

Métodos Supervisados - caret - Forest Cover Type

Gustavo Quevedo

19/06/2020

Descripción del problema

El **objetivo** de esta competición es clasificar un conjunto de instancias según su **tipo de cubierta forestal**, es decir, el tipo de árbol mayoritario del bosque, entre un conjunto de 7 tipos diferentes. Este conjunto de instancias incluye un conjunto de valores obtenidos a partir de una parcela de 30 x 30 metros de cada bosque, proporcionado por el USFS (Servicio Forestal de los Estados Unidos).

La **competición en Kaggle** puede encontrarse en el siguiente enlace:

<https://www.kaggle.com/c/forest-cover-type-kernels-only>

Los diferentes **tipos de cubierta forestal** podrán ser:

- 1 - Spruce/Fir (Abeto)
- 2 - Lodgepole Pine (Pino contorta)
- 3 - Ponderosa Pine (Pino ponderosa)
- 4 - Cottonwood/Willow (Álamo de Virginia/Sauce)
- 5 - Aspen (Álamo temblón)
- 6 - Douglas-fir (Abeto Douglas)
- 7 - Krummholz (Madera de rodilla)

Las **variables predictoras** incluyen **características de la parcela** como: pendiente, elevación, nivel de sombra, tipo de suelo/tierra o distancia a carreteras, hidrológica (a río, océano, lago, etc) o a puntos de potenciales incendios.

El conjunto de **entrenamiento** comprende más de **15 mil observaciones**. Nos encontramos ante un problema de **clasificación multiclas** y la **métrica** que kaggle emplea para evaluar los modelos es la **accuracy** (precisión).

Inicialización de datos

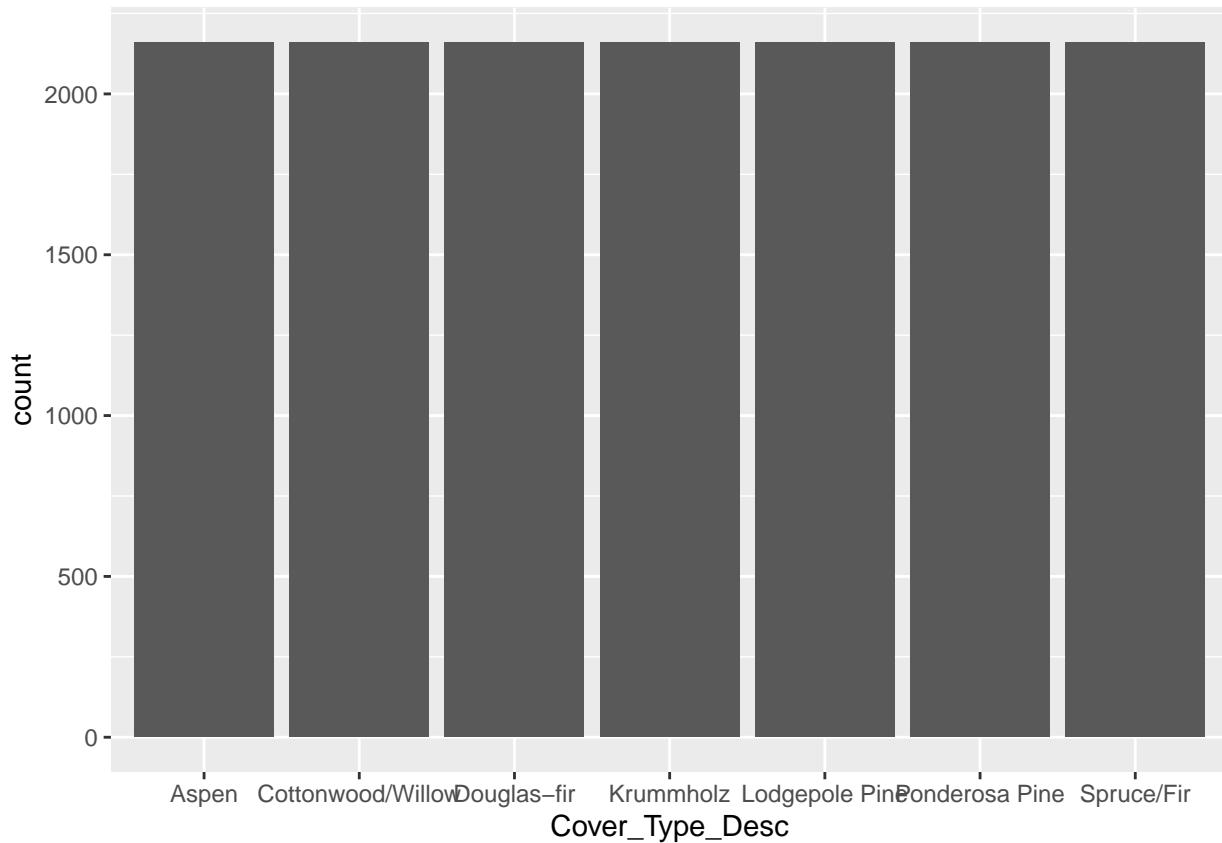
Lo primero, crearemos una nueva variable clase del tipo factor, cuyos valores serán los nombres de las cubiertas forestales, en lugar de los índices del 1 al 7 que son poco clarificadores.

```
data$Cover_Type_Desc <- ""  
data[which(data$Cover_Type == 1),]$Cover_Type_Desc <- "Spruce/Fir"  
data[which(data$Cover_Type == 2),]$Cover_Type_Desc <- "Lodgepole Pine"  
data[which(data$Cover_Type == 3),]$Cover_Type_Desc <- "Ponderosa Pine"  
data[which(data$Cover_Type == 4),]$Cover_Type_Desc <- "Cottonwood/Willow"  
data[which(data$Cover_Type == 5),]$Cover_Type_Desc <- "Aspen"  
data[which(data$Cover_Type == 6),]$Cover_Type_Desc <- "Douglas-fir"  
data[which(data$Cover_Type == 7),]$Cover_Type_Desc <- "Krummholz"
```

```
data$Cover_Type_Desc <- as.factor(data$Cover_Type_Desc)
data$Cover_Type <- NULL
```

