Clasificación y evaluación: el paquete `caret`

José Tomás Palma Méndez

Dept. of Ingeniería de la Información y las Comunicaciones.Universidad de Murcia.
Contacting author: `jtpalma@um.es`

1. El paquete `caret`

El paquete `caret` (classification and regresion training) nos proporciona una serie de funciones que facilitan la resolución de problemas complejos de clasificación y regresión. El paquete `caret` utiliza funciones para construir modelos de clasificación y regresión proporcionadas por otros paquetes, cargando estos sólo cuando son necesarios, siempre y cuando estén instalados. Para instalar el paquete `caret` y el resto de paquetes que se necesitan basta con ejecutar la siguiente instrucción:

```
> install.packages("caret", dependencies = c("Depends", "Suggests"))
> library(caret)
```

A través de las funciones ofrecidas por el paquete `caret` podemos realizar todos los pasos necesarios para construir un modelo de clasificación: visualización y análisis inicial de los datos, preprocesamiento, selección de variables, optimización de parámetros y evaluación. [1]

2. Visualización

Para este guión vamos a utilizar el fichero generado en la fase de preprocesamiento al que le aplicamos la imputación por el método kNN.Una vez cargada la tabla con los datos, podemos analizar los datos mediante gráficos de dispersión. Para realizar estos gráficos hay que utilizar la función `featurePlot()`:

```
> featurePlot(x = hepatitis.KnnImp[, 1:4],
              y = hepatitis.KnnImp$PRONOSTICO,
              plot = "pairs",auto.key = list(columns = 2))
```

El parámetro `x` indica las características que vamos a representar (en este caso las cuatro primeras), el parámetro `y` se utiliza para indicar cuál es la variable que define la clase de cada elemento, el parámetro `plot` indica el tipo de gráfica: `pairs`, `box`,

[1] Una lista con las distintas técnicas de clasificación y regresión que podemos utilizar a través del paquete `caret`, así como los parámetros que requiere cada técnica, se puede ver en http://caret.r-forge.r-project.org/modelList.html

`density`, `strip` o `ellipse`, en el caso de problemas de clasificación; `pairs` o `scatter` en el caso de problemas de regresión. El parámetro `auto.key` se utiliza para indicar que queremos mostrar el código de colores asociados a las clases. Esto se conseguiría con `auto.key=TRUE`. En nuestro ejemplo, además de colocar la leyenda, le estamos indicando que las coloque en una fila de dos columnas (por defecto las coloca en la misma columna). La figura 1 muestra el resultado del comando anterior.

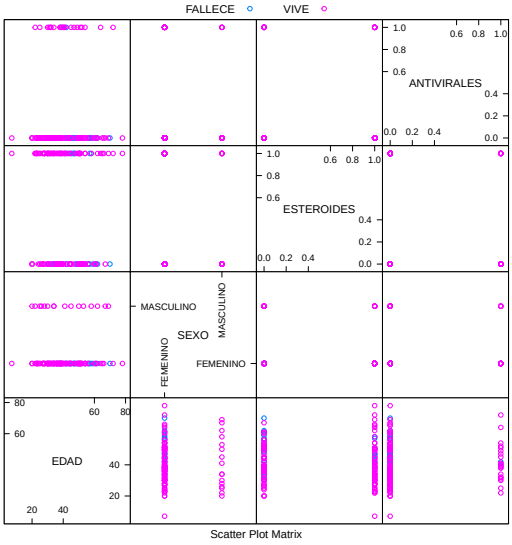

Figura 1. `featurePlot` con la opción `pairs`

Ejercicio 1. Genera un gráfico de dispersión para las variables numéricas FOSFATOalc, SGOT, ALBUMINA Y PROTIME.

1.a) Prueba todas las opciones del parámetro `pairs`, `box`, `strip`, `density` y `ellipse`.

1.b) ¿Se pueden extraer alguna conclusión de las gráficas anteriormente generadas?

3. Preprocesamiento

El paquete `caret` también nos ofrece funciones que permiten realizar aplicar algunas técnicas de preprocesamiento. Concretamente, podemos realizar transformaciones de escala, centrado y reducción de la dimensionalidad por medio del análisis de componentes principales (ACP) o análisis de componentes independientes (ACI).

Antes de empezar con las tareas de procesamiento, tenemos que dividir el conjunto inicial de datos en dos conjuntos: entrenamiento (para crear los clasificadores) y prueba (para evaluarlos). Esto lo podemos realizar mediante la función `createDataPartition()`. Esta función realiza un muestreo de los datos para cada una de las clases intentando preservar la distribución original de clases. Por ejemplo, para crear una partición del conjunto de datos en dos conjuntos, uno con el 66 % de los datos (entrenamiento) y otro con el 30 % (prueba), bastaría con:

```
> set.seed(342)
> trainIndex <- createDataPartition(hepatitis.KnnImp$PRONOSTICO,
                                    p = 0.66,
                                    list = FALSE,
                                    times = 1)
> hepatitisTrain <- hepatitis.KnnImp[trainIndex,]
> hepatitisTest <- hepatitis.KnnImp[-trainIndex,]
```

El primer parámetro indica en qué columna se encuentra la información sobre a qué clase pertenece cada objeto. El parámetro `p=0.66` indica la proporción de objetos que vamos a seleccionar. Con `list=FALSE` indicamos que no queremos que el resultado sea una lista, sino que devolverá un vector con los índices seleccionados. El parámetro `times` nos permite crear múltiples particiones. Una vez obtenido el vector con los índices seleccionados, ya podemos realizar la partición de los datos, seleccionando para el conjunto de entrenamiento aquellos objetos cuyos índices están incluidos en el vector de índices.

Para tareas de procesamiento vamos a utilizar la función `preProcess()`. Esta función determina los parámetros necesarios para realizar la transformación requerida, con lo que para transformar realmente los datos hay que utilizar posteriormente la función `predict()`. La función `preProcess` tiene los siguientes parámetros:

- `x`: matriz o data frame de datos. En el caso de clasificación hay que eliminar la columna que especifica la clase.
- `method`: que indica el método de preprocesamiento que vamos a utilizar: `BoxCox`, `center`, `scale`, `range`, `knnImpute`, `bagImpute`, `pca`, `ica` and `spatialSign`.
  - En el caso de realizar un análisis de componentes principales, (`method=pca`), tenemos que indicarle el número de componentes que vamos a utilizar, bien indicando un umbral para especificar el punto de corte del porcentaje acumulado de varianza, `tresh`, o indicando directamente el número de componentes que queremos, `pcaComp`. Las componentes recibirán los nombres `PC1`, `PC2`, ...
  - En el caso de realizar un análisis de componente independientes, (`method=ica`), necesitaremos indicar el número de componentes que queremos, `n.comp`. Las componentes recibirán los nombres `IC1`, `IC2`, ...

- Para el caso de la imputación de valores, (`method=knnImpute`), necesitaríamos indicar el número de vecinos más cercanos a utilizar, `k`, además sólo se podrá aplicar para atributos numéricos siempre y cuando no existan elementos con todos sus valores desconocidos.

