Introducción al paquete `caret`: "classification and regression training in `R`"

Iñaki Inza, Borja Calvo

El paquete `caret` [2, 1] se está convirtiendo durante los últimos años en un referente clave para realizar tareas de clasificación y regresión en `R`. También podríamos mencionar otro paquete, `rattle`, con su útil interfaz gráfico, en muchas cuestiones similar a WEKA. `caret` por otro lado trabaja en línea de comandos, con una amplísima variedad de funciones que puedes consultar con el comando `ls("package:caret")`. Entre éstas, destacar las relacionadas con la construcción de modelos y su evaluación, así como algoritmos para selección de variables.

Esta breve introducción nos dibujará los ejes principales del software. Se mostrarán referencias para ampliar los conocimientos sobre el paquete.

Empecemos con ejemplos sobre la construcción de modelos de clasificación supervisada. Este paso final de la construcción nos lo ofrece la función `train`: sus opciones, amplísimas, las puedes encontrar en `https://www.rdocumentation.org/packages/caret/versions/6.0-84/topics/train`. Recuerda que la carga de paquetes en `R` se realiza desde la línea de comandos `install.packages("caret")`. Partiremos de la base de datos *sonar*, con 208 casos, 60 predictoras y dos valores en la variable clase. Puedes consultar en numerosos enlaces de la web acerca de la naturaleza de este dataset. Cargaremos la librería `mlbench`, que recoge numerosos datasets benchmark para tareas de aprendizaje automático. Es interesante localizar estos datasets en la propia instalación del paquete sobre el disco duro del ordenador.

Se pueden encontrar cientos de manuales, para diferentes niveles de conocimiento, para introducirse en el mundo del software `R`. Por nombrar un par de entre los más prestigiosos, Introducción de Torfs y Brauer y una serie de videotutoriales de Google Developers. Por otro lado, nuestra experiencia también nos dice que una solvente manera de aprendizaje del software es simplemente "aprender jugando y practicando".

Dependiendo los paquetes de `R` que tengamos instalados y los que se necesiten para seguir el flujo de la ejecución que os proponemos, la consola nos avisará sobre los paquetes ausentes que deberemos instalar: comando `install.packages(‘‘nombrePaquete")`.

La ayuda de cada función en `R` se obtiene mediante los comandos `?nombreFuncion` or `help(nombreFuncion)`.
Una interesante y cómoda forma de consultar en la web la ayuda de cada paquete y sus funciones, así como de sus parámetros, es el portal `https://www.rdocumentation.org/`. En nuestro caso, las funciones del paquete `caret` se pueden consultar en `https://www.rdocumentation.org/packages/caret/`.
Consejo. Buscando en la web el nombre de la función de interés, seguida del término `R`, nos ofrece los enlaces clave para consultar la ayuda que posiblemente necesitamos.

Año: Estudios de Master
Profesor(a): Iñaki Inza, Borja Calvo
Teléfono: 943 01 50 26
E-mail: inaki.inza,borja.calvo@ehu.es

GIPUZKOAKO CAMPUSA
CAMPUS DE GUIPUZCOA
Paseo Manuel de Lardizabal, 1
20018 Donostia (Gipuzkoa)

Antes de aprender un modelo de clasificación es necesario definir las instancias de entrenamiento y test que darán forma a nuestro modelado. Fijamos una semilla para futuras generaciones de numeros aleatorios. Hacemos uso de la función `createDataPartition` para generar una partición train-test de las 208 instancias, que mantendremos durante todo el pipeline de análisis. Consulta sus parámetros y compréndelos en el contexto de nuestra llamada. Consulta otras dos funciones hermanas de ésta, `createFolds` y `createResample` y entiende las diferencias entre las tres: claro, podríamos validar el modelo utilizando un modelo de 'cross-validation'. A partir de la ristra de integer (índices) generada, procedemos a particionar nuestra matriz original en dos objetos.

```
library(caret)
library(mlbench)
data(Sonar)
names(Sonar)
set.seed(107)
inTrain <- createDataPartition(y=Sonar$Class,p=.75,list=FALSE)
str(inTrain)
training <- Sonar[inTrain,]
testing <- Sonar[-inTrain,]
nrow(training)
```

Ya que tenemos los datos para el aprendizaje (y testado) del modelo en la línea de salida, vamos a ello. Aprendemos todo un clásico, un modelo de análisis discriminante lineal (LDA). En la misma llamada al entrenamiento del modelo hay que resaltar otro parámetro clave en cualquier tarea de análisis de datos: los filtros de preproceso por los pasarán las predictoras previo al aprendizaje (parámetro `preProc`. Nos ofrece un conjunto de filtros realmente interesantes: imputación de valores perdidos, escalado, PCA... A continuación un listado de éstos y su definición. En este caso filtramos mediante un centrado y escalado de las variables.