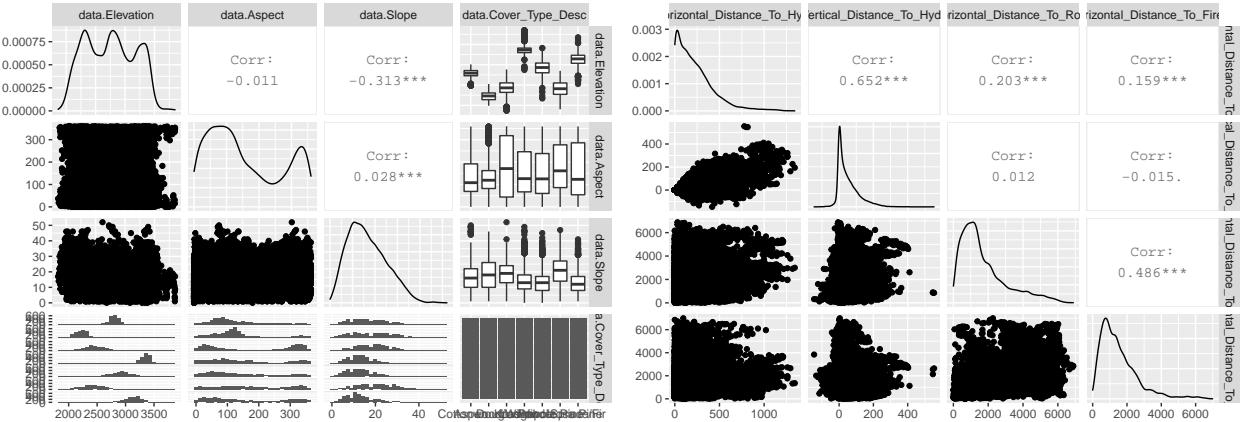
En el gráfico siguiente comprobamos que las clases están totalmente balanceadas en el *dataset* recibido por lo que no habrá que aplicar *resampling* para casos minoritarios.

```
ggplot(data, aes(Cover_Type_Desc)) + geom_bar()
```



Correlación

Visualizaremos la correlación entre algunas de las variables:



Vemos que podemos obtener cierta información de las gráficas anteriores y que no se observa una alta correlación entre las variables.

Se ha observado que los conjuntos de variables predictoras **Wilderness_Area[1-4]**(area silvestre) y **Soil_Type[1-40]** (tipo de suelo/tierra) son en realidad variables categóricas que han sido *dummificadas*.

Esto dificultaba su comprensión, pero una vez comprobado lo podemos dejar así ya que será útil en el modelado. Por tanto, no habrá necesidad de utilizar la función `dummy_cols` de la biblioteca **fastDummies**.

De entre ellas, comprobamos que hay dos variables cuyo valor siempre es 0, por lo que no aportan información por lo que procedemos a eliminarlas.

```
str(factor(data$Soil_Type7))
```

```
##  Factor w/ 1 level "0": 1 1 1 1 1 1 1 1 1 1 ...
```

```
data$Soil_Type7 <- NULL
submit_data$Soil_Type7 <- NULL

str(factor(data$Soil_Type15))
```

```
##  Factor w/ 1 level "0": 1 1 1 1 1 1 1 1 1 1 ...
```

```
data$Soil_Type15 <- NULL
submit_data$Soil_Type15 <- NULL
```

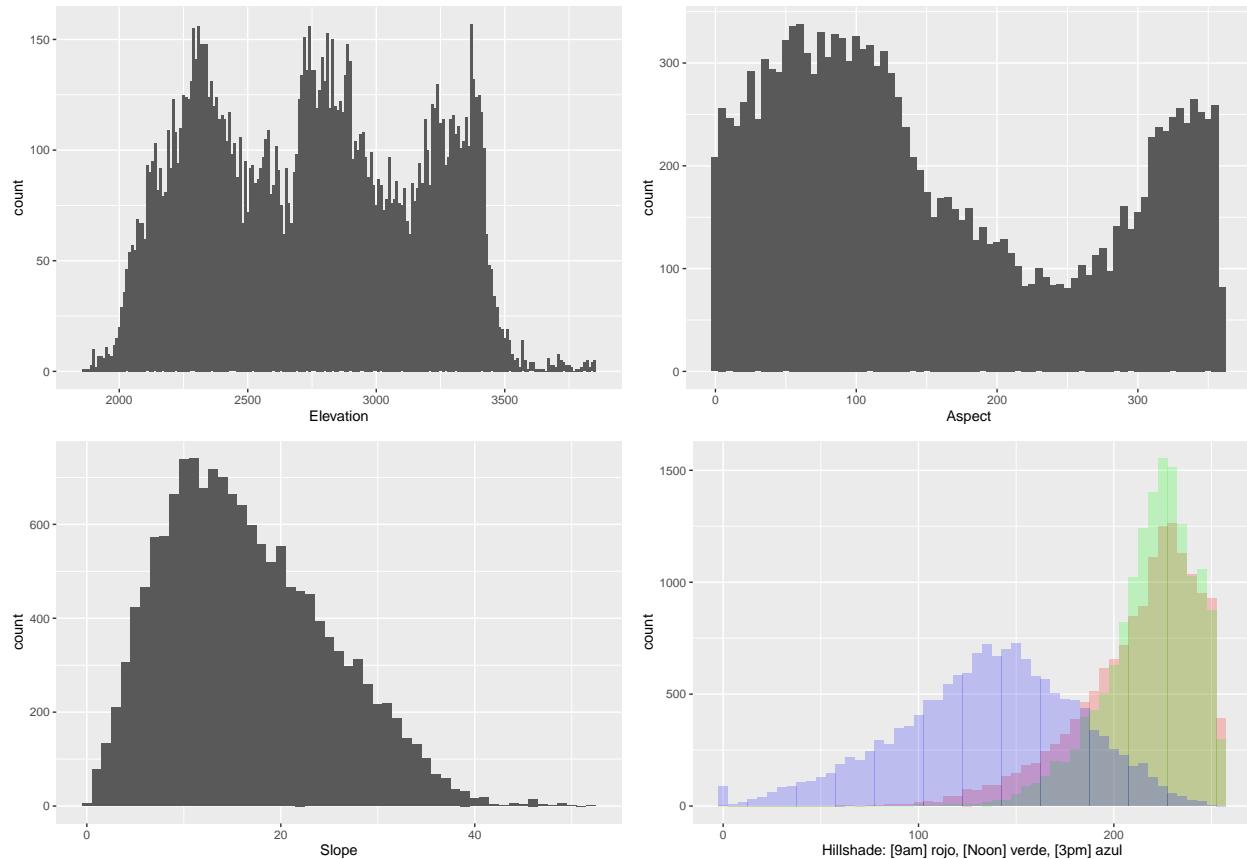
Por otro lado, tenemos la variable **Id**, útil para enviar nuestras predicciones a *Kaggle* pero que no tendrá ninguna relevancia para el modelado. Por tanto la eliminaremos del *dataset*, guardando una referencia para un futuro uso.

```
data.Id <- data@Id
submit_data.Id <- submit_data@Id

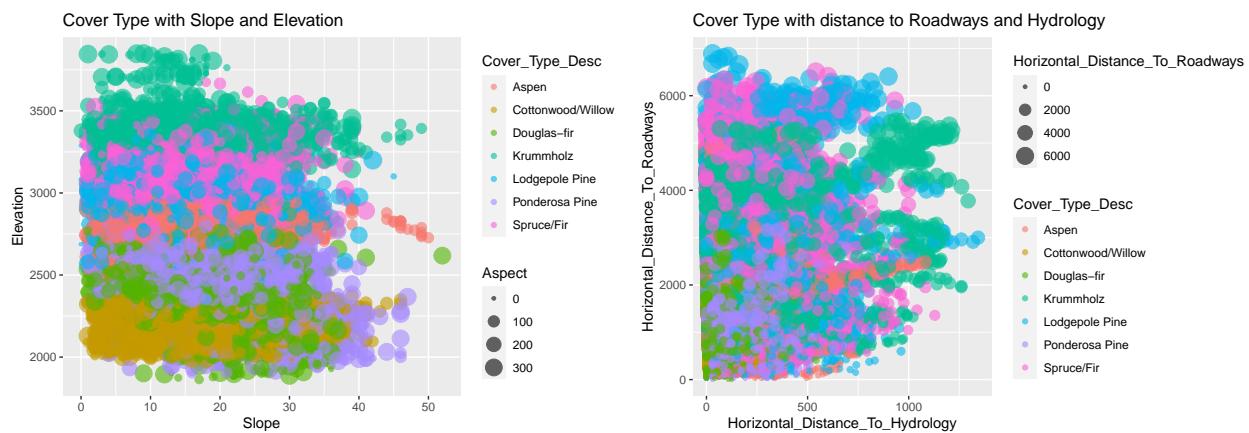
data@Id <- NULL
submit_data@Id <- NULL
```

