Mejora del rendimiento del modelo (Construyendo mejores aprendices), Parte II

Gradient boosting ('impulso de gradiente')

El **gradient boosting** es una evolución del algoritmo de boosting, basada en el descubrimiento de que es posible tratar el proceso de boosting como un problema de optimización que se resuelve mediante la técnica de descenso de gradiente. El descenso de gradiente se estudia fuertemente en los métodos de caja negra: más especialmente en las redes neuronales, donde se trata de contar con una solución para optimizar los pesos en una red neuronal. Retomando esas ideas, una función de costo (esencialmente, el error de predicción) relaciona los valores de entrada con el objetivo.

Luego, al analizar sistemáticamente cómo los cambios en los pesos afectan al costo, es posible encontrar el conjunto de pesos que minimiza el costo. El gradient boosting trata el proceso de boosting de forma similar, considerando a los aprendices débiles del conjunto como los parámetros a optimizar. Los modelos que utilizan esta técnica se denominan **máquinas de gradient boosting** o **modelos de boosting generalizados** (GBM, *Generalized Boosting Models*).

Para más información sobre los GBM, consulta Greedy Function Approximation: A Gradient Boosting Machine, Friedman JH, 2001, Annals of Statistics 29(5):1189-1232.

La siguiente tabla resume las fortalezas y debilidades de los GBM. En resumen, el aumento de gradiente es extremadamente potente y puede producir algunos de los modelos más precisos, pero puede requerir ajustes para encontrar el equilibrio entre el sobreajuste y el subajuste.

Fortalezas	Debilidades
• Un clasificador multipropósito con un rendimiento excepcional tanto en clasificación como en predicción numérica. • Puede alcanzar un rendimiento incluso mejor que los bosques aleatorios. • Buen rendimiento en grandes conjuntos de datos.	• Puede requerir ajustes para igualar el rendimiento del algoritmo de bosque aleatorio y un ajuste más exhaustivo para superarlo. • Dado que hay varios hiperparámetros que ajustar, encontrar la mejor combinación requiere muchas iteraciones y mayor potencia de cálculo.

Utilizaremos la función gbm() del paquete gbm para crear GBM tanto para clasificación como para predicción numérica. Necesitarás instalar y cargar este paquete en su sesión de R

si aún no lo has hecho. Como se muestra en el siguiente recuadro, la sintaxis es similar a la de las funciones de aprendizaje automático utilizadas anteriormente, pero incluye varios parámetros nuevos que podrían requerir ajustes.

Estos parámetros controlan la complejidad del modelo y el equilibrio entre el sobreajuste y el subajuste. Sin ajustes, el GBM podría no tener el mismo rendimiento que los métodos más simples, pero generalmente puede superar el rendimiento de la mayoría de los demás métodos una vez optimizados los valores de los parámetros.

Gradient Boosting Machine (GBM) syntax

Using the `gbm()` function in the `gbm` package

Building the classifier:

```
m <- gbm(target ~ predictors, data = mydata,
  distribution = "bernoulli", n.trees = 100, interaction.depth = 1,
  n.minobsinnode = 10, shrinkage = 0.1)
```

- `target` is the outcome in the `mydata` data frame to be modeled
- `predictors` is an R formula specifying the features in the `mydata` data frame to use for prediction
- `data` specifies the data frame in which the `target` and `predictors` variables can be found
- `distribution` is the form of target variable; can generally omit this to allow `gbm()` to determine it automatically
- `n.trees` is the number of boosting iterations (total number of trees to fit)
- `interaction.depth` is an integer specifying the maximum depth of each decision tree; deeper trees allow more interactions among predictors
- `n.minobsinnode` is an integer specifying the minimum number of observations in the trees' leaf nodes; smaller or larger values may lead to over- or under-fitting
- `shrinkage` refers to the learning rate, usually between 0.001 and 0.1; smaller values require more trees but may result in better models

The function will return a `gbm` object that can be used to make predictions.

Making predictions:

```
p <- predict(m, test, type = "link")
```

- `m` is a model trained by the `gbm()` function
- `test` is a data frame containing test data with the same features as the training data used to build the classifier
- `type` is either `"link"` or `"response"` and for classification models is used to indicate whether the predictions vector should contain the predicted class or the predicted probabilities, respectively. For numeric prediction this is unnecessary.

