto `control` y el número de variables definidos previamente. Finalmente, debemos tener en cuenta otros parámetros que no aparecen en el código por tener valores correctos por defecto: `replace` (muestreo con o sin reemplazo) o `sampsize` ($n'$, tamaño de los conjuntos obtenidos).

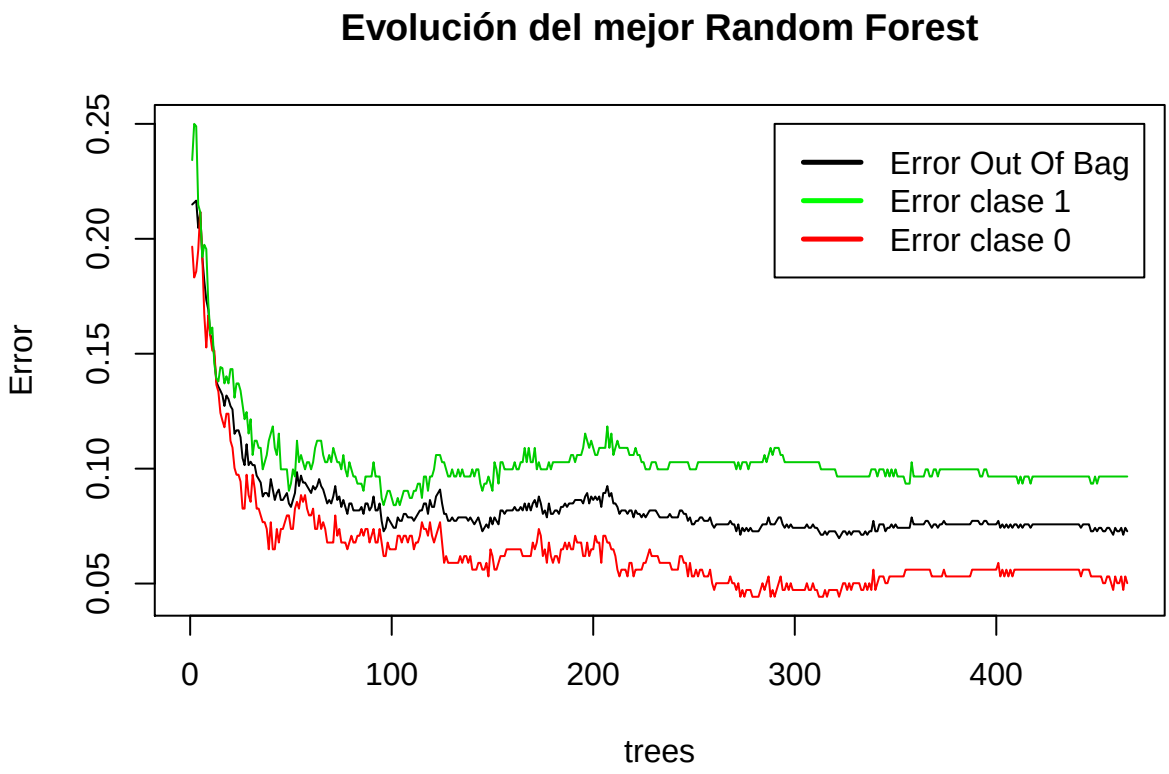
Ya que vamos a realizar validación cruzada, es importante no contaminar el conjunto de datos realizando el preprocesamiento antes de crear las particiones de validación. La razón de esto es que esta técnica toma información de todos los datos a su disposición para homogeneizar los valores de las características, por lo que si preprocesamos el conjunto completo las particiones de validación no serán totalmente independientes. Es por esto que `train` aplica las transformaciones en cada iteración de la validación cruzada.

Una vez obtenidos los $E_{CV}$, decidiremos qué número de árboles es el más adecuado entre los candidatos según este criterio. Esta decisión se ilustra mejor con la siguiente gráfica:




El punto señalado en naranja indica el número de árboles con el que se ha obtenido el menor $E_{CV}$: 465. Por ello, ajustaremos un modelo con 465 árboles utilizando toda la partición de entrenamiento para posteriormente evaluar su calidad sobre la partición de test.

Para finalizar, veamos cómo ha evolucionado el error de este modelo conforme aumenta su número de árboles:




Vemos que el error *Out Of Bag* disminuye progresivamente hasta estabilizarse. Esto es así porque un modelo con pocos árboles adolece de sobreajuste, problema que se soluciona en parte al promediar las predicciones de un número creciente de clasificadores.