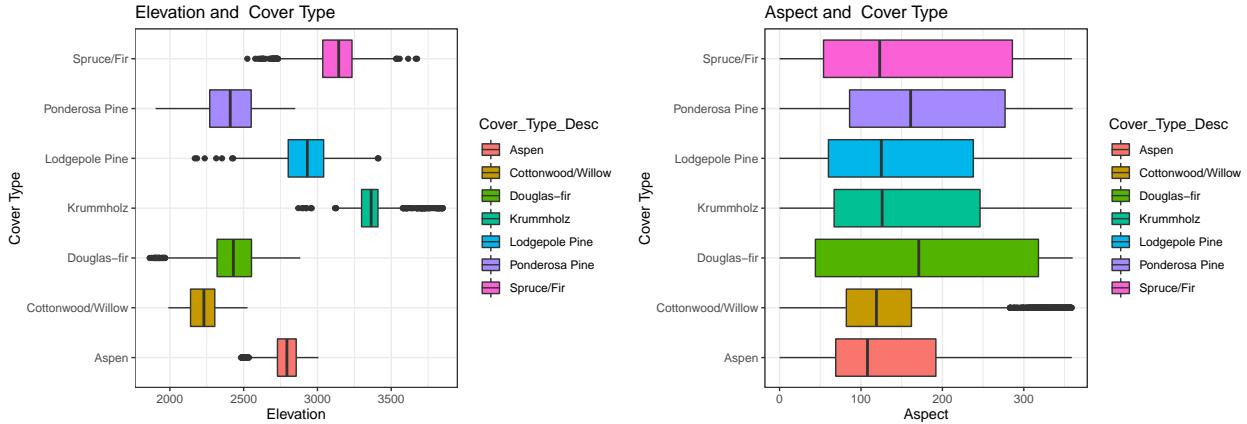
Visualización

A continuación se muestran algunos gráficos que muestran la distribución de los valores de diferentes variables:



Las siguientes gráficas muestran la distribución de la variable clase en diferentes colores según algunas de las variables predictoras:





Referencias de esta sección

- <https://www.kaggle.com/arjundas/forest-run>
- <https://www.kaggle.com/ambarish/forest-cover-type-eda-and-modelling>

Partición de los datos

Repasaremos algunas de las funciones que ofrece *caret* utilizadas para la partición de datos:

createResample es útil para realizar bootstrapping, con muestras del mismo tamaño que la original, donde algunas instancias aparecerán repetidas y otras no aparecerán

```
library(caret)
samples <- createResample(y=data$Cover_Type_Desc, 10)
```

El resultado podría ahora, por ejemplo, pasarse a la función **trainControl** como índice, utilizando “**boot**” como *method*. De este modo establecemos el remuestreo deseado:

```
ctrl <- trainControl(
  method="boot",
  number=10,
  savePredictions="final",
  classProbs=TRUE,
  index=samples
)
```

Mediante **createFolds** podemos dividir la muestra en grupos o pliegues (sin repetir elementos) para posteriormente hacer validación cruzada.

```
folds <- createFolds(y=data$Cover_Type_Desc, k = 10)
```

El resultado es un vector con el grupo al que pertenece cada instancia. Esto lo podemos utilizar del siguiente modo

```
cvMetodo <- lapply(folds, function(x){
  training_fold <- data[-x, ]
  test_fold <- data[x, ]
  #modelado
})
```

Finalmente, decidimos utilizar la función **createDataPartition** para realizar la separación de datos por su simplicidad. Vamos a dividir el conjunto de datos que se nos proporcionan con valores en la variable clase (training), a su vez en dos conjuntos de entrenamiento y test para poder entrenar y evaluar nuestro modelo.

```
inTrain <- createDataPartition(y=data$Cover_Type, p=.75, list=FALSE)
train <- data[inTrain,]
test <- data[-inTrain,]
```

Preprocesado

Comprobamos que no existen valores perdidos por lo que no será necesario eliminar instancias o imputarlos ningún otro valor dentro del preprocesado.

```
sum(is.na(data))
```

```
## [1] 0
```

A continuación se resumen las opciones de preproceso disponibles en la función **preProcess()**

- “**BoxCox**”: aplica una transformación Box-Cox, los valores deben ser distintos de cero y positivos.
- “**YeoJohnson**”: aplica una transformación Yeo-Johnson, como un BoxCox, pero los valores pueden ser negativos.
- “**expoTrans**”: aplica una transformación de potencia como BoxCox y YeoJohnson.
- “**zv**”: elimina los atributos con una varianza cero (todo el mismo valor).
- “**nzv**”: elimina los atributos con una variación cercana a cero (cerca del mismo valor).
- “**center**”: resta la media de los valores.
- “**scale**”: divide los valores por desviación estándar.
- “**range**”: normalizar valores.
- “**pca**”: transforma los datos en los componentes principales.
- “**ica**”: transforma los datos en componentes independientes.
- “**spatialSign**”: datos del proyecto en un círculo unitario.