The function will return predictions according to the value of the `type` parameter.

Example:

```
credit_model <- gbm(default ~ ., data = credit_train)
credit_prediction <- predict(credit_model, credit_test,
                             type = "response")
```

Figura 7: Sintaxis de la máquina de refuerzo de gradiente (GBM).

Podemos entrenar un GBM simple para predecir los impagos de préstamos en el conjunto de datos de crédito de la siguiente manera. Para simplificar, establecemos stringsAsFactors = TRUE para evitar la recodificación de los predictores, pero luego la característica predeterminada objetivo debe convertirse de nuevo a un resultado binario, ya que la función

gbm() lo requiere para la clasificación binaria. Crearemos una muestra aleatoria para entrenamiento y pruebas, y luego aplicaremos la función gbm() a los datos de entrenamiento, dejando los parámetros predeterminados:

```
> credit <- read.csv("credit.csv", stringsAsFactors = TRUE)
> credit$default <- ifelse(credit$default == "yes", 1, 0)
> set.seed(123)
> train_sample <- sample(1000, 900)
> credit_train <- credit[train_sample, ]
> credit_test <- credit[-train_sample, ]
> library(gbm)
> set.seed(300)
> m_gbm <- gbm(default ~ ., data = credit_train)
```

Escribir el nombre del modelo proporciona información básica sobre el proceso GBM:

```
> m_gbm
gbm(formula = default ~ ., data = credit_train)
A gradient boosted model with bernoulli loss function.
100 iterations were performed.
There were 16 predictors of which 14 had non-zero influence
```

Más importante aún, podemos evaluar el modelo en el conjunto de pruebas. Ten en cuenta que debemos convertir las predicciones a binario, ya que se expresan como probabilidades. Si la probabilidad de impago del préstamo es superior al 50 %, predeciremos impago; de lo contrario, predeciremos no impago. La tabla muestra la concordancia entre los valores predichos y los reales:

```
> p_gbm <- predict(m_gbm, credit_test, type = "response")
Using 100 trees...

> p_gbm_c <- ifelse(p_gbm > 0.50, 1, 0)
> table(credit_test$default, p_gbm_c)
   p_gbm_c
      0   1
  0  60   5
  1  21  14
```

Para medir el rendimiento, aplicaremos la función Kappa() a esta tabla:

```
> library(vcd)
> Kappa(table(credit_test$default, p_gbm_c))
```

```
            value     ASE    z  Pr(>|z|)
Unweighted 0.3612 0.09529 3.79 0.0001504
Weighted   0.3612 0.09529 3.79 0.0001504
```

El valor kappa resultante, de aproximadamente 0.361, es mejor que el obtenido con el árbol de decisión mejorado, pero peor que el del modelo de bosque aleatorio. Quizás, con un poco de ajuste, podamos aumentarlo.

Utilizaremos el paquete caret para ajustar el modelo GBM y obtener una medida de rendimiento más robusta.

Recuerda que el ajuste requiere una rejilla de búsqueda, que podemos definir para GBM de la siguiente manera. Esto probará tres valores para tres de los parámetros de la función gbm() y un valor para el parámetro restante, lo que resulta en 3 * 3 * 3 * 1 = 27 modelos para evaluar:

```
> grid_gbm <- expand.grid(
+   n.trees = c(100, 150, 200),
+   interaction.depth = c(1, 2, 3),
+   shrinkage = c(0.01, 0.1, 0.3),
+   n.minobsinnode = 10
+ )
```

A continuación, configuramos el objeto trainControl para seleccionar el mejor modelo de un experimento de CV de 10-fold:

```
> library(caret)
> ctrl <- trainControl(method = "cv", number = 10,
+   selectionFunction = "best")
```

Por último, leemos el conjunto de datos de crédito y proporcionamos los objetos necesarios a la función caret(), especificando el método gbm y la métrica de rendimiento kappa. Dependiendo de la capacidad de tu computadora, la ejecución puede tardar unos minutos:

```
> credit <- read.csv("credit.csv", stringsAsFactors = TRUE)
> set.seed(300)
> m_gbm_c <- train(default ~ ., data = credit, method = "gbm",
+   trControl = ctrl, tuneGrid = grid_gbm,
+   metric = "Kappa",
+   verbose = FALSE)
```