### 3.2.2. *Boosting*

El siguiente tipo de modelo no lineal a estudiar es *Boosting*; en particular, la variante *AdaBoost* utilizando funciones *stump*.

El concepto subyacente a *Boosting* es combinar un conjunto predefinido de clasificadores débiles, entendiendo éstos como clasificadores cuya eficacia es marginalmente superior a uno aleatorio. Para constituir el modelo complejo final, se realiza iterativamente una agregación ponderada de dichos clasificadores en función de su $E_{in}$.

En nuestro caso, los clasificadores débiles que utilizaremos son denominados *stumps*, los cuales pueden ser vistos como árboles de decisión de un único nivel. La ponderación de cada *stump* $t$ viene dada por el valor de su error $\epsilon_t$, calculado en base a una distribución $D_t$. Esta distribución confiere una importancia a cada observación, de forma que aquellas que han sido mal clasificadas durante más iteraciones tienen preferencia en las siguientes.

Para la implementación vamos a utilizar de nuevo `train`; en este caso con el método `ada`. Este método tiene tres hiperparámetros: la profundidad de los árboles clasificadores simples (`maxdepth`); el número de iteraciones o, de forma equivalente, el número de clasificadores distintos (`iter`); el coeficiente de aprendizaje (`nu`).

Puesto que vamos a utilizar como clasificadores débiles *stumps*, la profundidad de los árboles estará fijada a 1.

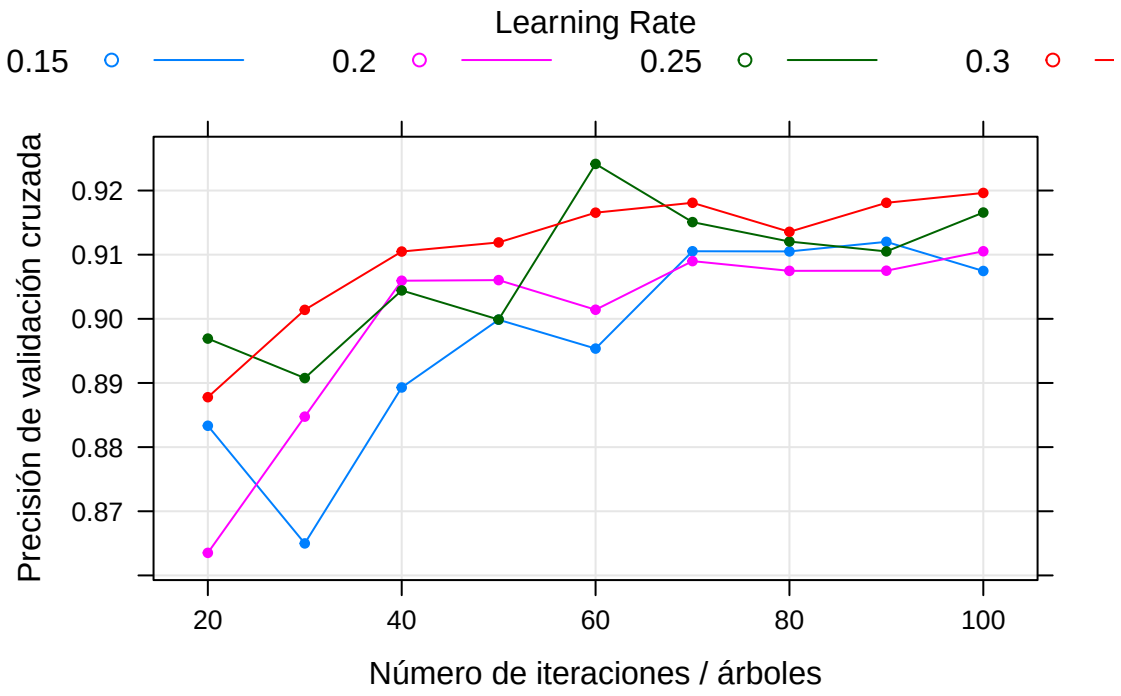
Para optimizar los restantes hiperparámetros emplearemos validación cruzada. Los valores entre los que se van a escoger son:

```
grid = expand.grid(maxdepth=1, iter=seq(20,100,10),
                   nu=c(0.15,0.2,0.25,0.3))
# La documentación del paquete ada indica que se deben especificar además los siguientes
# parámetros usando rpart para la utilización de stumps.
rcontrol = rpart.control(maxdepth=1,cp=-1,minsplit=0)
```

Una vez establecidos los valores entre los que se van a elegir los hiperparámetros, pasamos a obtener el mejor modelo y estudiar su calidad. De igual forma que en la sección anterior utlizaremos el preprocesamiento proporcionado por `train`.