El *dataset* no cuenta con valores perdidos, y no se han identificado situaciones por las cuales sea adecuado usar opciones adicionales al centrado y escalado de los datos. Finalmente se opta por incluir las mismas opciones de preprocesado que en el tutorial (**center** y **scale**), que por otro lado son las más utilizadas.

```
preProc <- preProcess(data, method = c("center", "scale"))
```

Referencias de esta sección

- <https://machinelearningmastery.com/pre-process-your-dataset-in-r/>

Función *trainControl()*

Resumiremos los métodos de validación más empleados en la función **trainControl()**:

- **Bootstrap** (method=“boot”): El remuestreo de Bootstrap implica tomar muestras aleatorias del conjunto de datos (con reemplazo) para evaluar el modelo. En conjunto, los resultados proporcionan una indicación de la varianza del rendimiento de los modelos.

- **k-fold Cross Validation** (method="cv"): La validación cruzada implica dividir el conjunto de datos en k pliegues o *folds*. Un pliegue utiliza como conjunto de test mientras el modelo se entrena con los demás subconjuntos.
- **Repeated k-fold Cross Validation** (method="repeatedcv"): Se trata de validación cruzada con repetición, de modo que, en función del número de repeticiones, todos pliegues podrían ser utilizados como conjunto de test y el *score* resultante será la media del *score* de todas las iteraciones.
- **Leave One Out Cross Validation** (method="LOOCV"): La validación cruzada dejando uno fuera implica separar los datos de forma que para cada iteración tengamos una sola muestra para los datos de prueba y todo el resto conformando los datos de entrenamiento.

El parametro **number** indica el número de pliegues o iteraciones de resampling (en función del método).

Para el método **repeated-cv** se utiliza el parámetro **repeats** para indicar el número de repeticiones a realizar en la validación cruzada.

Crearemos una semilla y utilizaremos el método **repeated-cv** con un número relativamente bajo de pliegues y repeticiones para evitar que el tiempo de procesamiento aumente en exceso:

```
set.seed(1715)
control <- trainControl(method="repeatedcv", number=3, repeats=3)
```

Referencias de esta sección

- <https://machinelearningmastery.com/how-to-estimate-model-accuracy-in-r-using-the-caret-package/>

Algoritmos de clasificación supervisada

Para este proyecto hemos decidido utilizar dos de los métodos de clasificación supervisados más populares hoy en día: **XGBoost** y **Random Forest**.

Utilizaremos primero valores personalizados para los parámetros, mediante **tuneGrid** y a continuación valores aleatorios para el tuning mediante **tuneLength**.

tuneGrid permite asignar valores concretos para los parámetros que se utilizarán en el modelo y **tuneLength** asignará un número de niveles para cada parámetro de tuning utilizando valores aleatorios.

El valor del parámetro **metric** enviado a la función **train()** será *Accuracy*, ya que ésta es la medida que se utilizará en *Kaggle* para evaluar los modelos.

XGBoost

XGBoost Extreme Gradient Boosting es un algoritmo predictivo supervisado que utiliza el principio de boosting.

La idea detrás del boosting es generar múltiples modelos de predicción “débiles” secuencialmente, y que cada uno de estos tome los resultados del modelo anterior, para generar un modelo más “fuerte”, con mejor poder predictivo y mayor estabilidad en sus resultados.

Durante el entrenamiento, los parámetros de cada modelo débil son ajustados iterativamente tratando de encontrar el mínimo de una función objetivo.

Este algoritmo se caracteriza por obtener buenos resultados de predicción con relativamente poco esfuerzo, en muchos casos equiparables o mejores que los devueltos por modelos más complejos computacionalmente, en particular para problemas con datos heterogéneos.

Parámetros La Lista completa de parámetros puede encontrarse aquí: <https://github.com/dmlc/xgboost/blob/master/doc/parameter.rst>

Los Parámetros utilizados:

- **nrounds**: El número de rondas para ejecutar la capacitación (único parámetro obligatorio en cualquier caso)
- **max_depth**: Profundidad máxima de un árbol. El aumento de este valor hace que el modelo sea más complejo y que se sobreajuste con más probabilidad. 0 indica que no hay límite.
- **eta**: Contracción del tamaño del paso utilizado en las actualizaciones para evitar el ajuste excesivo.
- **gamma**: La reducción de pérdida mínima necesaria para realizar una partición mayor en un nodo de hoja del árbol. Conforme mayor sea, más conservador será el algoritmo.
- **colsample_bytree**: Proporción de la submuestra de columnas cuando se construye cada árbol.
- **min_child_weight**: Suma mínima de la ponderación de instancias (hessiana) necesaria en un elemento secundario. Si el paso de partición del árbol genera un nodo de hoja con la suma de la ponderación de instancia inferior a **min_child_weight**, el proceso de creación deja de realizar la partición.
- **subsample**: La proporción de la submuestra de la instancia de capacitación. Si se establece en 0,5, significa que XGBoost recopila de forma aleatoria la mitad de las instancias de datos para que los árboles crezcan. Esto evita el sobreajuste.

```
library(xgboost)
xgb_tune_grid_1 <- expand.grid(nrounds = 50,
                                 max_depth = 10,
                                 eta = 0.3,
                                 gamma = 0.01,
                                 colsample_bytree = 0.75,
                                 min_child_weight = 1,
                                 subsample = 0.8)

xgb_tune_grid_2 <- expand.grid(nrounds = 50,
                                 max_depth = 6,
                                 subsample=0.85,
                                 colsample_bytree=0.7,
                                 min_child_weight = 2,
                                 eta = 0.1,
                                 gamma = 0.01)

xgb_custom_1 <- train(Cover_Type_Desc ~.,
                       data = train,
                       method = "xgbTree",
                       metric='Accuracy',
                       trControl=control,
                       tuneGrid = xgb_tune_grid_1)

xgb_custom_2 <- train(Cover_Type_Desc ~.,
                       data = train,
                       method = "xgbTree",
                       metric='Accuracy',
                       trControl=control,
                       tuneGrid = xgb_tune_grid_2)

xgb_random <- train(Cover_Type_Desc ~.,
                      data = train,
```

```
    method = "xgbTree",
    metric='Accuracy',
    trControl=control,
    tuneLength = 2)
```

A continuación mostraremos la medida de *Accuracy* para los dos modelos creados con variables personalizadas

```
xgb_custom_1$results$Accuracy
```

```
## [1] 0.842328
```

```
xgb_custom_2$results$Accuracy
```

```
## [1] 0.7987066
```

Y los valores para las generadas aleatoriamente

```
xgb_random$results$Accuracy
```

```
## [1] 0.6992945 0.6963845 0.6997061 0.6945032 0.7103469 0.7071723 0.7126396
## [8] 0.7079071 0.7442975 0.7412698 0.7479424 0.7422105 0.7535273 0.7496473
## [15] 0.7583774 0.7520282 0.7256614 0.7213404 0.7281011 0.7213698 0.7326573
## [22] 0.7293651 0.7336861 0.7311875 0.7688713 0.7638448 0.7705173 0.7644033
## [29] 0.7762493 0.7716049 0.7787772 0.7738095
```