Por ejemplo, la siguiente instrucción nos permite reducir el número de características mediante ACP, seleccionando el conjunto de componentes principales que acumulen el 95 % de la varianza:

```
> hepatitisPCA <- preProcess(hepatitis.KnnImp[1:ncol(hepatitis.KnnImp)-1],
                             method = "pca",thresh = 0.95)
> print(hepatitisPCA)

Created from 155 samples and 19 variables

Pre-processing:
  - centered (6)
  - ignored (13)
  - principal component signal extraction (6)
  - scaled (6)

PCA needed 6 components to capture 95 percent of the variance
```

Obsérvese que para efectuar el ACP hemos eliminado la última columna de la matriz de datos que es la que contiene la información sobre las clases. Una vez realizado el proceso ACP, sólo tenemos que transformar los datos y volver a añadir la columna que define las clases de los objetos:

```
> PCATrain <- predict(hepatitisPCA,hepatitisTrain[,1:ncol(hepatitisTrain)-1])
> PCATest <- predict(hepatitisPCA,hepatitisTest[,1:ncol(hepatitisTest)-1])
> PCATrain <- data.frame(PCATrain,hepatitisTrain$PRONOSTICO)
> PCATest <- data.frame(PCATest,hepatitisTest$PRONOSTICO)
```

La función `preProcess` nos permite aplicar varios métodos al mismo tiempo. Para ello bastaría con definir un vector con los distintos métodos, `method=c("center", "scale")`. De todas formas, hay que tener en cuenta que ciertos métodos requieren de la aplicación de varios métodos de preprocesamiento. Por ejemplo, el ACP e ICA requieren de un escalado y un centrado previo de los datos. Hay que tener en cuenta que el procesamiento sólo se aplica a atributos numéricos, los atributos no numéricos son añadidos a las componentes extraídas.

El paquete `caret` también nos ofrece funciones que nos permiten determinar las características que poseen varianza cercana a cero, `nearZeroVar()`, identificar características correladas, `findCorrelation()` o detectar dependencias lineales, `findLinearCombos()`.

Ejercicio 2.

2.a) ¿Cómo podríamos generar un conjunto de datos qué solo contenga las componentes principales seleccionadas y el atributo que identifica la clase? Realiza el proceso para los conjuntos de entrenamiento y prueba.
2.b) Crea un gráfico en el qué se muestren las cuatro primeras componentes principales del conjunto de entrenamiento.
2.c) Repite el proceso seguido para el ACP para determinar las componentes independientes, tanto para el conjunto de entrenamiento como el de prueba.
2.d) Crea un gráfico en el qué se muestren las cuatro primeras componentes principales del conjunto de entrenamiento.
2.e) Al realizar el análisis ACP o ICA ¿Qué otros métodos de preprocesamiento se han aplicado?
2.f) ¿Existe alguna variable que presente una varianza cercana a 0?

4. Selección de variables

El paquete `caret` nos permite seleccionar variables mediante cuatro métodos. Tres son de tipo wrapper: eliminación recursiva de variables y dos con búsqueda aleatoria: algoritmos genéticos y enfriamiento simulado. El otro método que queda es un filtro de tipo ranker.

4.1. Eliminación recursiva de variables

La eliminación recursiva de variables se realiza a través de la función `rfe`. Para poder utilizar la función `rfe` necesitamos crear un objeto de control que defina alguno de los parámetros del algoritmo de selección de variables:

```
> ctrl.rfe <- rfeControl(functions=rfFuncs,
                         method = "cv",
                         number = 5,
                         returnResamp="final",
                         verbose = TRUE)
```

El parámetro `functions` indica qué tipo de clasificador se va a utilizar para evaluar los distintos conjuntos de variables. Los posibles valores son: regresión lineal (`lmFuncs`), random forests (`rfFuncs`), naive Bayes (`nbFuncs`), bagged trees (`treebagFuncs`) y cualquier clasificador accesible a través del paquete `caret` (`caretFuncs`). En este ultimo caso debemos especificar el método a utilizar con el parámetro `method` en la función `rfe`.

A través del parámetro `method` indicamos el método de evaluación a utilizar: `boot` y `boot.632` para bootstrap, `cv` para validación cruzada (en este caso deberemos indicar el número de particiones a utilizar con el parámetro `number`), `repeatedcv` para

validación cruzada repetida (además de `number` hay que indicar el número de repeticiones mediante el parámetro `repeats`). `LOOCV`, para leave-one-out cross validation o `LGOCV`, para leave-group cross validation (el parámetro `p` indicará el porcentaje de objetos que se va a utilizar para construir los conjunto de entrenamiento y test). Recordad que este último es lo mismo que Hold-out y su versión con repetición a través del parámetro `repeats`. También podemos forzar que cada vez que se elimine una variable se vuelvan a calcular la importancia de las variables (`rerank=TRUE`). El parámetro `returnResamp` nos permite indicar qué resultados de evaluación se deben almacenar: `all`, `final` o `none` y con `saveDetails` podemos indicarle que almacene la información relativa a las variables seleccionadas y su importancia.

Ahora sólo tenemos que invocar a la función que realiza la selección de variables:

```
> subsets <- c(3:19)
> set.seed(342)
> rf.rfe <- rfe(PRONOSTICO~., data=hepatitis.KnnImp,
                sizes=subsets,
                rfeControl=ctrl.rfe)
```

En la función `rfe`, los dos primeros parámetros indican la matriz de datos a utilizar y cuál es la columna con la información sobre la clase a que pertenece cada elemento. Esto lo podemos hacer también mediante la fórmula "`PRONOSTICO~., data=hepatitis.KnnImp,sizes`", que indica que la columna de clasificación es `PRONOSTICO` y que vamos a utilizar todas las variables (se pueden seleccionar las variables que se van a utilizar añadiendo las columnas que queremos con el símbolo `+`).

Con el parámetro `sizes` indicamos los tamaños de los conjuntos de variables que se tienen que probar. En nuestro caso se van a utilizar los tamaños entre 3 y 19. El objeto de control a utilizar se especifica a través del parámetro `rfeControl()`. También hay que indicar qué métrica se va a utilizar para seleccionar el modelo óptimo a través del parámetro `metric`. En el caso de problemas de regresión los valores posibles son `RMSE`, para el error cuadrático medio, y `Rsquared`, para el coeficiente de determinación. En el caso de problemas de clasificación los posibles valores son `Accuracy`, para la precisión, y el índice `Kappa`. El parámetro `maximize` nos indicará si queremos que la métrica se maximice o minimice.