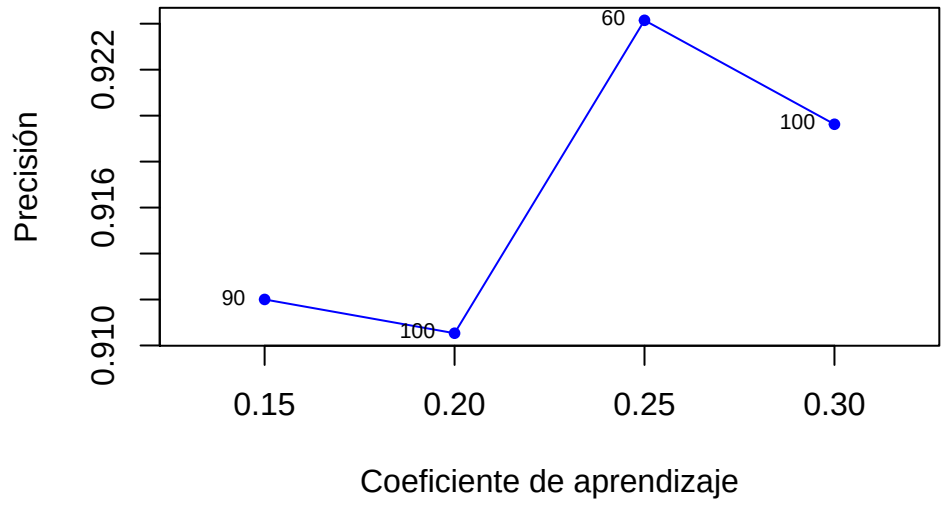
```
ada_fit = train(x = datos[indices_train,], y = as.factor(etiquetas[indices_train]),
                method = "ada", trControl = control,
                preProcess = c("YeoJohnson","center","scale"),
                tuneGrid = grid, control = rcontrol)
```

Para cada una de las combinaciones de hiperparámetros `train` proporciona la precisión del modelo asociado. Veamos los resultados de precisión para todas las combinaciones posibles:




Se observa una tendencia al alza en la precisión de la validación cruzada conforme aumentamos el número de iteraciones (o clasificadores simples distintos). Esto probablemente sea debido al *underfitting* que se da de forma natural en muchos casos cuando el modelo que construye *Boosting* aún no tiene la complejidad suficiente para explicar los datos.

Para visualizar mejor los mejores resultados, en la siguiente gráfica se muestran los valores del coeficiente de aprendizaje frente a la mayor precisión posible, junto con el número de iteraciones que se necesitaron para obtener dicho resultado.




Como podemos observar, la mayor precisión se obtiene con un coeficiente de aprendizaje igual a 0.25 con 60 iteraciones; por ello, éste sería el modelo a elegir en principio.

### 3.2.3. *Support Vector Machines*

El tercer modelo a estudiar será *SVM*. Frente a modelos lineales como el perceptrón o las regresiones, *SVM* busca construir un separador con mayor poder de generalización; logra esto buscando un separador cuya separación (margen) respecto a los puntos más cercanos (vectores de soporte) sea máxima. El razonamiento subyacente se basa en que los nuevos datos que debamos clasificar serán presumiblemente similares a los de entrenamiento; por lo tanto, si nuestro clasificador se acerca demasiado a algunos puntos, hay cierta probabilidad de que una nueva observación cercana a ellos sea etiquetada de forma incorrecta.

Análogamente a los modelos lineales mencionados, *SVM* tiene su propia forma de tratar con datos no linealmente separables: los *kernels*. Un *kernel* es una función que determina la similitud entre dos observaciones de forma equivalente a medir la distancia entre ellas en un espacio diferente al original. Nosotros usaremos el *kernel RBF* gaussiano, cuya expresión se muestra tras estas líneas, que tiene un hiperparámetro $\gamma$ a ajustar. En el paquete `kernlab` que hemos utilizado a través de `train` este hiperparámetro es llamado $\sigma$.

$$K(x, x') = e^{-\gamma||x-x'||^2}$$

Además, ya que los datos en la mayoría de los casos no son linealmente separables, `kernlab` nos permite ajustar el parámetro $C$, que controla la complejidad del modelo de forma similar a la regularización; en este caso, valores altos significan hipótesis más precisas que minimicen el error $\xi$ de cada punto (distancia al margen de su clase correcta; $\xi = 0$ si es un vector de soporte o está bien clasificado más allá de su margen; $0 < \xi < 2$ si está dentro de alguno de los dos márgenes; $\xi \geq 2$ si está mal clasificado más allá del margen opuesto).