Como podemos observar caret ha creado 32 modelos con diferente parametrización. Nos quedamos con el que haya obtenido una mejor evaluación:

```
max(xgb_random$results$Accuracy)
```

```
## [1] 0.7787772
```

Y echaremos un vistazo a cuál fueron los valores aleatorios asignados a los parámetros en dicho modelo

```
xgb_random$bestTune
```

```
##      nrounds max_depth eta gamma colsample_bytree min_child_weight subsample
## 30        100         2  0.4     0            0.8             1           0.5
```

Referencias de esta sección

- <https://medium.com/@jboscomendoza/xgboost-en-r-398e7c84998e>
- https://docs.aws.amazon.com/es_es/sagemaker/latest/dg/xgboost_hyperparameters.html

Random Forest

Random forest también conocidos en castellano como “Bosques Aleatorios” es una combinación de árboles predictores tal que cada árbol depende de los valores de un vector aleatorio probado independientemente y con la misma distribución para cada uno de estos. Es una modificación del proceso de bagging que consigue mejores resultados gracias a que decorrelaciona los árboles generados en el proceso.

Parámetros

- **ntree**: número de árboles en el bosque. Se quiere estabilizar el error, pero usar demasiados árboles puede ser innecesariamente ineficiente.
- **mtry**: número de variables aleatorias como candidatas en cada ramificación.
- **sampsize**: el número de muestras sobre las cuales entrenar. El valor por defecto es 63.25%.
- **nodesize**: mínimo número de muestras dentro de los nodos terminales. Equilibrio entre bias-varianza
- **maxnodes**: máximo número de nodos terminales.

Creamos el modelo utilizando la raíz cuadrada del número de columnas como valor para el parámetro **mtry**

```
library(randomForest)
rf_tune_grid_1 <- expand.grid(.mtry = sqrt(ncol(train)))

rf_custom_1 <- train(Cover_Type_Desc~.,
                      data=train,
                      method='rf',
                      metric='Accuracy',
                      tuneGrid=rf_tune_grid_1,
                      trControl=control)

rf_random <- train(Cover_Type_Desc~.,
                     data=train,
                     method="rf",
                     metric='Accuracy',
                     tuneLength=2,
                     trControl=control)
```

Comprobamos los valores de *Accuracy*

```
rf_custom_1$results$Accuracy
```

```
## [1] 0.8167842
```

```
rf_random$results$Accuracy
```

```
## [1] 0.6544680 0.8377131
```

De forma análoga al modelo anterior, obtenemos los valores de la mejor parametrización de las asignadas al azar por caret:

```
max(rf_random$results$Accuracy)
```

```
## [1] 0.8377131
```

```
rf_random$bestTune
```

```
##   mtry
## 2    52
```

Referencias de esta sección

- https://es.wikipedia.org/wiki/Random_forest
- https://rpubs.com/Joaquin_AR/255596
- <https://rpubs.com/phamdinhkhanh/389752>
- <https://bookdown.org/content/2031/ensambladores-random-forest-parte-i.html>

Feature selection

Son varias las opciones que nos ofrece **caret** para hacer selección de variables:

Podemos inicialmente, obtener la matriz de correlación:

```
correlationMatrix <- cor(data[, names(data) != "Cover_Type_Desc"])
```

Ahora, vemos los nombres de las variables con un valor de correlación mayor de 0.5:

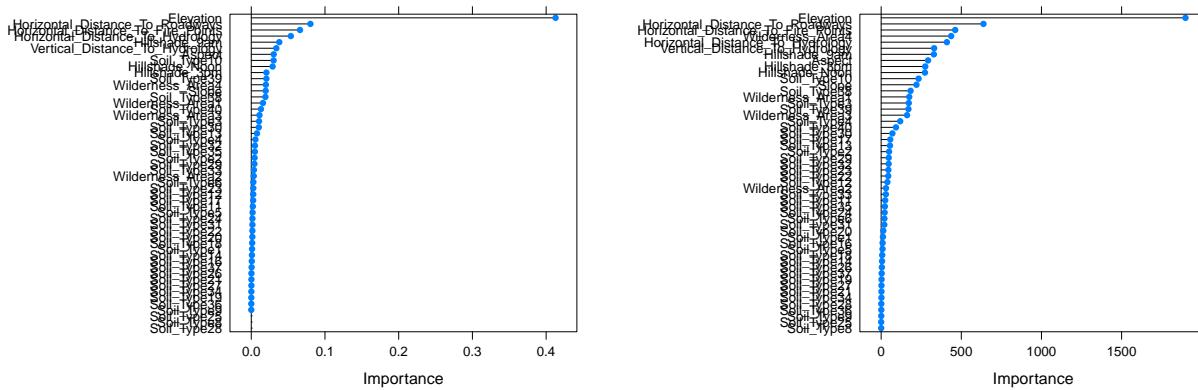
```
highlyCorrelated <- findCorrelation(correlationMatrix, cutoff=0.5)
print(colnames(data[highlyCorrelated]))
```

```
## [1] "Elevation"                      "Wilderness_Area4"
## [3] "Wilderness_Area1"                 "Slope"
## [5] "Hillshade_Noon"                  "Hillshade_3pm"
## [7] "Horizontal_Distance_To_Hydrology" "Hillshade_9am"
```

Mediante la función **varImp()** podemos obtener un valor que muestre la *importancia* de cada variable predictora a la hora de predecir la variable clase con un modelo.

```
xgb_custom_1_importance <- varImp(xgb_custom_1, scale=FALSE)
rf_custom_1_importance <- varImp(rf_custom_1, scale=FALSE)
```

En los siguientes gráficos podemos observar esta clasificación de importancia de variables de nuestros modelos:



Pese a que el tamaño del gráfico obliga a hacer mucho zoom para leer los nombres de las variables, queda bastante claro que algunas de ellas son muy relevantes, mientras que otras no tienen apenas relevancia a la hora de predecir la variable clase.

Finalmente podemos utilizar las funciones **rfe()** y **rfeControl()** para obtener el mejor subconjunto de variables.