Al escribir el nombre del objeto, se muestran los resultados del experimento. La salida completa contiene 27 filas, una por cada modelo evaluado:

> m_gbm_c

```
Stochastic Gradient Boosting

1000 samples
  16 predictor
   2 classes: 'no', 'yes'

No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 900, 900, 900, 900, 900, 900, ...
Resampling results across tuning parameters:

  shrinkage  interaction.depth  n.trees  Accuracy  Kappa
  0.01       1                  100      0.700     0.000000000
  0.01       1                  150      0.701     0.007202666
  0.01       1                  200      0.702     0.029773904
  0.01       2                  100      0.702     0.022283199
  0.01       2                  150      0.712     0.090116363
  0.01       2                  200      0.724     0.155099453
  0.01       3                  100      0.706     0.045207862
  0.01       3                  150      0.719     0.128844080
  0.01       3                  200      0.727     0.186716400
  0.10       1                  100      0.737     0.269966697
  0.10       1                  150      0.738     0.295886773
  0.10       1                  200      0.742     0.320157816
  0.10       2                  100      0.747     0.327928587
  0.10       2                  150      0.750     0.347848347
  0.10       2                  200      0.759     0.380641164
  0.10       3                  100      0.747     0.342691964
  0.10       3                  150      0.748     0.356836684
  0.10       3                  200      0.764     0.394578005
  0.30       1                  100      0.741     0.324737072
  0.30       1                  150      0.744     0.344089484
  0.30       1                  200      0.740     0.333844713
  0.30       2                  100      0.745     0.346766663
  0.30       2                  150      0.751     0.369483496
  0.30       2                  200      0.745     0.357998758
  0.30       3                  100      0.735     0.324131549
  0.30       3                  150      0.736     0.332225952
  0.30       3                  200      0.739     0.346842501

Tuning parameter 'n.minobsinnode' was held constant at a value of 10
Kappa was used to select the optimal model using the largest value.
The final values used for the model were n.trees = 200, interaction.depth = 3, shrinkage = 0.1 and n.minobsinnode
 = 10.
```

A partir de la salida, podemos ver que el mejor modelo GBM tuvo un kappa de 0.394, que supera el bosque aleatorio entrenado previamente. Con ajustes adicionales, es posible aumentar aún más el kappa. O, como verá en la siguiente sección, se puede emplear un método de refuerzo más intensivo para lograr un rendimiento aún mejor.

Gradient boosting extremo con XGBoost

Una implementación innovadora de la técnica de gradient boosting se encuentra en el algoritmo **XGBoost** (https://xgboost.ai), que lleva el boosting al "extremo" al mejorar la eficiencia y el rendimiento del algoritmo. Desde su introducción en 2014, XGBoost se ha posicionado en la cima de las clasificaciones de numerosas competiciones de aprendizaje automático.

De hecho, según los autores del algoritmo, de las 29 soluciones ganadoras en Kaggle en 2015, 17 utilizaron el algoritmo XGBoost. Asimismo, en la Copa KDD de 2015, los 10 ganadores

principales utilizaron XGBoost. Hoy en día, el algoritmo sigue siendo el líder en problemas tradicionales de aprendizaje automático que involucran clasificación y predicción numérica, mientras que su rival más cercano, las redes neuronales profundas, tienden a ganar solo en datos no estructurados, como imágenes, audio y procesamiento de texto.

Para más información sobre XGBoost, consulta: XGBoost: A Scalable Tree Boosting Systme, Chen T y Guestrin C, 2016. https://arxiv.org/abs/1603.02754.

La gran potencia del algoritmo XGBoost tiene la desventaja de que no es tan fácil de usar y requiere un ajuste mucho mayor que otros métodos examinados hasta la fecha. Por otro lado, su límite de rendimiento tiende a ser mayor que el de cualquier otro enfoque. Las fortalezas y debilidades de XGBoost se presentan en la siguiente tabla:

Fortalezas	**Debilidades**
• Un clasificador multipropósito con un rendimiento excepcional tanto en clasificación como en predicción numérica. • Quizás, indiscutiblemente, el líder actual en rendimiento en problemas de aprendizaje tradicionales. Gana prácticamente todas las competencias de aprendizaje automático con datos estructurados. • Altamente escalable, funciona bien con grandes conjuntos de datos y puede ejecutarse en paralelo en plataformas de computación distribuida.	• Es más difícil de usar que otras funciones, ya que depende de frameworks externos que no utilizan estructuras de datos nativas de R. • Requiere un ajuste exhaustivo de un gran conjunto de hiperparámetros que pueden ser difíciles de entender sin una sólida formación matemática. • Debido a la gran cantidad de parámetros de ajuste, encontrar la mejor combinación requiere muchas iteraciones y mayor potencia de cálculo. • Da como resultado un modelo de "caja negra" casi imposible de interpretar sin herramientas de explicabilidad.