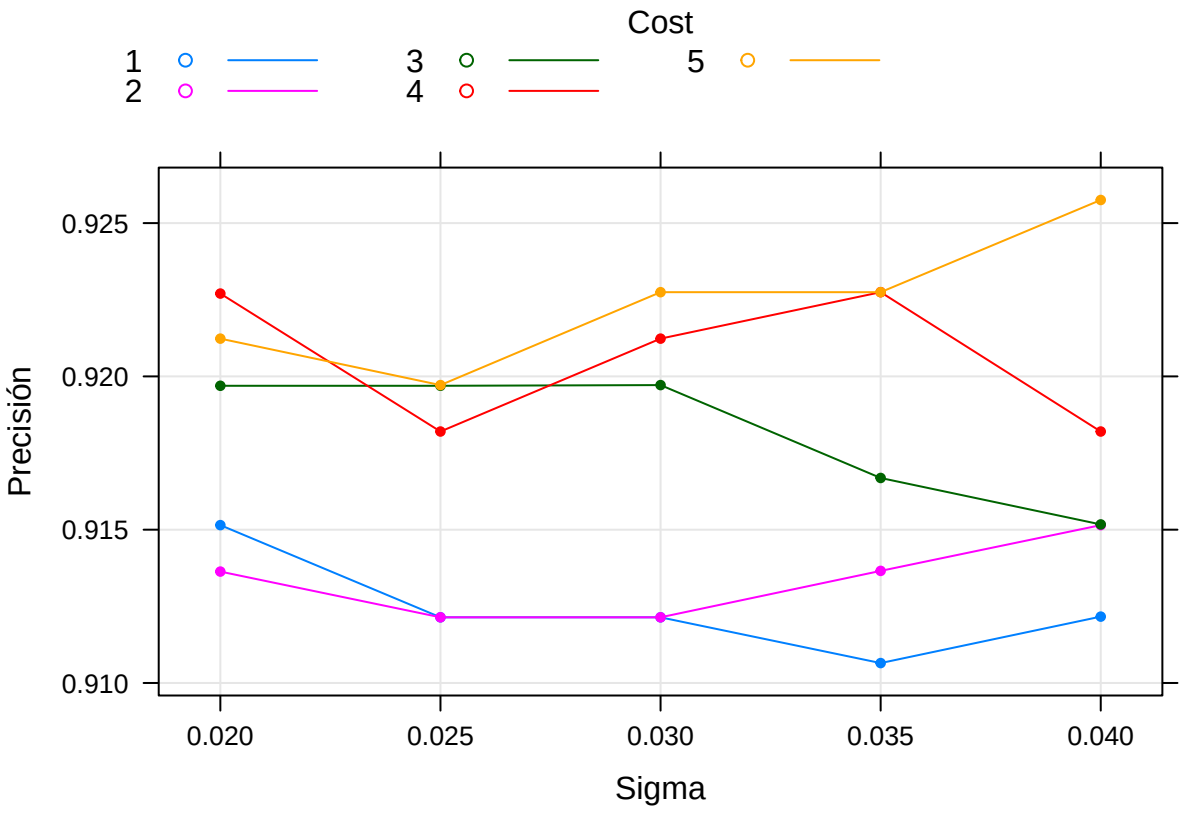
Por medio de validación cruzada estimaremos los parámetros $C$ y $\sigma$ para el modelo que nos ocupa. Como en las secciones anteriores, se realizarán 10 particiones de validación. Los valores entre los que se elegirán se definen en la siguiente línea de código:

```
grid = expand.grid(C=seq(1,5,1), sigma=seq(0.02, 0.04, 0.005))
```

Una vez hecho esto, veamos la llamada correspondiente a la función `train`. El método a utilizar es `svmRadial`; el resto de los parámetros se han comentado en modelos anteriores.