Utilizaremos el algoritmo **Random Forest** (parámetro **rfFuncs**) y validación cruzada.

```

rfe_control <- rfeControl(functions=rffFuncs, method="cv", number=10)
#Indicaremos los tamaños de los subconjuntos que queremos probar (en el ejemplo 5 y 10).
results <- rfe(data[, names(data) != "Cover_Type_Desc"], data$Cover_Type_Desc, sizes = c(5, 10), rfeControl)
#Obteniendo las variables resultado
predictors(results)

## [1] "Elevation"                                "Horizontal_Distance_To_Roadways"
## [3] "Horizontal_Distance_To_Hydrology"          "Horizontal_Distance_To_Fire_Points"
## [5] "Vertical_Distance_To_Hydrology"            "Hillshade_Noon"
## [7] "Hillshade_9am"                            "Aspect"
## [9] "Hillshade_3pm"                           "Wilderness_Area4"

#Guardamos una copia de nuestros dataset con las variables seleccionadas
data_rfe <- data[,c(predictors(results),"Cover_Type_Desc")]

```

Referencias de esta sección

- <https://machinelearningmastery.com/feature-selection-with-the-caret-r-package/>

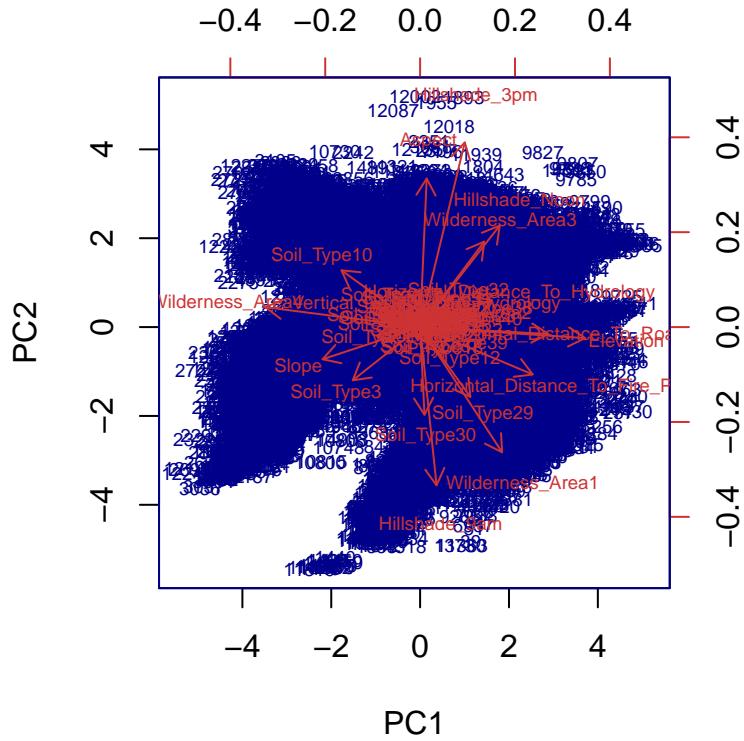
Feature extraction

Utilizamos la función **prcomp()** para aplicar *Principal Component Analysts (PCA)* en el proceso de *feature extraction.

```
data.pca <- prcomp(data[, names(data) != "Cover_Type_Desc"], scale = TRUE)
```

Mediante la función **biplot()** se puede obtener una representación bidimensional de las dos primeras componentes. Es recomendable indicar el argumento `scale = 0` para que las flechas estén en la misma escala que las componentes.

```
biplot(x = data.pca, scale = 0, cex = 0.6, col = c("blue4", "brown3"))
```



Interpretación del gráfico El eje inferior e izquierdo representan la escala de valores de los scores o puntuaciones de las observaciones, mientras que el eje superior y derecho representan la escala de los loadings o cargas, comprendida en un rango [-1, 1].

Para los **vectores (variables)**, nos fijamos en su longitud y en el ángulo con respecto a los ejes de las componentes principales y entre ellos mismos:

- **Ángulo:** cuanto más paralelo es un vector al eje de una componente, más ha contribuido a la creación de la misma.
- **Longitud:** cuanto mayor la longitud de un vector relacionado con x variable (en un rango normalizado de 0 a 1), mayor variabilidad de dicha variable está contenida en la representación de las dos componentes del biplot, es decir, mejor está representada su información en el gráfico.

Para los **scores (observaciones)**, nos fijamos en los posibles agrupamientos. Puntuaciones próximas representan observaciones de similares características. Puntuaciones con valores de las variables próximas a la media se sitúan más cerca del centro del biplot (0, 0).

Referencias de esta sección

- https://rpubs.com/Joaquin_AR/287787#ejemplo-calculo-directo-de-pca-con-r
- https://rpubs.com/Cristina_Gil/PCA

Función *predict()*

Tras realizar una búsqueda en la documentación y en Internet, no se han encontrado parámetros adicionales para la función **predict()** aparte de los ya mencionados **object**, **newdata** y **type (raw/prob)**.

Tan solo es posible asignar un valor a **na.action** para gestionar valores faltantes. También parece que utilizando el paquete **mlr** se pueden utilizar otros parámetros como **subset** y **task** que no encajan con el problema propuesto.

Por tanto lo utilizamos con los parámetros habituales:

```
xgb_custom_1_pred_raw <- predict(xgb_custom_1, newdata = test, type = 'raw')
rf_custom_1_pred_raw <- predict(rf_custom_1, newdata = test, type = 'raw')
```

A continuación obtendremos mediante la función **confusionMatrix()** la matriz de confusión de uno de los modelos, que como vemos nos ofrece mucha información interesante:

```
confusionMatrix(xgb_custom_1_pred_raw, test$Cover_Type_Desc)
```

```
## Confusion Matrix and Statistics
##
##                                     Reference
## Prediction          Aspen Cottonwood/Willow Douglas-fir Krummholtz
##   Aspen              513            0            8            1
##   Cottonwood/Willow    0           526           14            0
##   Douglas-fir          5            7           467            0
##   Krummholtz          0            0            0           525
##   Lodgepole Pine      15            0            5            1
##   Ponderosa Pine       7            7           45            0
##   Spruce/Fir           0            0            1           13
##
##                                     Reference
## Prediction          Lodgepole Pine Ponderosa Pine Spruce/Fir
##   Aspen                  58            5            10
##   Cottonwood/Willow      0            18            0
##   Douglas-fir             14           66            1
##   Krummholtz              4            0           33
##   Lodgepole Pine         358            3           96
##   Ponderosa Pine          16           448            0
##   Spruce/Fir              90            0           400
##
##                                     Overall Statistics
##
##                                     Accuracy : 0.8563
##                                     95% CI : (0.8448, 0.8674)
##                                     No Information Rate : 0.1429
##                                     P-Value [Acc > NIR] : < 2.2e-16
##
##                                     Kappa : 0.8324
##
##                                     Mcnemar's Test P-Value : NA
##
##                                     Statistics by Class:
##
##                                     Class: Aspen Class: Cottonwood/Willow Class: Douglas-fir
```

```

## Sensitivity          0.9500          0.9741          0.8648
## Specificity         0.9747          0.9901          0.9713
## Pos Pred Value     0.8622          0.9427          0.8339
## Neg Pred Value     0.9915          0.9957          0.9773
## Prevalence          0.1429          0.1429          0.1429
## Detection Rate     0.1357          0.1392          0.1235
## Detection Prevalence 0.1574          0.1476          0.1481
## Balanced Accuracy   0.9623          0.9821          0.9181
##                                         Class: Krummholtz Class: Lodgepole Pine
## Sensitivity          0.9722          0.66296         0.66296
## Specificity         0.9886          0.96296         0.96296
## Pos Pred Value      0.9342          0.74895         0.74895
## Neg Pred Value      0.9953          0.94488         0.94488
## Prevalence          0.1429          0.14286         0.14286
## Detection Rate      0.1389          0.09471         0.09471
## Detection Prevalence 0.1487          0.12646         0.12646
## Balanced Accuracy   0.9804          0.81296         0.81296
##                                         Class: Ponderosa Pine Class: Spruce/Fir
## Sensitivity          0.8296          0.7407          0.7407
## Specificity         0.9769          0.9679          0.9679
## Pos Pred Value      0.8566          0.7937          0.7937
## Neg Pred Value      0.9718          0.9573          0.9573
## Prevalence          0.1429          0.1429          0.1429
## Detection Rate      0.1185          0.1058          0.1058
## Detection Prevalence 0.1384          0.1333          0.1333
## Balanced Accuracy   0.9032          0.8543          0.8543