La función `rfe()` nos devuelve el objeto `rf.rfe` que tiene la siguiente información.

```
> rf.rfe

Recursive feature selection

Outer resampling method: Cross-Validated (5 fold)

Resampling performance over subset size:

 Variables Accuracy  Kappa AccuracySD KappaSD Selected
         3   0.8981 0.6766    0.05880 0.19693
         4   0.8844 0.6565    0.04027 0.11466
         5   0.8844 0.6525    0.05979 0.18908
```

```
         6   0.8973 0.6949    0.05037 0.14482
         7   0.8911 0.6554    0.03358 0.10529
         8   0.8977 0.6853    0.05506 0.15627
         9   0.9106 0.7315    0.05929 0.16933
        10   0.9044 0.6999    0.04811 0.14903
        11   0.9167 0.7431    0.02624 0.07244
        12   0.9046 0.7108    0.06122 0.18410
        13   0.9175 0.7410    0.05569 0.17410
        14   0.9238 0.7668    0.04630 0.14460        *
        15   0.9108 0.7182    0.05004 0.17190
        16   0.9171 0.7448    0.04063 0.12606
        17   0.9173 0.7354    0.05576 0.18783
        18   0.9173 0.7354    0.05576 0.18783
        19   0.9108 0.7215    0.05004 0.15815

The top 5 variables (out of 14):
   PROTIME, ALBUMINA, ASCITISTRUE, BILIRRUBINA, VARICESTRUE

> rf.rfe$optVariables

 [1] "PROTIME"          "ALBUMINA"
 [3] "ASCITISTRUE"      "BILIRRUBINA"
 [5] "VARICESTRUE"      "ARANIASvascTRUE"
 [7] "SEXOMASCULINO"    "FATIGATRUE"
 [9] "FOSFATOalc"       "MALAISETRUE"
[11] "HIGgrandeTRUE"    "HISTIOLOGIATRUE"
[13] "HIGfirmeTRUE"     "EDAD"
```

El objeto `rf.rfe` nos indica que el mejor clasificador que clasifica los objetos del conjunto de entrenamiento contiene 14 variables. Dichas variables quedan almacenadas en el objeto `rf.rfe$optVariables`, lo que nos permite volver a generar los conjuntos de entrenamiento y prueba, pero sólo con las variables seleccionadas.

```
> sel.cols <- c(rf.rfe$optVariables,"PRONOSTICO")
> hepatitisTrain.sel <- hepatitisTrain[,sel.cols]
> hepatitisTest.sel <- hepatitisTest[,sel.cols]
```

> Ejercicio 3. ¿Por qué no funcionan las operaciones anteriores? ¿Que hay que hacer para resolverlo?

También podemos acceder al clasificador más óptimo de todos los que se han probado. Esta información está en el objeto `fit` incluido dentro de `rf.rfe`:

```
> rf.rfe$fit

Call:
 randomForest(x = x, y = y, importance = first)
```

```
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 3

        OOB estimate of  error rate: 8.39%
Confusion matrix:
        FALLECE VIVE class.error
FALLECE      24    8  0.25000000
VIVE          5  118  0.04065041
```

Este objeto lo podemos utilizar para realizar predicciones o evaluar el conjunto de prueba. Los resultados del proceso de selección de variables también pueden ser mostrados por medio de gráficas creadas a partir de la función `plot` (figura 2).

```
> plot(rf.rfe$fit)
```

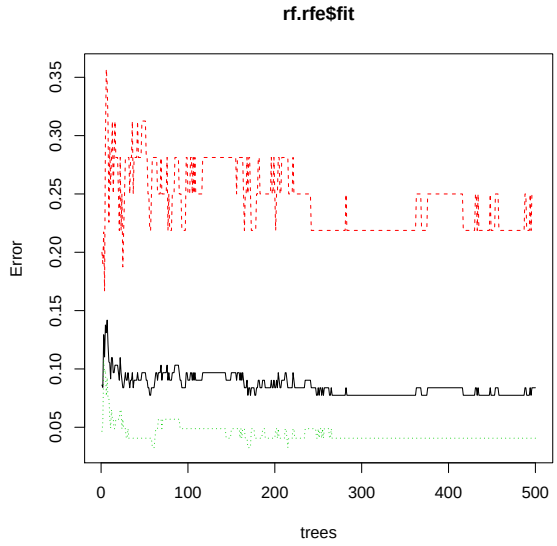

Figura 2. Error Out of the Bag en el conjunto de datos (verde), la clase VIVE (negro) y la clase FALLECE (rojo).

Ejercicio 4. Realiza una selección de variables utilizando la función `rfe` con las funciones `treebagFuncs` y `nbFuncs`.

4.a) ¿Qué técnicas de clasificación utilizan? ¿Cuántas variables seleccionan cada técnica?

4.b) Compara estos resultados aplicando la misma técnica pero con las técnicas de clasificación `svmLinear` y `rpart` (recuerda que esto se debe hacer a través de la opción `functions=caretFuncs` y el parámetro `method` de la función `rfe`.

4.2. Eliminación de variables por filtros

En `caret` el filtro está implementado a través de la función `sbf()`. Esta función implementa un test estadístico. En el caso de problemas de clasificación se utiliza un test ANOVA en el que la hipótesis nula es que el valor medio de la variable es el mismo para todas las clases. En problemas de regresión, se aplica un modelo aditivo generalizado para ajustar los valores de la variable a los del resultado. En ambos casos se utiliza el valor p como puntuación sobre la relevancia de la variable. En definitiva, lo que se trata es de encontrar aquellas variables que presenten diferencias significativamente estadísticas entre las distintas clases o el resultado.

Su utilización es muy parecida a la de la eliminación recursiva de variables:

```
> ctrl.ranker <- sbfControl(functions = ldaSBF,
                            method = "cv", number = 5)
> set.seed(234)
> rf.ranker <- sbf(PRONOSTICO~., data=hepatitis.KnnImp,
                   sbfControl = ctrl.ranker)
> rf.ranker

Selection By Filter

Outer resampling method: Cross-Validated (5 fold)

Resampling performance:

 Accuracy  Kappa AccuracySD KappaSD
   0.8648 0.5726    0.03287  0.1083

Using the training set, 13 variables were selected:
   EDAD, SEXO, FATIGA, MALAISE, BAZOpalpa...

During resampling, the top 5 selected variables (out of a possible 14):
   ALBUMINA (100%), ARANIASvasc (100%), ASCITIS (100%), BAZOpalpa (100%),
   BILIRRUBINA (100%)

On average, 12.6 variables were selected (min = 12, max = 13)
```

En este caso, la función definida en el parámetro `functions` indica qué técnica clasificación o regresión se va a utilizar para evaluar la eficacia de las variables seleccionadas en un clasificador. Las funciones que están disponibles son: `ldaSBF`, `rfSBF`,

nbSBF y nbSBF. En el objeto rf.ranker$fit podemos ver información sobre la importancia de las variables.

```
> rf.ranker$fit

Call:
lda(x, y)

Prior probabilities of groups:
  FALLECE      VIVE
0.2064516 0.7935484

Group means:
            EDAD     SEXO    FATIGA   MALAISE
FALLECE 46.59375 1.000000 0.9375000 0.7187500
VIVE    39.79675 1.130081 0.5691057 0.3089431
        BAZOpalpa ARANIASvasc    ASCITIS    VARICES
FALLECE 0.4062500   0.7187500 0.46875000 0.37500000
VIVE    0.1463415   0.2357724 0.04878049 0.05691057
        BILIRRUBINA FOSFATOalc ALBUMINA  PROTIME
FALLECE    2.446875   38.96875 3.143750 40.75000
VIVE       1.139024   55.28455 3.979675 69.55285
        HISTIOLOGIA
FALLECE   0.7812500
VIVE      0.3658537

Coefficients of linear discriminants:
                     LD1
EDAD        -0.0065942440
SEXO         0.9473449743
FATIGA       0.1992066778
MALAISE     -0.3571488940
BAZOpalpa   -0.3886910636
ARANIASvasc -0.4077920511
ASCITIS     -1.0551982291
VARICES     -0.1564351905
BILIRRUBINA -0.2618962941
FOSFATOalc  -0.0009865545
ALBUMINA     0.4984390483
PROTIME      0.0254589063
HISTIOLOGIA -0.1307498740
```

Ejercicio 5. Aplica la selección por filtros utilizando las funciones de evaluación disponibles e indica cuantas variables se seleccionan en cada caso.