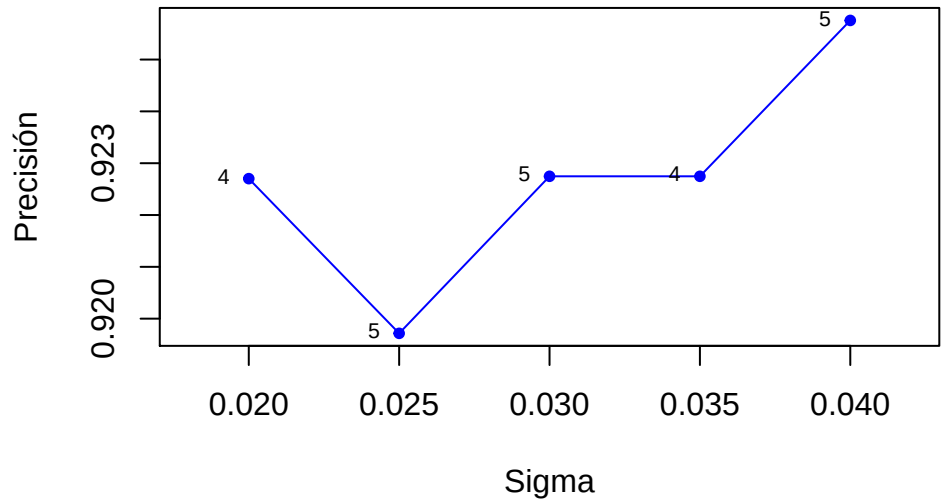
```
svm_fit = train(x = datos[indices_train,], y = as.factor(etiquetas[indices_train]),
                method = "svmRadial", trControl = control,
                preProcess = c("YeoJohnson","center","scale"), tuneGrid = grid)
```

Observemos las diferentes precisiones obtenidas por todas las combinaciones de valores de $\sigma$ y $C$:




Como se puede apreciar, valores más altos de $C$ resultan por lo general en ajustes de validación cruzada con una calidad un poco mayor. Esto concuerda con la explicación dada anteriormente.

Finalmente, vamos a analizar la máxima precisión posible con cada valor de $\sigma$, junto con sus valores de $C$ asociados:




Como se puede apreciar en la gráfica anterior, la mayor precisión (alternativamente, el menor error) de validación cruzada se obtiene con $\sigma = 0.04$ y $C = 5$. Esta combinación de hiperparámetros será, pues, la elegida.