Estudiando el output del modelo, puedes comprobar que la llamada `train` de `caret` siempre estima un porcentaje de bien clasificados sobre la partición inicial de entrenamiento que ya hemos fijado (objeto `training` en nuestro caso). Por defecto, esta estimación se realiza mediante la técnica de *bootstrap*. La función `trainControl` controla el tipo de estimación del error: realizaremos una validación cruzada de (por defecto) 10 hojas ("folds"), repitiéndola 3 veces. El parámetro `method` de la función `trainControl` hace posible utilizar distintos tipos de validación: puedes consultarlos en la ayuda de la función `train` expuesto previamente (`?train`). Aparte del tipo de validación, la función `trainControl` permite fijar multitud de parámetros del proceso de validación. Aún así, recuerda que el modelo sigue siendo el mismo, aprendido sobre la partición de entrenamiento: pero esta partición se ha resampleado de formas diferentes para estimar el error del modelo. Observarás que junto con la estimación del porcentaje de bien clasificados ("accuracy"), se nos muestra el valor del estadístico *Kappa*, el cual es una medida que compara el "accuracy" observado respecto al "accuracy" esperado (de un predictor al azar). Esto es, cuánto mejor estimamos que lo hace nuestro clasificador respecto a otro que predijese al azar la clase (siguiendo la probabilidad a priori de éstas). Un ameno ejemplo para entenderla mejor se encuentra en aquí.

```
ldaModel <- train (Class ~ ., data=training,method="lda",preProc=c("center","scale"))
ldaModel
ctrl <- trainControl(method = "repeatedcv",repeats=3)
ldaModel3x10cv <- train (Class ~ ., data=training,method="lda",trControl=ctrl,
                         preProc=c("center","scale"))
ldaModel3x10cv
```

> El formato de expresión `Class ~` es todo un clásico en `R` para denotar primeramente la variable que se quiere predecir, y después del símbolo `~`, representando explícitamente el subconjunto de variables predictoras, o bien mediante un punto indicando que el resto de variables del dataset son predictoras.

La llamada al proceso de entrenamiento del clasificador se puede seguir enriqueciendo con argumentos adicionales a la función `trainControl`. Uno de ellos es `summaryFunction`, referente a las medidas de evaluación del clasificador. Así, activando su opción `twoClassSummary` en un escenario de valores binarios a predecir, obtendremos medidas como el área bajo la curva ROC, sensibilidad y especificidad. Para ello y ya que no se computan de manera automática, también hay que activar la opción `classProbs` para que se tengan en cuenta, en cada instancia, las probabilidades predecidas para cada valor de la variable clase. Estudia el output de la llamada.

```
ctrl <- trainControl(method = "repeatedcv",repeats=3, classProbs=TRUE,
                     summaryFunction=twoClassSummary)
ldaModel3x10cv <- train (Class ~ ., data=training,method="lda",trControl=ctrl,metric="ROC",
                         preProc=c("center","scale"))
ldaModel3x10cv
```