4.3. Eliminación de variables con búsqueda aleatoria

En Caret podemos aplicar dos métodos de selección de variables de tipo wrapper con búsqueda aleatoria: Algoritmos genéticos, a través de la función gafs() y Enfria-

miento simulado `safs()`. El funcionamiento es similar a las funciones anteriormente descritas. Primero se definen los parámetros de ejecución del proceso y después se ejecuta el proceso.

Para realizar una selección de variables utilizando algoritmos genéticos debemos de utilizar un código parecido a este:

```
> ctrl.ga <- gafsControl(functions = rfGA, method ="boot",
        returnResamp="final", verbose = TRUE)
> set.seed(342)
> rf.ga <- gafs(x = hepatitis.KnnImp[,1:ncol(hepatitis.KnnImp)-1],
        y = hepatitis.KnnImp$PRONOSTICO,
        iters = 50,
        popSize = 40,
        gafsControl = ctrl.ga)
```

Los distintos métodos de clasificación que se pueden utilizar se especifican a través del parámetro `functions` de la función `gafsControl` que puede tomar los siguientes valores: `caretGA` (que nos obliga a que en la función `gafs()` utilicemos el parámetro `method`), `rfGA` y `treebagGA`. En la función `gafs()` también podemos indicar el número de individuos por generación (`popSize`) y otros parámetros para configurar el algoritmo genético (probabilidad de mutación, probabilidad de cruzamiento, ..). El código anterior produce la siguiente salida:

```
Resample01  1 0.9677419 (12)
Resample01  2 0.9677419->0.9677419 (12->14, 62.5%)
Resample01  3 0.9677419->0.9677419 (12->13, 56.2%)
Resample01  4 0.9677419->0.9677419 (12->13, 66.7%)
Resample01  5 0.9677419->0.9741935 (12->13, 66.7%) *
Resample01  6 0.9741935->0.9677419 (13->11, 60.0%)
Resample01  7 0.9741935->0.9741935 (13-> 8, 40.0%)
Resample01  8 0.9741935->0.9741935 (13-> 8, 50.0%)
Resample01  9 0.9741935->0.9677419 (13->12, 78.6%)
Resample01 10 0.9741935->0.9741935 (13->12, 56.2%)
```

El primer campo indica el conjunto de datos utilizado para evaluación y obtenidos a paritr del conjunto de entrenamiento. El segundo campo indica la iteración. El resto de campos indica el fitness del mejor individuo y entre paréntesis el número de variables seleccionadas. Por ejemplo, en la primera línea se nos dice que en la primera generación se obtiene un mejor fitness de 0.9677419 con 12 variables. Después, en la segunda línea se nos indica que en la segunda iteración no se ha mejorado el mejor fitness de la primera, pero ahora dicho valor se consigue con 14 variables. El porcentaje indica el grado de similitud, utilizando la medida de Jaccard entre el mejor individuo de la generación anterior y el mejor de la generación actual. Las generaciones donde se producen mejoras se marcan con un "*".

La selección de variable utilizando enfriamiento se realiza a través de las funciones `safsControl()` y `safs()` y se utilizan de la misma forma que para los algoritmos

genéticos. Los distintos métodos de clasificación que se pueden utilizar son: `caretSA` (que nos obliga a que en la función `gafs()` utilicemos el parámetro `method`), `rfSA` y `treebagSA`. Por ejemplo, este proceso lo podemos lanzar de la siguiente forma:

```
> ctrl.sa <- safsControl(functions = rfSA, method ="holdout", holdout=.66,
        returnResamp="final", verbose = TRUE,
        improve = 50)

> rf.sa <- safs(x = hepatitis.KnnImp[,1:ncol(hepatitis.KnnImp)-1],
        y = hepatitis.KnnImp$PRONOSTICO,
        iters = 50,
        safsControl = ctrl.sa)
```

La salida que genera durante la ejecución es similar a la de la selección de variables utilizando un algoritmo genético.

5. Entrenamiento de clasificadores y ajuste de parámetros

El paquete `caret` proporciona varias funciones que permiten implementar todos los pasos necesarios en la construcción y evaluación de modelos. La construcción de modelos se realiza a través de la función `train()` que nos permite:

- Evaluar el efecto de los parámetros del modelo en la eficacia del clasificador.
- Elegir el modelo óptimo.
- Estimar la eficacia del modelo con un conjunto de prueba determinado.

El proceso es muy similar al proceso de selección de variables. Primero, hay que indicar a la función `train()` cómo vamos a realizar el proceso de construcción del modelo. Esto se realiza a través de la función `trainControl()` que tiene los siguientes parámetros[2]:

- `method:` que tiene los mismo valores que los comentados en la función `rfeControl`.
- `number` y `repeats`: `number` indica el número de particiones que se realizan cuando se elige como método de evaluación `cv` o el número de remuestreos con `boot` y `LGOCV`. `repeats` indica el número de veces que ser repite el proceso de `cv`.
- `p`: el porcentaje de elementos que se van a utilizar para entrenamiento cuando se elige el método `LGOCV`.
- `returnResamp`: para especificar cuanta información relativa al proceso de resampleado se va a devolver: `all`, `final` o `none`.
- `vervoseIter`: un valor lógico para indicar si se imprime información durante el proceso de aprendizaje.

Por ejemplo, el siguiente objeto `trainControl()` nos permite programar un método de entrenamiento utilizando como mecanismo de evaluación leave-group-out cross

[2] Para una lista más completa de parámetros consultar la documentación del paquete `caret`.

validation (LGOV), en el que el conjunto de entrenamiento está formado por el 75 % de los elementos del conjunto original, se construirán 10 particiones del conjunto de entrenamiento y se devolverán todos los datos obtenidos en todas las etapas de evaluación.

```
> fitcontrol <- trainControl(method = "LGOCV",p=.75,number=10,
                             returnResamp = "final",verboseIter=FALSE)
```

Una vez configurado el proceso de entrenamiento, sólo tenemos que invocar la función `train()` para comenzar el proceso de aprendizaje:

```
> set.seed(342)
> rpartFit <- train(PRONOSTICO~.,data=hepatitisTrain.sel,
                    method="rpart",
                    tuneLength=10,
                    trControl=fitcontrol)
```

Los dos primeros parámetros nos indican cuál es la columna que define la clase a la que pertenecen los objetos y qué atributos vamos a utilizar. Esta información también se habría podido dar indicando explícitamente la matriz de datos con las variables como primer parámetro y la columna que define la clase como segundo. El parámetro `method="rpart"` se utiliza para indicar qué tipo de clasificador queremos construir. Con `tuneLength=10` indicamos el tamaño del conjunto de posible valores de los parámetros utilizados para encontrar el clasificador óptimo.