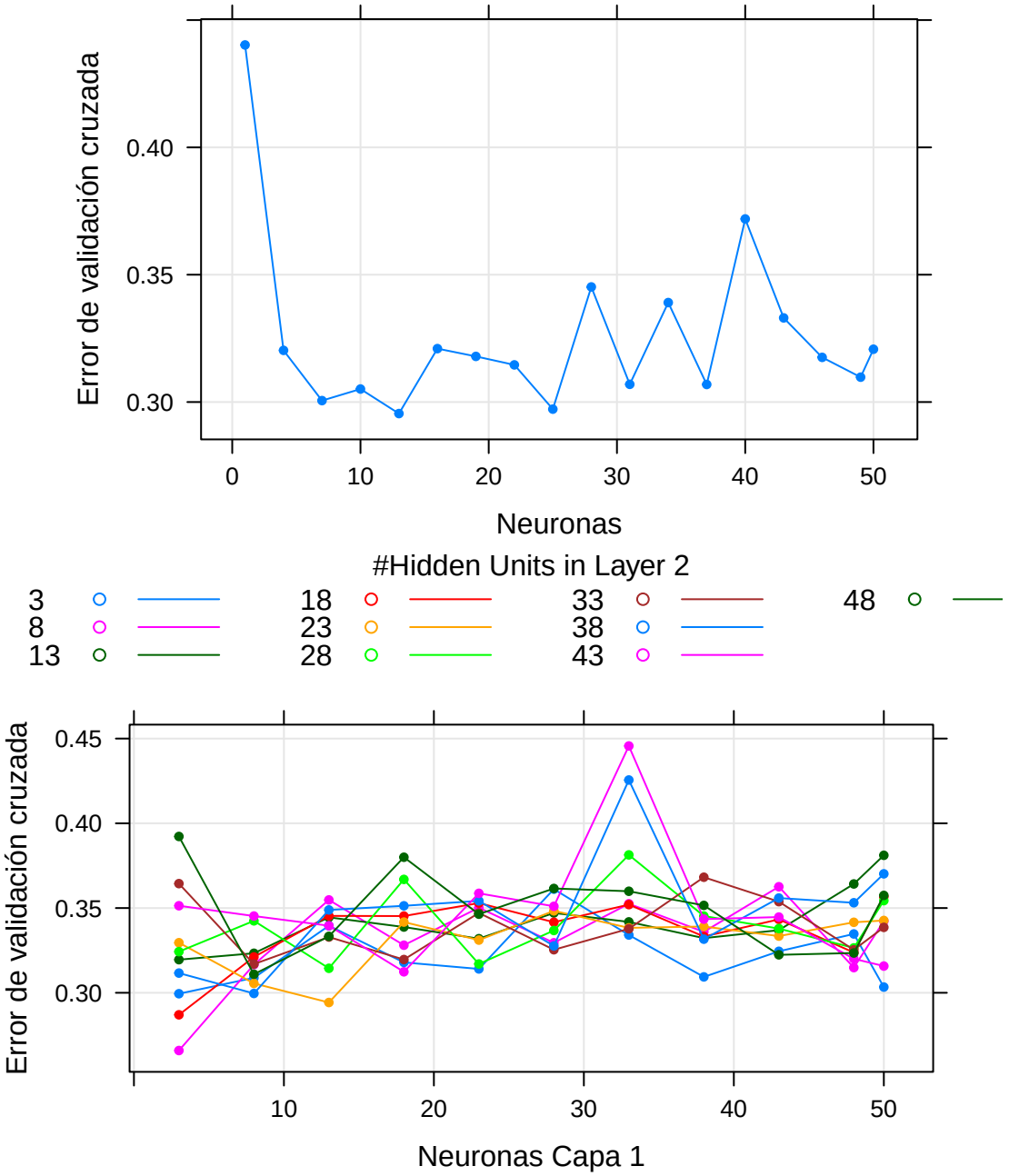
### 3.2.4. Redes Neuronales

En este último apartado sobre modelos no lineales vamos a hablar de las redes neuronales. Las redes neuronales son una generalización del perceptrón. Puesto que es un modelo muy potente, es también muy sensible al sobreajuste. Por consiguiente, dado que para nuestro problema una simple regresión logística ya consigue un error bajo, es probable que una red neuronal aporte complejidad innecesaria.

Las redes neuronales pueden presentar distintas arquitecturas según el número de capas ocultas y el número de neuronas de cada una de éstas. En nuestro caso, probaremos redes neuronales con una, dos o tres capas ocultas que contendrán neuronas en el rango [1,50]. El peso inicial de cada neurona se asigna aleatoriamente.

```
grid = expand.grid(layer1=c(seq(1,50,3),50), layer2=0, layer3=0)
nn_fit_una = train(x = datos[indices_train,], y = etiquetas[indices_train],
                   method = "neuralnet", linear.output=FALSE, trControl = control,
                   preProcess = c("YeoJohnson","center","scale"), tuneGrid = grid)
grid = expand.grid(layer1=c(seq(3,50,5),50), layer2=seq(3,50,5), layer3=0)
nn_fit_dos = train(x = datos[indices_train,], y = etiquetas[indices_train],
                   method = "neuralnet", linear.output=FALSE, trControl = control,
                   preProcess = c("YeoJohnson","center","scale"), tuneGrid = grid)
grid = expand.grid(layer1=c(seq(1,50,10),50), layer2=seq(1,50,10), layer3=seq(1,50,10))
nn_fit_tres = train(x = datos[indices_train,], y = etiquetas[indices_train],
                    method = "neuralnet", linear.output=FALSE, trControl = control,
                    preProcess = c("YeoJohnson","center","scale"), tuneGrid = grid)
```

Ahora representaremos de forma gráfica el resultado de la validación cruzada para una y dos capas:




El eje Y en ambos casos representa el error obtenido en la validación cruzada y el eje X el número de neuronas existentes en la primera capa oculta. Puesto que la segunda gráfica corresponde a una arquitectura de dos capas, las distintas líneas representan el número de neuronas de la segunda capa oculta. En la primera observamos que el menor error se obtiene con 13 neuronas y en la segunda con 3 y 8 neuronas por capa.

Los mejores resultados obtenidos por cada una de las tres variantes se pueden ver en la siguiente tabla:

| Número de capas | Neuronas por capa | $E_{CV}$ |
|---|---|---|
| Una capa | 13 | 1.6667 |
| Dos capas | 3, 8 | 5.4545 |
| Tres capas | 11, 31, 20 | 1.5151 |

Guiándonos por los números, la elección parece ser una red neuronal de tres capas. Sin embargo, ya que el rendimiento de la redes neuronales depende mucho de su arquitectura y no es sencillo encontrar un equilibrio entre potencia de ajuste a los datos de entrenamiento y generalización, en esta ocasión hemos decidido comparar las tres redes neuronales con los otros tipos de modelos.