Para aplicar el algoritmo, utilizaremos la función xgboost() del paquete xgboost, que proporciona una interfaz de R para el framework XGBoost. Se podrían escribir libros enteros sobre este framework, ya que incluye funciones para diversos tipos de tareas de aprendizaje automático y es altamente extensible y adaptable a diversos entornos computacionales de alto rendimiento.

Para obtener más información sobre el framework XGBoost, consulta la excelente documentación disponible en la web: https://xgboost.readthedocs.io.
Nuestro trabajo se centrará en una pequeña parte de su funcionalidad, como se muestra en el siguiente cuadro de sintaxis, que es mucho más denso que el de otros algoritmos debido a un gran aumento en la complejidad y a los hiperparámetros que se pueden ajustar:

Uno de los desafíos de usar XGBoost en R es la necesidad de usar datos en formato matricial en lugar de los formatos preferidos de R, como tibbles o data frames. Dado que XGBoost está diseñado para conjuntos de datos extremadamente grandes, también puede usar matrices dispersas, como las que se describieron en temas anteriores. Recordarás que una matriz

dispersa solo almacena valores distintos de cero, lo que la hace más eficiente en memoria que las matrices tradicionales cuando muchos valores de características son cero.

XGBoost (XGB) syntax

Using the `xgboost()` function in the `xgboost` package

Setting model parameters:

```
params.xgb = list(objective = "binary:logistic",
  max_depth = 6, eta = 0.3, gammma = 0, colsample_bytree = 1,
  min_child_weight = 1, subsample = 1)
```

- `objective` is determined by the target to be modeled; `"binary:logistic"` is for binary classification; use `"multi:softprob"` for categorical outcomes, `"reg:squarederror"` for regression, or `"count:poisson"` for count data
- `max_depth` is between 0 and infinity and defines the maximum depth of any tree; larger values can find more specific patterns but risk overfitting
- `eta` is between 0 and 1 and affects the learning rate; low values limit overfitting, but increase training time and will require more boosting iterations (`nrounds`)
- `gamma` is between 0 and 1 and governs whether the algorithm will continue splitting; lower values can find more specific patterns but risk overfitting
- `colsample_bytree` is between 0 and 1 and defines the % of features that will be selected at random for each tree; use `colsample_bynode` to instead select random features at each split
- `min_child_weight` is between 0 and infinity and is analogous to the minimum examples needed to split; small values find more specific patterns but may overfit
- `subsample` is between 0 and 1 and defines the % of randomly selected examples for each iteration; small values may prevent overfitting but require more iterations

Note that the above includes only a subset of parameters and hyperparameters. See `?xgboost` or https://xgboost.readthedocs.io/en/latest/parameter.html for the full list.

Building the classifier:

```
m <- xgboost(params, data = mydata, label = mylabels, rounds = n)
```

- `params` is a list of XGBoost hyperparameters (as defined above)
- `data` is a matrix or sparse matrix with the features to be used for training
- `label` is a vector of target values to be used for training
- `rounds` is the maximum number of boosting iterations

The function will return an `xgb` object that can be used to make predictions.

Making predictions:

```
p <- predict(m, test)
```

- `m` is a model trained by the `xgboost()` function
- `test` is a data frame containing test data in the same form as the training data

The function returns predicted probabilities (for classifiers) or values (for numeric models).

Figura 8: Sintaxis de XGBoost (XGB).

Los datos en formato matricial suelen ser dispersos porque los factores suelen codificarse con un valor único o ficticio durante la transición entre el data frame y la matriz. Estas codificaciones crean columnas adicionales para niveles adicionales del factor, y todas las columnas se establecen en cero excepto el valor “único (*hot*)” que indica el nivel para el ejemplo dado. En el caso de la codificación ficticia, se omite un nivel de característica en la transformación, por lo que se obtiene una columna menos que con un valor único. El nivel faltante puede indicarse por la presencia de ceros en las n-1 columnas.