Hasta ahora hemos trabajado con un análisis discriminante lineal, un clasificador que no tiene parámetros extra para ser construido. En la amplia lista de modelos que podemos aprender en `caret`, la gran mayoría tienen parámetros que se pueden adaptar para el entrenamiento. Puedes consultarlos, tanto los clasificadores, agrupados por familias, como sus respectivos parámetros, en este enlace. Así, para clasificadores con al menos un parámetro para su aprendizaje, `caret` realiza las mismas evaluaciones que hemos visto hasta ahora para (por defecto) 3 valores de cada parámetro. Para ver el efecto de lo expuesto en el output de aprendizaje-evaluación, escogeremos, por ejemplo, el clasificador *partial least squares discriminant analysis (PLSDA)*, hermanado con el LDA utilizado hasta ahora. Observa que para su proceso de entrenamiento hay un único parámetro a tunear. Por otro lado, date cuenta que mantenemos los mismos parámetros de control del entrenamiento utilizados hasta ahora, `ctrl`: ésta es una gran ventaja de `caret`. Mediante el parámetro `tuneLength` podemos ampliar el número de valor por parámetro a considerar en el entrenamiento. Estudia los distintos outputs. Sencillos plots para ambos modelados (3 o 15 valores por parámetro): fíjate en sus ejes horizontales y verticales.

```
plsFit3x10cv <- train (Class ~ ., data=training,method="pls",trControl=ctrl,metric="ROC",
                       preProc=c("center","scale"))
plsFit3x10cv
plot(plsFit3x10cv)
plsFit3x10cv <- train (Class ~ ., data=training,method="pls",trControl=ctrl,metric="ROC",
                      tuneLength=15,preProc=c("center","scale"))
plsFit3x10cv
plot(plsFit3x10cv)
```

Para predecir la clase de casos futuros, `caret` tendrá en cuenta el clasificador con el mejor valor de sus parámetros. Consulta los parámetros de la llamada a la función `predict` en este enlace: entre éstos, la opción `type` merece ser estudiada. Con la opción `probs` se calcula, por caso de test, la probabilidad a-posteriori para cada valor de la clase; con la opción `raw` nos quedamos, por caso de test, con el valor de la variable clase con mayor probabilidad a-posteriori. A partir de esta segunda opción podemos calcular la matriz de confusión y la colección de estadísticos de evaluación asociados; al fin y al cabo esto supone "cruzar", por caso de test, los valores clase predichos con los valores clase reales.

```
plsProbs <- predict(plsFit3x10cv, newdata = testing, type = "prob")
plsClasses <- predict(plsFit3x10cv, newdata = testing, type = "raw")
confusionMatrix(data=plsClasses,testing$Class)
```

¿Podemos realizar una comparativa en base a los resultados de evaluación interna de 3x10cv entre LDA y PLSDA? Ya que no hemos cambiado la semilla de aleatorización, las particiones (de casos de training) utilizadas en los procesos de "resampling"-validación de cada clasificador han sido las mismas: esto es, las "folds"-hojas del proceso de validación cruzada han sido las mismas en ambos clasificadores. Primero, un resumen cruzando ambos resultados. Podemos plotearlos de muchas maneras: una, con un "Bland-Altman" plot, tan intuitivo, reflejando

las resta del valor de la métrica ROC entre ambos modelos en cada una de las 3x10 hojas de la validación cruzada (ver Figura 1. Al compartir las "folds"-hojas, un t-test pareado calculará la significancia de las diferencias entre los estimadores de error de ambos modelo. Aunque parezcan a primera vista una sopa de números sin sentido, no es así. El output de la comparativa es muy rico y nos da mucha información. Así, contiene, para cada medida (área bajo la curva ROC, sensibilidad y especificidad), la diferencia de medias (positiva o negativa) entre clasificadores (siguiendo el orden de la llamada a la función `resamps`); y el p-value asociado para interpretar el nivel de significatividad de las diferencias entre los modelos. Se trata de chequear las diferencias, para cada score, entre el par de clasificadores comparado. Y de interpretar, mediante el p-value asociado, si estas diferencias son estadísticamente significativas o no, `https://en.wikipedia.org/wiki/Statistical_significance`, usando el clásico umbral de 0.05 o 0.10.

```
resamps=resamples(list(pls=plsFit3x10cv,lda=ldaModel3x10cv))
summary(resamps)
xyplot(resamps,what="BlandAltman")
diffs<-diff(resamps)
summary(diffs)
```