```

Comparativa entre dos modelos

La función `resamples()` es muy útil para hacer una comparación cualitativa entre modelos.

```
resamps=resamples(list(XGBoost=xgb_custom_1,RandomForest=rf_custom_1))
```

Es aconsejable que las predicciones hechas por los modelos separados tengan una baja correlación (< 0.75) cuando se desea aplicar un método de *stacking*.

Para evaluar la correlación se pueden utilizar las funciones `resamples()` y `modelCor()`.

```
modelCor(resamps)
```

```

##                               XGBoost RandomForest
## XGBoost           1.0000000   0.2508742
## RandomForest      0.2508742   1.0000000

```

Se puede comprobar que hay una muy baja correlación entre los dos modelos.

Con la función `summary()` obtendremos una comparativa en forma de tabla de los *scores* de nuestros modelos a comparar.

```
summary(resamps)
```

```
##
```

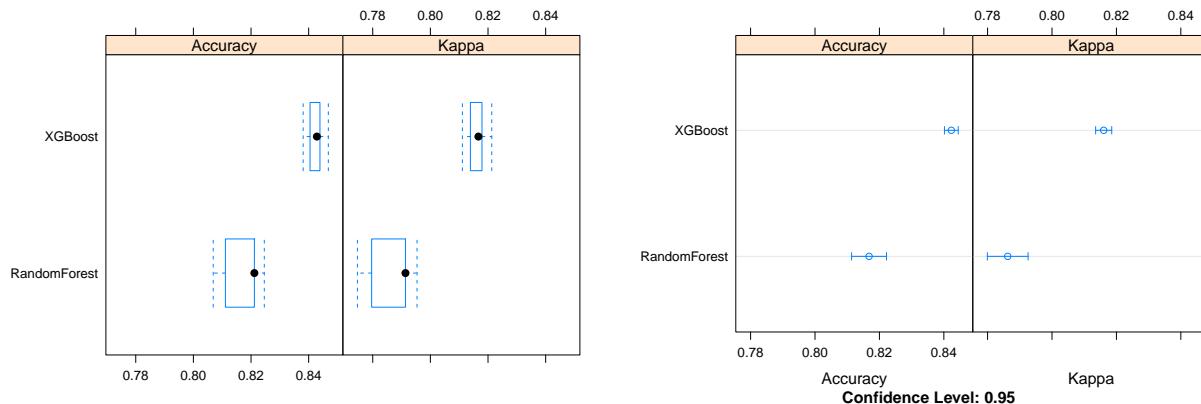
```

## Call:
## summary.resamples(object = resamps)
##
## Models: XGBoost, RandomForest
## Number of resamples: 9
##
## Accuracy
##           Min.   1st Qu.    Median      Mean   3rd Qu.    Max. NA's
## XGBoost 0.8380952 0.8404762 0.8428571 0.8423280 0.8439153 0.8468254 0
## RandomForest 0.8068783 0.8111111 0.8211640 0.8167842 0.8211640 0.8246032 0
##
## Kappa
##           Min.   1st Qu.    Median      Mean   3rd Qu.    Max. NA's
## XGBoost 0.8111111 0.8138889 0.8166667 0.8160494 0.8179012 0.8212963 0
## RandomForest 0.7746914 0.7796296 0.7913580 0.7862483 0.7913580 0.7953704 0

```

En los resultados vemos que el método **XGBoost** se comporta mejor en todas las porciones de los datos.

Podemos comparar visualmente el comportamiento de los dos modelos mediante las funciones **bwplot()** y **dotplot()** que nos devolverán gráficos similares.

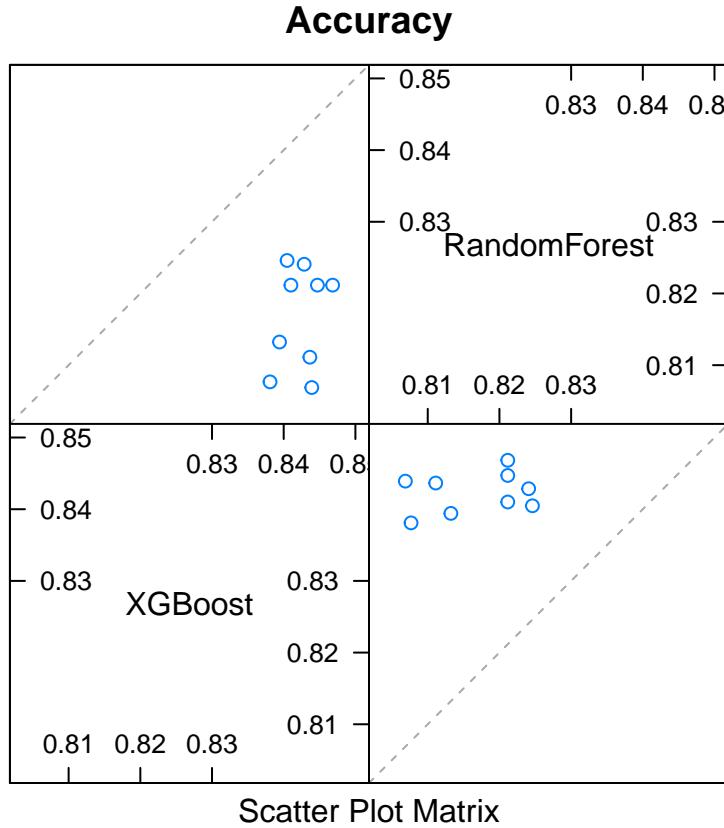


En ellos observamos, aparte de los valores *Accuracy* y *Kappa* medios (punto central), los mejores y peores resultados de cada método en los diferentes tramos.