La codificación one-hot y la codificación ficticia generalmente producen los mismos resultados, con la excepción de que los modelos basados en estadísticas, como la regresión,

requieren codificación ficticia y mostrarán errores o mensajes de advertencia si se utiliza one-hot.

Comencemos leyendo el archivo credit.csv y creando una matriz dispersa de datos a partir del frame de datos credit. El paquete Matrix proporciona una función para realizar esta tarea, que utiliza la interfaz de fórmulas de R para determinar las columnas que se incluirán en la matriz. Aquí, la fórmula ~.-default indica a la función que utilice todas las características excepto la predeterminada, que no queremos en la matriz, ya que es nuestra característica objetivo para la predicción (nota, debes usar una versión de R superior a 4.4 por el paquete Matrix):

```
> credit <- read.csv("credit.csv", stringsAsFactors = TRUE)
> library(Matrix)
> credit_matrix <- sparse.model.matrix(~ . -default, data = credit)
```

Para confirmar nuestro trabajo, revisemos las dimensiones de la matriz:

```
> dim(credit_matrix)
[1] 1000   36
```

Aún tenemos 1000 filas, pero las columnas han aumentado de 16 características en el frame de datos original a 36 en la matriz dispersa. Esto se debe a la codificación ficticia que se aplicó automáticamente al convertir a formato matricial. Podemos comprobarlo si examinamos las primeras cinco filas y 15 columnas de la matriz dispersa con la función print():

```
> print(credit_matrix[1:5, 1:15])
```

```
5 x 15 sparse Matrix of class "dgCMatrix"
  [[ suppressing 15 column names '(Intercept)', 'checking_balance> 200 DM', 'checking_balance1 - 200 DM' ... ]]

1 1 . . .  6 . . . . . . . 1 . 1169
2 1 . 1 . 48 1 . . . . . . 1 . 5951
3 1 . . 1 12 . . . . . . 1 . . 2096
4 1 . . . 42 1 . . . . . . 1 . 7882
5 1 . . . 24 . . 1 . 1 . . . . 4870
```

La matriz se representa con el punto (.) que indica las celdas con valores cero. La primera columna (1, 2, 3, 4, 5) corresponde al número de fila y la segunda (1, 1, 1, 1, 1) corresponde al término de intersección, añadido automáticamente por la interfaz de fórmulas de R. Dos columnas tienen los números (6, 48, …) y (1169, 5951, …) que corresponden a los valores numéricos de las características "months_loan_duration" e "mount", respectivamente.

El resto de las columnas son versiones con codificación ficticia de las variables factoriales. Por ejemplo, la tercera, cuarta y quinta columnas reflejan la característica

"checking_balance": un 1 en la tercera indica un valor de ">200 DM", un 1 en la cuarta indica "1 – 200 DM" y un 1 en la quinta indica el valor de la característica "desconocido". Las filas que muestran la secuencia... en las columnas 3, 4 y 5 pertenecen a la categoría de referencia, que era el nivel de característica '< 0 DM'.

Dado que no estamos construyendo un modelo de regresión, la columna de intersección, llena de valores 1, es inútil para este análisis y puede eliminarse de la matriz:

```
> credit_matrix <- credit_matrix[, -1]
```

A continuación, dividiremos la matriz aleatoriamente en conjuntos de entrenamiento y prueba utilizando una división 90-10, como hicimos anteriormente:

```
> set.seed(12345)
> train_ids <- sample(1000, 900)
> credit_train <- credit_matrix[train_ids, ]
> credit_test <- credit_matrix[-train_ids, ]
```

Para confirmar que el trabajo se realizó correctamente, comprobaremos las dimensiones de estas matrices:

```
> dim(credit_train)
[1] 900  35
> dim(credit_test)
[1] 100  35
```

Como era de esperar, el conjunto de entrenamiento tiene 900 filas y 35 columnas, y el conjunto de prueba tiene 100 filas y un conjunto de columnas coincidente.