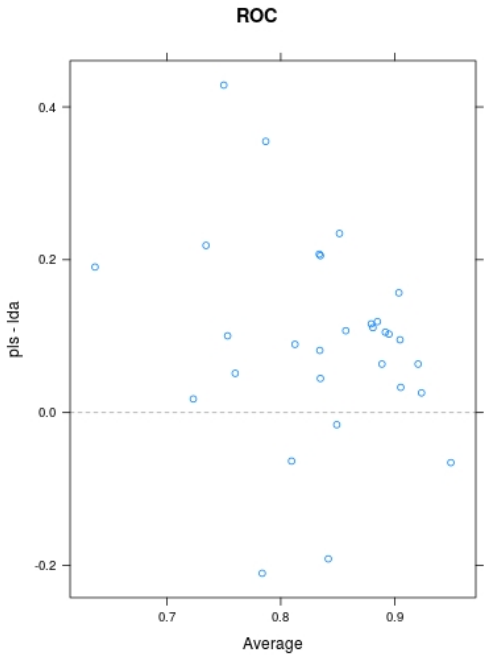

Figura 1: Comparativa PLS versus LDA. "Bland-Altman" plot

De los más de 150 modelos de clasificación soportados por `caret`, una pequeña parte de ellos se aprenden sobre software propio de este paquete. Un gran mérito de `caret` es que ofrece un útil y transparente acceso a funciones de otros paquetes que aprenden la amplia lista de clasificadores expuestos. Por ello, es posible que, dependiendo del tipo de clasificador que quieras construir, se te pida instalar otros paquetes. Otro popular paquete en `R` con el mismo fin de aglutinar algoritmos y procedimientos de aprendizaje automático es `mlr`.

Ejercicio propuesto

kaggle.com es un popular repositorio donde empresas e instituciones vuelcan un problema de análisis de datos para que grupos de "data miners" de todo el mundo aprendan sobre ellos sus modelos y "compitan" para proponer el mejor modelo. Ciñéndonos a un formato de clasificación supervisada, el formato habitual consiste en que el propietario de los datos haga disponible un conjunto de datos etiquetado sobre el que los competidores deben aprender sus modelos; mientras que por otro lado el propietario también hace disponible un segundo set de datos que los evaluadores deben etiquetar-categorizar con sus modelos aprendidos. Así, ya que el propietario dispone del etiquetado de este segundo set de datos, puede evaluar a los distintos competidores. El formato habitual de las matrices de datos es `csv`, fácilmente legible en `R` mediante el popular comando `read.csv()`.

No todas las competiciones propuestas se ciñen a la clasificación supervisada. Hay también sistemas de recomendación, clustering, visualización, etc. El ejercicio consiste en que primeramente selecciones una competición concreta en la que subyazca un problema de clasificación supervisada (puedes ver el listado de todas las competiciones, tanto las que están en curso como las finalizadas). Puede tratarse de una competición 'active' o 'completed'. Escoge una competición que que te motive, cercana a tus intereses, intenta que no sea Titanic ni ningún 'toy dataset' similar... Y en **un documento-notebook similar al aquí expuesto, intercalando en éste explicaciones sobre tus decisiones y el código que las implementa**, describas tu experiencia con la base de datos y "pipeline" de análisis supervisado que has implementado: esto es, coloquialmente, "haz tu tutorial". Este documento que tienes delante se ha editado con el paquete `knitr` y el software `RStudio`: formato `Rnw`). Aúna código y LaTex, aunque ya ves que no aprovecho para escribir las elegantes fórmulas de LaTex. Si te animas a editarlo con la misma tecnología, perfecto: creo que es elegante y es popular hoy día. Pero puedes escoger otra manera de editarlo, por supuesto: ya sabes que la variedad de formatos de notebook es amplia.

Esperamos que en tu documentación, en tu 'notebook' a entregar, **desarrolles y documentes al menos los siguientes puntos**:

- breve **descripción** acerca del dataset escogido: variable clase a predecir y sus posibles valores, variables predictoras, correlaciones entre predictoras, objetivo general del modelo, métrica de evaluación, naturaleza general del problema...
- relacionado con el punto anterior, **gráficas de visualización** de algunas variables de interés: histogramas, densidad, etc. Visualizaciones-matriciales de correlaciones entre pares de predictoras (i.e. búsqueda de redundancias). Hay muchos paquetes para ello en `R`: algunos pueden ser `ggplot2`, `skimr`, `lattice`;
- la forma de partición inicial train-test no tiene porqué ser la misma que la del tutorial. Plantéate el uso de otras funciones como `createFolds(), creteResample()`. Entiéndelas;
- teniendo en cuenta las características de los datos, en este tutorial se han utilizado dos opciones de **preproceso** concretas. Las características de tus datos pueden exigir la aplicación de las mismas, o bien otras. Chequea las **opciones del parámetro** `preProc` en el entrenamiento del modelo. Si tu dataset tiene valores perdidos, encontrarás en la web múltiples tutoriales para imputarlos. `caret` nos ofrece esta atractiva ayuda-web sobre las posibilidades de preproceso. O la ayuda en rdocumentation;
- escoge otros **algoritmos de clasificación supervisada** distintos a los del tutorial. Para los que escogas, describe brevemente su funcionamiento y describe sus parámetros. Explica el efecto del parámetro `tuneLength` de `caret` en el proceso de aprendizaje del modelo y **tunning** de parámetros: es clave para que entiendas el posterior output de los resultados. Ten en cuenta que si el tipo de clasificador tiene más de un parámetro a tunear, se chequean-tunean *combinaciones* de valores de parámetros. Mientras que con el parámetro `tuneLength` es `caret` quien elige los valores de los parámetros a tunear, con `tuneGrid` es el usuario el que fija los valores a evaluar en el proceso de tunning. Realizar pruebas tanto con `tuneLength` como con `tuneGrid`;
- el parámetro `trainControl` nos permite fijar las opciones de validación del clasificador. Chequea en la ayuda de `R` sus opciones;

- la **métrica de evaluación** de los modelos en el tutorial ha sido el área bajo la curva ROC. `caret` nos ofrece un amplio abánico de métricas: `Accuracy, kappa, Sens, Spec`, `RMSE` en el caso de problemas de regresión... no he encontrado en la web un listado con todas las métricas, pero podrás encontrar información interesante en la propia web de `caret`, así como en foros de preguntas-respuestas;

- en caso de que tu problema sufra un notable **desbalanceo de clases**, prueba algún método de 'resampling' para los casos minoritarios en momento de entrenamiento del clasificador. Siempre muestra en alguna sencilla visualización la distribución de los valores de la variable clase. Fíjate en las opciones del parámetro `sampling` de `trainControl`: enlace. Hay muchas formas de encontrar ayuda en la red sobre ésto. Una ya nos la ofrece `caret`. En las primeras frases de este enlace se resumen brevemente las opciones: su uso lo tienes descrito más adelante, en este enlace;

- indaga en las técnicas de **selección de variables** que nos ofrece `caret`. Aplica alguna de ellas en tu dataset. Aquí se dan la mano el mundo de los heurísticos de búsqueda (optimización combinatoria), y el análisis de datos. Voy observando que el paquete `caret` sigue ampliando su abanico de heurísticos para el denominado 'feature selection', esto es, para optimizar el subconjunto de variables escogidas en nuestro problema de clasificación: mediante algoritmos genéticos, 'simulated annealing', 'recursive-backward elimination'. O filtros univariados que rankeen las variables según su relevancia-correlación respecto a la clase-output a predecir;

- el punto previo ha sido de 'feature selection'. Pisemos ahora el campo del **feature extraction**: construyendo variables, a partir de las originales, habitualmente combinaciones lineales de estas últimas. En este área, es muy posible que conozcas todo un clásico, el principal components analysis (PCA). Es muy sencillo realizarlo en `R` mediante la función `prcomp`, y visualizar las dos primeras componentes (i.e. las que recogen mayor variabilidad de los datos originales), en busca de una intruitiva separabilidad de las clases del problema. Hazlo;

- conocer los parámetros de la función `predict()` es indispensable para que domines las opciones que tienes a la hora de predecir la clase de los casos de la partición de test. No te limites a la forma en la que he utilizado la función en el tutorial: sus posibilidades van bastante más allá;

- sigue el mismo proceder con la **comparativa estadística final entre dos modelos**, sus posibilidades son más amplias que las pocas líneas previas. Comprende el uso de las funciones `resamples(), summary(), diff()` y el output final: cada número tiene un mensaje interesante. Extrae conclusiones de la comparativa;

Bibliografía

[1] Max Kuhn. Contributions from Jed Wing, Steve Weston, Andre Williams, Chris Keefer, Allan Engelhardt, Tony Cooper, Zachary Mayer, and the R Core Team. *caret: Classification and Regression Training*, 2014. R package version 6.0-35.

[2] M. Kuhn and K. Johnson. *Applied Predictive Modeling*. Springer, 2013.