Vemos de nuevo que incluso el peor *score* de XGBoost supera al mejor de Random Forest.

También existe la función **splom()**, útil para visualizar matrices de diagramas de dispersión, que nos servirá para comparar los *scores* de los modelos.

```
splom(resamps)
```



Vemos que todos los puntos quedan a un lado de la línea, lo que una vez más muestra que XGBoost se comporta mejor que Random Forest en todos los casos.

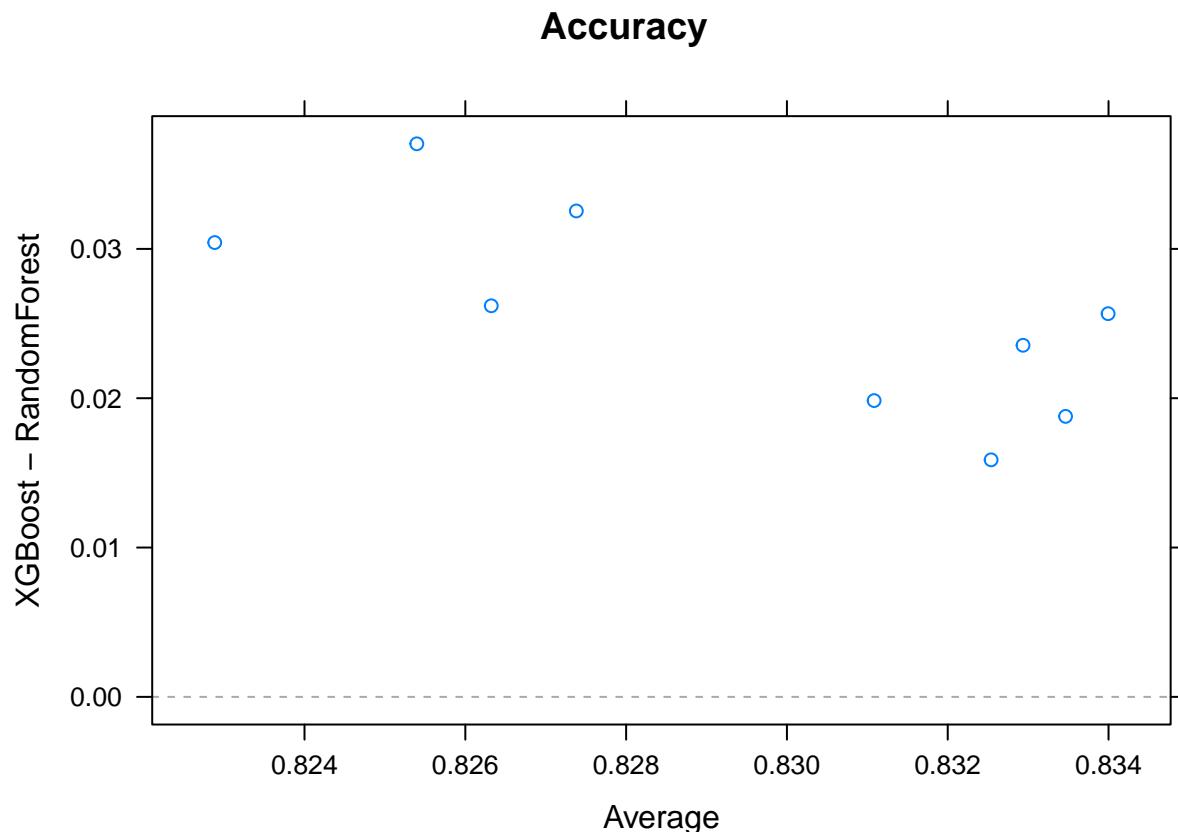
Por último, podemos utilizar la gráfica de Bland-Altman para evaluar la concordancia entre dos mediciones.

Este gráfico no se centra tanto en mostrar qué método se comporta mejor, sino que la dispersión de los puntos nos indicará el grado de concordancia entre ambos.

Para hacer la gráfica del método de **Bland y Altman** debemos calcular dos medidas derivadas de los dos métodos a comparar:

- Eje X: el promedio: $(x + y) / 2$
- Eje Y: la diferencia $(x - y)$

```
xypplot(resamps,what="BlandAltman")
```



Los puntos se agrupan con un grado de dispersión que vendrá determinado por la amplitud de las diferencias de resultados entre los dos métodos.

Cuanta mayor sea ese grado de dispersión, peor será el acuerdo entre los dos métodos.

Finalmente, podemos utilizar la función **diff()** para obtener las diferencias entre los *scores* de los métodos comparados.

```
diffs<-diff(resamps)
summary(diffs)

##
## Call:
## summary.diff.resamples(object = diffs)
##
## p-value adjustment: bonferroni
## Upper diagonal: estimates of the difference
## Lower diagonal: p-value for H0: difference = 0
##
## Accuracy
##           XGBoost   RandomForest
## XGBoost          0.02554
## RandomForest 3.864e-06
##
## Kappa
##           XGBoost   RandomForest
## XGBoost          0.0298
```

```
## RandomForest 3.864e-06
```

Además de las funciones propuestas, también se puede emplear la función **compare_models()**, que también forma parte de los métodos de **summary.diff.resamples**:

```
compare_models(xgb_custom_1, rf_custom_1)
```

```
##  
##  One Sample t-test  
##  
## data: x  
## t = 11.104, df = 8, p-value = 3.864e-06  
## alternative hypothesis: true mean is not equal to 0  
## 95 percent confidence interval:  
##  0.02023899 0.03084861  
## sample estimates:  
## mean of x  
## 0.0255438
```

Referencias de esta sección

- <http://amsantac.co/blog/es/2016/10/22/model-stacking-classification-r-es.html>
- <https://machinelearningmastery.com/compare-models-and-select-the-best-using-the-caret-r-package/>
- <https://anestesiar.org/2015/otra-piedra-con-la-que-no-tropezar-el-metodo-de-bland-altman-para-medir-acuerdo/>

Notas finales

Mediante la realización del proyecto se ha adquirido un mayor conocimiento del paquete **caret** y de todas las opciones que ofrece para las tareas de clasificación en los métodos supervisados.

Una de las mayores **dificultades** a las que ha habido que enfrentarse en este proyecto ha sido el alto **tiempo de computación** que requiere la ejecución de algunas funciones. Esto ha provocado que no se hayan podido realizar todas las pruebas deseadas, ya que la espera entre ejecuciones imposibilitaba el rápido progreso entre las diferentes secciones propuestas para el proyecto.

El **código fuente** del proyecto se incluye dentro del fichero de RMarkdown. Está **disponible aquí**: enlace 1 y enlace 2.