El objeto creado por la función `train()`, en este caso `rpartFit`, contiene todo la información relativa al proceso de construcción del clasificador:

```
> print(rpartFit)

CART

104 samples
 14 predictor
  2 classes: 'FALLECE', 'VIVE'

No pre-processing
Resampling: Repeated Train/Test Splits Estimated (10 reps, 75%)
Summary of sample sizes: 79, 79, 79, 79, 79, 79, ...
Resampling results across tuning parameters:

  cp          Accuracy  Kappa
  0.00000000  0.940     0.8251264
  0.07070707  0.940     0.8251264
  0.14141414  0.940     0.8251264
  0.21212121  0.940     0.8251264
  0.28282828  0.940     0.8251264
```

```
  0.35353535  0.940     0.8251264
  0.42424242  0.940     0.8251264
  0.49494949  0.940     0.8251264
  0.56565657  0.920     0.7251264
  0.63636364  0.856     0.3716380

Accuracy was used to select the optimal model using
  the largest value.
The final value used for the model was cp = 0.4949495.
```

Si realizamos una gráfica del objeto a través de la función `plot`, obtenemos información sobre cómo ha evolucionado el proceso de construcción del clasificador, como se puede ver en la figura 3.

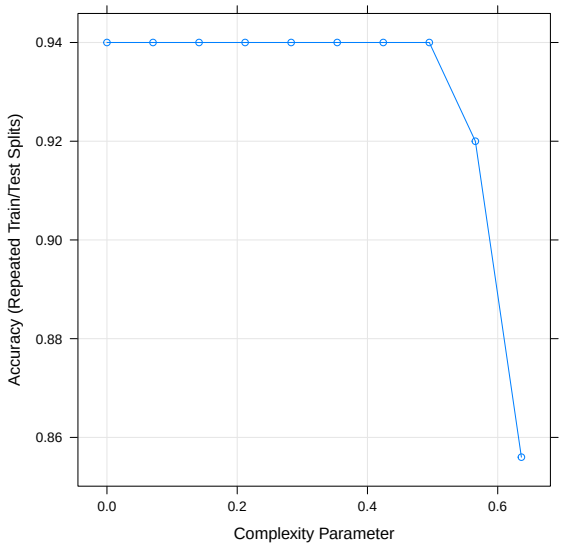

Figura 3. Evolución de la precisión del modelo en función del parámetro `cp`.

También, a través de la función `varImp()`, podemos obtener el información sobre la importancia de las variables para el proceso de clasificación. En la figura 4 se muestra de forma gráfica la importancia de las variables.

```
> varImp(rpartFit)

rpart variable importance

            Overall
PROTIME      100.00
```

```
BILIRRUBINA   42.86
ALBUMINA      42.49
VARICES       35.75
ASCITIS       31.85
HIGfirme       0.00
FATIGA         0.00
EDAD           0.00
MALAISE        0.00
HISTIOLOGIA    0.00
ARANIASvasc    0.00
SEXO           0.00
HIGgrande      0.00
FOSFATOalc     0.00
```

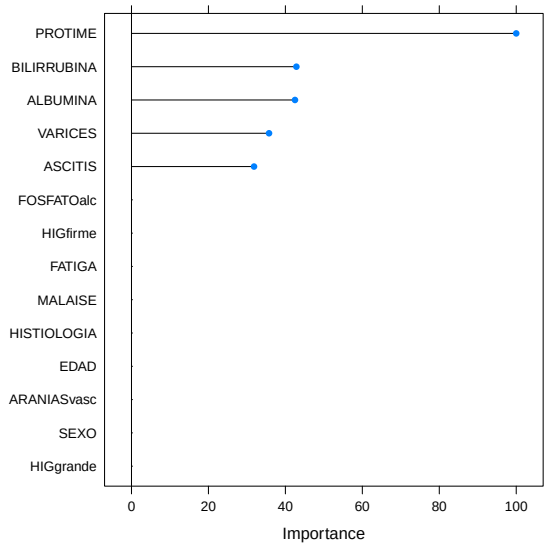

Figura 4. Importancia de las variables para el modelo `rpartFit`.

El objeto `rpartFit` también incluye el clasificador más óptimo encontrado, almacenado en el campo `finalModel`. El objeto almacenado será del tipo utilizado por el método de clasificación elegido, por lo que se podrán realizar sobre el todas las operaciones que se definan en su propia librería. Por ejemplo, en la figura 5 se puede visualizar el árbol resultante con utilizando la función `prp()`.

Ejercicio 6. Utiliza el conjunto de entrenamiento para generar los siguientes clasificadores: `C5.0`, `svmLinear`, `knn` y `lda`

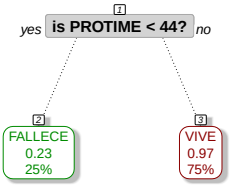

Figura 5. Árbol de clasificación obtenido dibujado con la función `prp()`.

6.a) Los nombres de cada uno de los modelos deben ser `C50Fit`, `svmFit`, `knnFit` y `ldaFit` respectivamente.
6.b) Crea una tabla en que contenga los resultados de la precisión y el índice Kappa para cada uno de los modelos. Las filas llevarán el nombre de cada uno de los modelos y las columnas el nombre del índice correspondiente.

5.1. Preprocesamiento a través de la función `train()`

También es posible definir, a través de la función `train()`, el tipo de preprocesamiento que queremos aplicar a los datos utilizando el parámetro `preProcess`. Por ejemplo, si queremos que las ajustar los valores de los atributos de nuestro conjunto de datos para que estén en el intervalo [-1, 1], podemos llamar a la función `train()` de la siguiente forma.

```
> set.seed(342)
> C50Fit <- train(PRONOSTICO~.,data=hepatitisTrain.sel,
                  method="C5.0",
                  preProc= "range",
                  trControl=fitcontrol)
```

En este caso estamos indicando que al conjunto de datos `hepatitisTrain.sel` se le aplique una transformación que centre los datos y los escale al intervalo [0,1]. En el caso de que queramos realizar un análisis de componentes principales o independientes, habría que indicar el número de componentes a utilizar por medio de los correspondientes parámetros. El preprocesamiento de datos a través de la función `train` permite que el clasificador creado tenga en cuenta la transformación realizada a la hora de realizar predicciones con conjunto de datos nuevos, es decir, será el propio

clasificador quien aplique las transformaciones necesarias liberando a usuario de dicha tarea.

Ejercicio 7.

7.a) Vuelve a generar un clasificador C50 pero aplicando un centrado y un escalado (`center` y `scale`). Repite la operación aplicando un cambio de rango (`range`) ¿Encuentras diferencias? ¿Qué opción da mejor resultado?