Por otra parte, los experimentos realizados en este proyecto nos indican que el coste en términos de tiempo es prohibitivo: calcular estas redes neuronales ha consumido más de 8 horas, frente a pocos minutos en los demás casos. Por ello, aun si observásemos mejores resultados que los de sus competidores, probablemente no compensaría el esfuerzo computacional añadido.

# 4. Selección del modelo final

## 4.1. Comparativa

Tras haber estudiado individualmente los modelos, es el momento de decantarnos por uno de ellos.

En primer lugar vamos a ver los $E_{in}$ y $E_{test}$ conseguidos por cada uno:

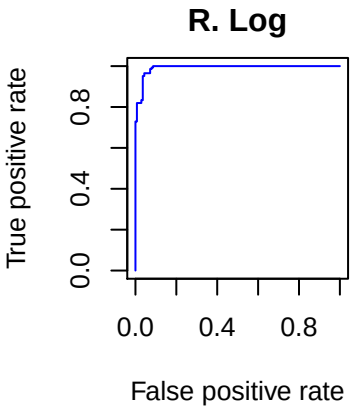
| Modelo | $E_{in}$ | $E_{test}$ |
|---|---|---|
| Regresión Logística | 5.9091 | 4.5936 |
| Random Forest | 0 | 6.3604 |
| Boosting | 8.3333 | 6.7137 |
| Support Vector Machine | 1.3636 | 8.8339 |
| Red neuronal de una capa | 1.6667 | 7.7738 |
| Red neuronal de dos capas | 5.4545 | 9.5406 |
| Red neuronal de tres capas | 1.5151 | 8.4805 |

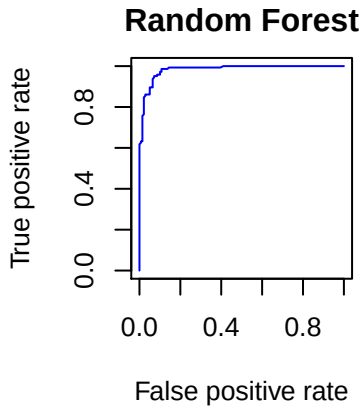
De esta tabla se extrae que la regresión logística es más que suficiente para predecir con alta fiabilidad nuevos datos; de hecho, parece realizar un ajuste más consistente que los modelos no lineales. Para confirmar esto, vamos a analizar también las curvas *ROC* y el área debajo de las mismas. Este último dato puede verse en la siguiente tabla:

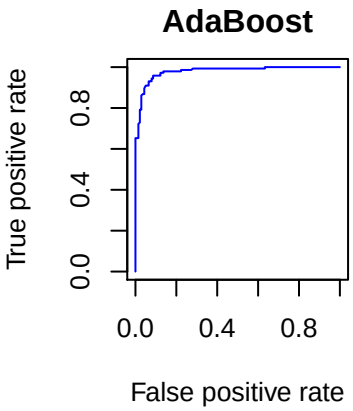
| Modelo | Area Under Curve |
|---|---|
| Regresión Logística | 0.9914568345 |
| Random Forest | 0.9774930056 |
| Boosting | 0.9792915667 |
| Support Vector Machine | 0.9732713829 |
| Red neuronal de una capa | 0.9581834532 |
| Red neuronal de dos capas | 0.9460431655 |
| Red neuronal de tres capas | 0.9407973621 |

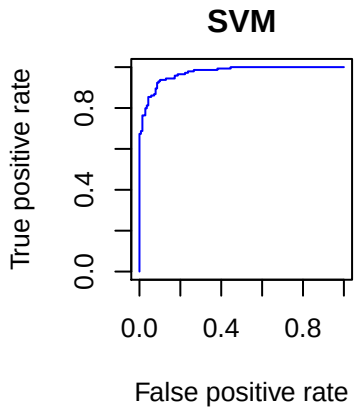
Comprobamos que, una vez más, la regresión logística obtiene el mejor valor. La interpretación es la siguiente: este modelo tiene mayor precisión cuando consideramos todos los posibles umbrales a partir de los cuales decide clasificar positivamente una observación. Hay que destacar que los valores obtenidos por los otros modelos indican también una alta calidad general, pero los dos criterios que hemos usado apuntan en la misma dirección: la regresión logística es el modelo a elegir.