7.b) Vuelve a generar una máquina de soporte de vectores con kernel lineal escalando y centrando los datos ¿Encuentras diferencias respecto a la calculada en el ejercicio anterior? ¿A qué crees que es debido?

5.2. Personalización de los valores de prueba.

En el caso de que tengamos información adicional sobre cuáles serían los posibles valores de los parámetros que permitirían obtener el modelo óptimo, podemos definir dichos valores a través de la función `expand.grid()`. Por ejemplo, si queremos construir una red neuronal y sólo queremos probar configuraciones con 5, 10, 15 o 20 neuronas en la capa intermedia, con las siguientes tasas de aprendizaje: 0.5,0.1,0.00 y 0.0001, sólo tendríamos que definir un nuevo *grid*:

```
> nnetGrid <- expand.grid(.size=c(5,10,15,20),
                          .decay=c(0.5,0.1,0.001))
> set.seed(342)
> nnetFit <- train(PRONOSTICO~.,data=hepatitisTrain.sel,
                   method="nnet",
                   tuneGrid=nnetGrid,
                   trControl=fitcontrol)
```

Ejercicio 8.

8.a) Utiliza el código anterior y extiende el parámetro `size` para que busque la mejor red neuronales variando el numero de neuronas de la capa intermedia entre 10 y 20 neuronas.

8.b) Repite el proceso anterior pero realizando un centrado y escalado de los datos.

8.c) Elimina el grid definido, e introduce el parámetro `tuneLength=10` ¿Sobre qué valores realiza la búsqueda?

8.d) Entrena una máquina de soporte de vectores lineal haciendo que la función `train()` varíe el parámetro `C` entre los siguientes valores: 0.25, 0.5, 1, 2, 4, 8, 16, 32 y 64 ¿Con qué valor se obtiene el mejor resultado?.

5.3. Predicción

Para predecir las clases a los que pertenecen un conjunto de objetos nuevos tenemos que utilizar la función `predict()`. Por ejemplo, para para predecir las clases

para el conjunto de prueba `hepatitisTrain.sel`, utilizando el árbol anteriormente construido, bastaría con hacer:

```
> hepatitis.predict <- predict(rpartFit,newdata = hepatitisTest.sel)
```

También podemos obtener predicciones simultáneas para varios modelos. A continuación se calculan las predicciones en el conjunto de tests para la red neuronal y el árbol anteriormente generados:

```
> models <- list(C50=C50Fit,Rpart=rpartFit)
> hepatitis.predict2 <- predict(models,newdata = hepatitisTest.sel)
> hepatitis.predict2

$C50
 [1] VIVE    VIVE    FALLECE VIVE    VIVE    VIVE
 [7] VIVE    VIVE    VIVE    VIVE    VIVE    VIVE
[13] VIVE    FALLECE VIVE    VIVE    VIVE    VIVE
[19] VIVE    VIVE    VIVE    VIVE    VIVE    VIVE
[25] VIVE    VIVE    VIVE    VIVE    VIVE    VIVE
[31] VIVE    FALLECE VIVE    VIVE    VIVE    VIVE
[37] FALLECE VIVE    VIVE    VIVE    FALLECE VIVE
[43] VIVE    FALLECE VIVE    VIVE    VIVE    FALLECE
[49] VIVE    VIVE    FALLECE
Levels: FALLECE VIVE

$Rpart
 [1] VIVE    VIVE    FALLECE VIVE    VIVE    VIVE
 [7] VIVE    VIVE    VIVE    VIVE    VIVE    VIVE
[13] VIVE    FALLECE FALLECE VIVE    VIVE    VIVE
[19] VIVE    VIVE    VIVE    VIVE    VIVE    VIVE
[25] VIVE    VIVE    VIVE    VIVE    VIVE    VIVE
[31] VIVE    FALLECE VIVE    VIVE    VIVE    VIVE
[37] FALLECE VIVE    VIVE    VIVE    FALLECE VIVE
[43] VIVE    FALLECE VIVE    VIVE    VIVE    FALLECE
[49] VIVE    VIVE    FALLECE
Levels: FALLECE VIVE
```

Si queremos observar al mismo tiempo la clase asignada al objeto y la clase predicha por el modelo, debemos utilizar la función `extractPrediction()`[2]:

```
> hepatitis.predict3 <- extractPrediction(models,
                          testX = hepatitisTest.sel[,-ncol(hepatitisTest.sel)],
                          testY = hepatitisTest.sel[,ncol(hepatitisTest.sel)])
> hepatitis.predict3 <- subset(hepatitis.predict3, dataType= "Test")
> head(hepatitis.predict3)
```

```
   obs pred model dataType object
1 VIVE VIVE  C5.0 Training    C50
2 VIVE VIVE  C5.0 Training    C50
3 VIVE VIVE  C5.0 Training    C50
4 VIVE VIVE  C5.0 Training    C50
5 VIVE VIVE  C5.0 Training    C50
6 VIVE VIVE  C5.0 Training    C50
```

Ejercicio 9.

9.a) Genera predicciones de forma conjunta para los modelos que presenten mejor precisión.

9.b) Los objetos devueltos por la función pueden visualizarse a través de la función `plotObsVsPred()`. Visualiza los resultados para el objeto obtenido en el apartado anterior y explica qué significa dicha gráfica.

6. Área bajo la curva: ROC

En el caso de problemas de clasificación en los que los objetos sólo pueden pertenecer a dos clases, se suele utilizar como medida de eficiencia del clasificador el *área bajo la curva* (ROC). Si queremos que el modelo que construyamos utilice esta medida como medida de la eficacia del clasificador, deberemos indicar a través de las funciones `trainControl()` y `train()` que se va a utilizar dicha métrica.

En la función `trainControl` le indicaremos que hay que calcular las probabilidades de pertenencia a las distintas clases mediante el parámetro `classProbs=TRUE` y que, además, que cómo métrica de eficiencia del clasificador se va a utilizar la definida para problemas con dos clases, que es el ROC, por medio del parámetro `summaryFunction = twoClassSummary`.

```
> fitcontrolROC <- trainControl(method = "cv",number=10,
                                returnResamp = "final",
                                summaryFunction = twoClassSummary,
                                classProbs=TRUE)
```

A la hora de invocar la función `train()` debemos advertirle de que se va a utilizar el ROC como métrica de la eficacia del clasificador mediante el parámetro `metric = ROC"`.

```
> set.seed(342)
> svmFitROC <- train(PRONOSTICO~.,data=hepatitisTrain.sel,
                     method="svmRadial",
                     tuneLength=10,
                     preProcess = c("center", "scale"),
```

```
                    trControl=fitcontrolROC,
                    metric = "ROC")
> svmFitROC

Support Vector Machines with Radial Basis Function Kernel

104 samples
 14 predictor
  2 classes: 'FALLECE', 'VIVE'

Pre-processing: centered (14), scaled (14)
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 94, 94, 93, 93, 93, 94, ...
Resampling results across tuning parameters:

  C       ROC        Sens       Spec
    0.25  0.9520833  0.7666667  0.9125000
    0.50  0.9583333  0.7333333  0.9500000
    1.00  0.9583333  0.6500000  0.9500000
    2.00  0.9520833  0.6166667  0.9625000
    4.00  0.9333333  0.5666667  0.9513889
    8.00  0.9222222  0.5666667  0.9513889
   16.00  0.9222222  0.5666667  0.9513889
   32.00  0.9222222  0.6000000  0.9513889
   64.00  0.9222222  0.5666667  0.9513889
  128.00  0.9222222  0.6000000  0.9513889

Tuning parameter 'sigma' was held constant at a
 value of 0.05259386
ROC was used to select the optimal model using
 the largest value.
The final values used for the model were sigma
 = 0.05259386 and C = 0.5.
```

Como podemos ver, ahora la eficacia del modelo queda determinada por el índice ROC. Llegados a este punto también podemos calcular las probabilidades de la pertenencia a clases de cada ejemplo por medio de la función `predict()`:

```
> svmProb <- predict(svmFitROC,
                    newdata = hepatitisTest.sel,
                    type = "prob")
> head(svmProb)

     FALLECE      VIVE
1 0.07674600 0.9232540
2 0.02983476 0.9701652
3 0.27074603 0.7292540
4 0.04216980 0.9578302
5 0.05315712 0.9468429
6 0.04537339 0.9546266
```

El cálculo de la curva ROC lo podemos hacer utilizando el paquete `pROC`:

```
> library(pROC)
> svmROC <- roc(hepatitisTest.sel$PRONOSTICO,
                svmProb$VIVE,
                dataGrid = TRUE,
                gridLength = 100)
> svmROC

Call:
roc.default(response = hepatitisTest.sel$PRONOSTICO, predictor = svmProb$VIVE,
        dataGrid = TRUE, gridLength = 100)

Data: svmProb$VIVE in 10 controls (hepatitisTest.sel$PRONOSTICO FALLECE) < 41
cases (hepatitisTest.sel$PRONOSTICO VIVE).

Area under the curve: 0.8659
```

En la figura 6 se puede ver el resultado del comando `plot(svmROC)`.

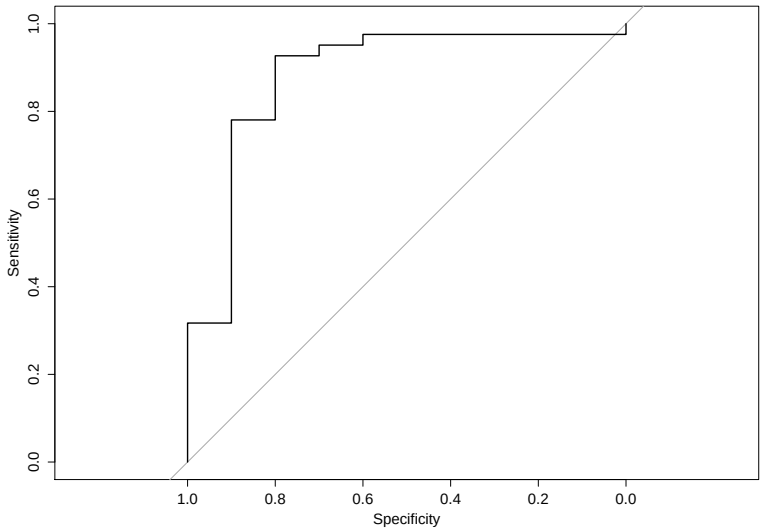

Figura 6. Curva ROC para el modelo svmFitROC

Ejercicio 10. Calcula las probabilidades de clase y la curva ROC para 2 de los tres modelos de clasificación generados en los ejercicios anteriores.

7. Evaluación de los conjuntos de prueba

Con la función `confusionMatrix()` podemos crear un objeto que contiene todas las medidas incluidas en una matriz de confusión. Podemos utilizar esta función de dos formas. En primer lugar podemos calcular la matriz de confusión a partir del objeto devuelto por la función `train()` (siempre que no se utilice para el remuestreo los métodos `boot` o `LOOCV`). En este caso se utiliza la información procedente del proceso de remuestreo:

```
> hepatitis.conf <- confusionMatrix(C50Fit)
> hepatitis.conf

Repeated Train/Test Splits Estimated (10 reps, 75%) Confusion Matrix

(entries are percentual average cell counts across resamples)

          Reference
Prediction FALLECE VIVE
   FALLECE    15.2  4.0
   VIVE        4.8 76.0

 Accuracy (average) : 0.912
```

La otra forma de calcular una matriz de confusión es la de utilizar directamente las clases predichas para el conjunto de prueba:

```
> hepatitis.pred <- predict(C50Fit,hepatitisTest.sel)
> hepatitis.conf1 <- confusionMatrix(hepatitis.pred,
                          hepatitisTest.sel[,ncol(hepatitisTest.sel)])
> hepatitis.conf1

Confusion Matrix and Statistics

          Reference
Prediction FALLECE VIVE
   FALLECE       7    1
   VIVE          3   40

               Accuracy : 0.9216
                 95% CI : (0.8112, 0.9782)
    No Information Rate : 0.8039
    P-Value [Acc > NIR] : 0.01869

                  Kappa : 0.7309
 Mcnemar's Test P-Value : 0.61708

            Sensitivity : 0.7000
            Specificity : 0.9756
```

```
        Pos Pred Value : 0.8750
        Neg Pred Value : 0.9302
            Prevalence : 0.1961
        Detection Rate : 0.1373
  Detection Prevalence : 0.1569
     Balanced Accuracy : 0.8378

      'Positive' Class : FALLECE
```

Ejercicio 11. Elige dos modelos de los anteriormente generados (ambos tienen que haber sido evaluados con la misma técnica).

11.a) Calcula la matriz de confusión y los principales índices de eficiencia de ambos modelos.

11.b) ¿Cuáles son los atributos del objeto generado por la función `confusionMatrix()`?

11.c) ¿Cómo podríamos generar una tabla con los nombres de los modelos en las filas y el de los indicadores en las columnas?

8. Comparación de diferentes modelos

El paquete `caret` también incluye funciones que nos permiten determinar las diferencias que existen entre distintos modelos, generados con la función `train()`, a través de un remuestreo de las distribuciones y obtener información estadística sobre la diferencia de efectividad de cada uno de ellos. Para poder hacer esto, primero debemos agrupar todos los resultados de remuestreo utilizando la función `resamples()`:

```
> hepatitis.resample <- resamples(models)
> summary(hepatitis.resample)

Call:
summary.resamples(object = hepatitis.resample)

Models: C50, Rpart
Number of resamples: 10

Accuracy
      Min. 1st Qu. Median  Mean 3rd Qu. Max. NA's
C50   0.88    0.88   0.92 0.912    0.92 0.96    0
Rpart 0.88    0.92   0.94 0.940    0.96 1.00    0

Kappa
        Min. 1st Qu. Median   Mean 3rd Qu.   Max. NA's
C50   0.5946  0.6087 0.7059 0.7147  0.7745 0.8837    0
Rpart 0.6512  0.7582 0.8332 0.8251  0.8837 1.0000    0
```

Existen diferentes funciones que nos permiten realizar gráficas para visualizar las distribuciones obtenidas a partir del remuestreo: `bwplots()`, para las gráficos de caja whisker, `densityplot()` para gráficos de densidad, `xyplot()` para gráficos de dispersión y `splom()` para gráficos de dispersión de resúmenes de estadísticas. La figura 7 muestra el resultado de ejecutar los comandos `bwplot(hepatitis.resample, main="bwplot")` y `densityplot(hepatitis.resample, metric = .Accuracy",auto.key=TRUE)` (figuras 8 y 8).