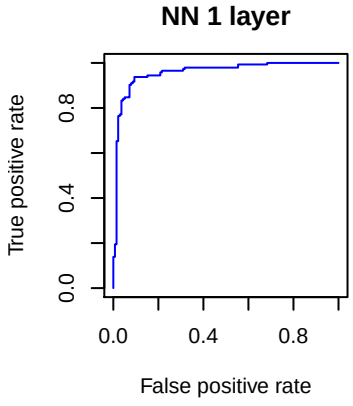
Con el objetivo de ofrecer mayor claridad acerca del significado de los valores de la tabla anterior, a continuación se ve una representación de las curvas *ROC* de todos los modelos:

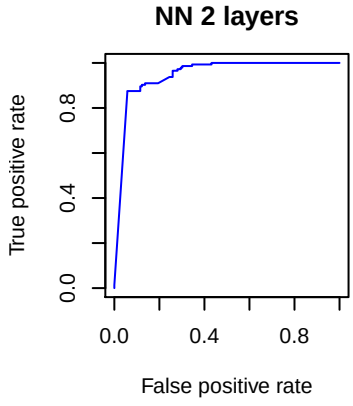


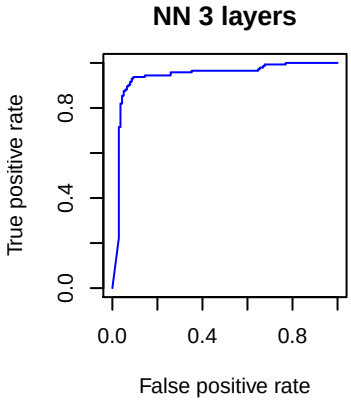



## 4.2. Descripción del modelo elegido

El modelo final será, como hemos mencionado anteriormente, la regresión logística. A continuación, vamos a presentar las características que han intervenido en el ajuste final junto con sus pesos y los valores utilizados en los distintos métodos de preprocesamiento:

| Var. | Peso | Centrado | Escalado | Yeo-Jonhson | Var. | Peso | Centrado | Escalado | Yeo-Jonhson |
|---|---|---|---|---|---|---|---|---|---|
| FSW.1 | 3.5753 | 6.9361 | 1.4599 | 0.3273 | SSP.2 | -5.1918 | 162.4274 | 49.1844 | 1.4974 |
| SSW.1 | 1.4204 | 4.3006 | 1.1066 | 0.2869 | SSW.2 | -1.0284 | 4.6731 | 1.3173 | 0.3366 |
| BPC.1 | 1.8938 | 2.2502 | 1.1279 | 0.3069 | BPC.2 | -1.7250 | 2.2651 | 1.1205 | 0.3316 |
| BPW.1 | 0.9310 | 3.0814 | 1.5783 | 0.4139 | BPW.2 | -0.8471 | 3.0681 | 1.5396 | 0.4159 |
| TPW.1 | 15.7692 | 7.9018 | 5.7159 | 0.3695 | NPW.2 | 0.2786 | 4.6566 | 2.4294 | 0.3841 |
| FSP.2 | -4.882 | 5.3850 | 0.2134 | 0.1221 | TPW.2 | -15.854 | 8.0051 | 5.7826 | 0.3730 |
| FSW.2 | -4.401 | 6.1526 | 1.1877 | 0.2717 | Intercept | 0.02848 | | | |

Para obtener un estimador del error fuera de la muestra vamos a emplear el error de test y la desigualdad de *Hoeffding*:

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

Donde $N$ es el número de observaciones de la partición de test y $\mathbb{P}[|E_{test}(g) - E_{out}(g)| > \epsilon]$ corresponde a una tolerancia $\delta$ que fijaremos a 0.05.

$$\delta \leq 2e^{-2N\epsilon^2}$$

Junto con el error en la partición de test obtendremos, despejando $\epsilon$ de la siguiente forma, un intervalo en el que se encuentra $E_{out}$ con una tolerancia $\delta = 0.05$.

$$\sqrt{\frac{log(2) - log(\delta)}{2N}} \geq \epsilon$$

La cota final obtenida es $E_{test} + \epsilon = 4.5936 + 0.0807 = 4.6743$. Ya que se trata de un valor bajo para $\epsilon$, podemos decir que la partición de test representa lo suficientemente bien el comportamiento del resto de la población.

Tras realizar este estudio y pese a la potencia de los modelos que se han propuesto, hay clara evidencia de que modelos más sencillos pueden realizar un mejor ajuste si los datos presentan tendencia lineal. En este caso es sensato afirmar ocurre, aunque la naturaleza del problema no lo sugiriera.