Dado que los modelos obtenidos han sido generados a partir de los mismos datos de entrenamiento, podemos realizar algún tipo de inferencia estadística con ellos. Por ejemplo, podemos calcular la diferencia entre modelos y después realizar la prueba T de Student para evaluar la hipótesis nula de que no hay diferencia entre los distintos modelos.

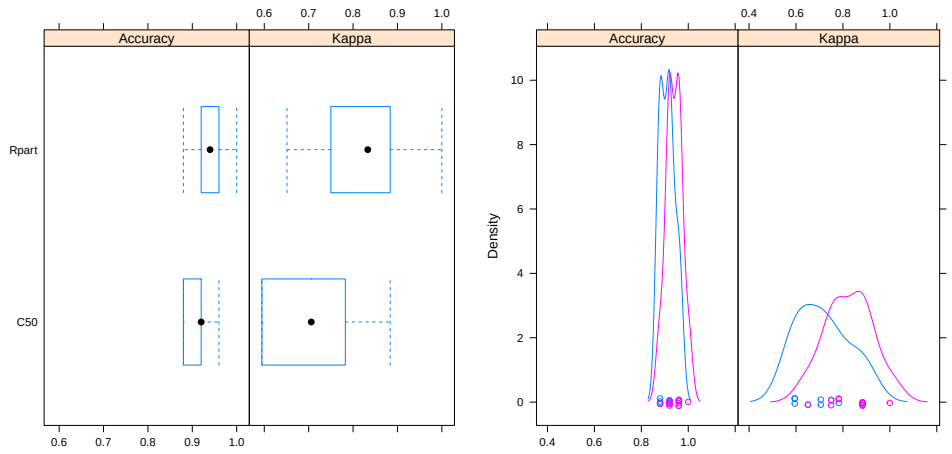

Figura 7. Gráficos utilizando las funciones `bwplot()` y `densityplot()` sobre el objeto los datos obtenidos por remuestreo.

Dado que los modelos obtenidos han sido generados a partir de los mismos datos de entrenamiento, podemos realizar algún tipo de inferencia estadística con ellos. Por ejemplo, podemos calcular la diferencia entre modelos y después realizar la prueba T de Student para evaluar la hipótesis nula de que no hay diferencia entre los distintos modelos.

```
> difValues <- diff(hepatitis.resample)
> summary(difValues)

Call:
summary.diff.resamples(object = difValues)

p-value adjustment: bonferroni
```

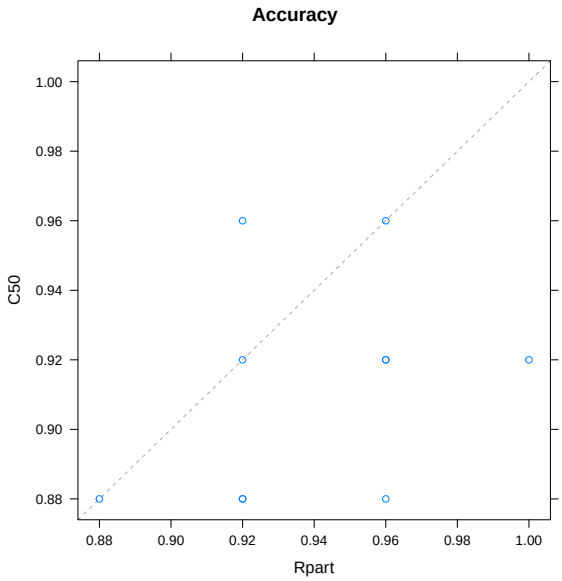

Figura 8. Gráfico utilizando la función `xyplot()` sobre el objeto los datos obtenidos por remuestreo.

```
Upper diagonal: estimates of the difference
Lower diagonal: p-value for H0: difference = 0

Accuracy
      C50    Rpart
C50          -0.028
Rpart 0.0445

Kappa
      C50     Rpart
C50           -0.1105
Rpart 0.03102
```

Los resultados sobre las diferencias entre métodos se pueden mostrar de forma gráfica a través de diferentes tipos de gráficas. La figura 9 muestra el resultado de aplicar las funciones `densityplot(difValues, metric = "Accuracy",auto.key =TRUE, pch = "|")` y `levelplot(difValues, what="diferences")` sobre las diferencias entre modelos (figura 9):

Ejercicio 12. Selecciona dos modelos de los anteriormente generados.

12.a) Extrae la información sobre el muestreo de los dos modelos y muestra un resumen con las medidas utilizadas para su evaluación.

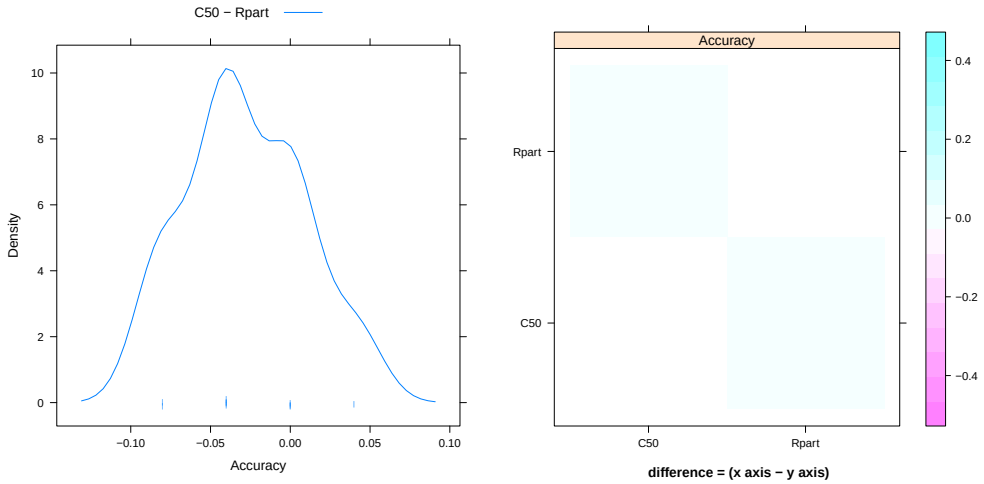

Figura 9. Gráficos utilizando las funciones `densityplot()` comparando las diferencias entre los distintos valores de la precisión y `levelplot()` del resultado del test T de Student.

12.b) Genera algunas gráficas que muestren el resultado de la operación anterior.
12.c) Aplica el test T de Student e interpreta el resultado.
12.d) La función `compare_models()` también realiza el mismo test. Aplícala y analiza los resultados.