Por último, crearemos vectores de entrenamiento y prueba de etiquetas para el valor predeterminado, el objetivo a predecir. Estos se transforman de factores a valores binarios 1 o 0 mediante la función ifelse() para que puedan usarse para entrenar y evaluar el modelo XGBoost, respectivamente:

```
> credit_train_labels <-
+   ifelse(credit[train_ids, c("default")] == "yes", 1, 0)
> credit_test_labels <-
+   ifelse(credit[-train_ids, c("default")] == "yes", 1, 0)
```

Ahora estamos listos para empezar a construir el modelo. Tras instalar el paquete xgboost, cargaremos la biblioteca y comenzaremos a definir los hiperparámetros para el entrenamiento. Sin saber por dónde empezar, restableceremos los valores predeterminados:

```
> library(xgboost)
> params.xgb <- list(objective = "binary:logistic",
+   max_depth = 6,
+   eta = 0.3,
+   gamma = 0,
+   colsample_bytree = 1,
+   min_child_weight = 1,
+   subsample = 1)
```

A continuación, tras establecer la semilla aleatoria, entrenaremos el modelo, proporcionando nuestro objeto de parámetros, así como la matriz de datos de entrenamiento y las etiquetas objetivo. El parámetro nrounds determina el número de iteraciones de refuerzo. Sin una mejor estimación, lo estableceremos en 100, un punto de partida común debido a la evidencia empírica que sugiere que los resultados tienden a mejorar muy poco más allá de este valor. Por último, las opciones verbose y print_every_n se utilizan para activar la salida de diagnóstico y mostrar el progreso después de cada 10 iteraciones de refuerzo:

```
> set.seed(555)
> xgb_credit <- xgboost(params = params.xgb,
+   data = credit_train,
+   label = credit_train_labels,
+   nrounds = 100,
+   verbose = 1,
+   print_every_n = 10)
```

La salida debería aparecer al finalizar el entrenamiento, mostrando que se realizaron las 100 iteraciones y que el error de entrenamiento (etiquetado como "train-logloss") continuó disminuyendo con rondas adicionales de refuerzo:

```
[1]     train-logloss:0.586271
[11]    train-logloss:0.317767
[21]    train-logloss:0.223844
[31]    train-logloss:0.179252
[41]    train-logloss:0.135629
[51]    train-logloss:0.108353
[61]    train-logloss:0.090580
[71]    train-logloss:0.077314
[81]    train-logloss:0.065995
[91]    train-logloss:0.057018
[100]   train-logloss:0.050837
```

Podemos determinar si las iteraciones adicionales mejorarán el rendimiento del modelo o provocarán un sobreajuste mediante un ajuste posterior. Antes de hacerlo, analicemos el rendimiento de este modelo entrenado en el conjunto de prueba, que presentamos anteriormente. Primero, la función predict() obtiene la probabilidad predicha de impago del préstamo para cada fila de datos de prueba:

```
> prob_default <- predict(xgb_credit, credit_test)
```

Luego, usamos ifelse() para predecir un impago (valor 1) si la probabilidad de impago es de al menos 0,50, o un impago no previsto (valor 0) en caso contrario:

```
> pred_default <- ifelse(prob_default > 0.50, 1, 0)
```

Comparando los valores predichos con los reales, encontramos una precisión de (62 + 14) / 100 = 76 %:

```
> table(pred_default, credit_test_labels)
```

```
            credit_test_labels
pred_default  0  1
           0 62 13
           1 11 14
```

Por otro lado, el estadístico kappa sugiere que aún hay margen de mejora:

```
> library(vcd)
> Kappa(table(pred_default, credit_test_labels))
```

```
             value    ASE     z  Pr(>|z|)
Unweighted 0.3766 0.1041 3.618 0.0002967
Weighted   0.3766 0.1041 3.618 0.0002967
```

El valor de 0.3766 es ligeramente inferior al 0.394 obtenido con el modelo GBM, por lo que quizás un pequeño ajuste de hiperparámetros pueda ser útil. Para ello, usaremos caret, comenzando con una rejilla de ajuste que incluye diversas opciones para cada hiperparámetro:

```
> grid_xgb <- expand.grid(
+   eta = c(0.3, 0.4),
+   max_depth = c(1, 2, 3),
+   colsample_bytree = c(0.6, 0.8),
+   subsample = c(0.50, 0.75, 1.00),
+   nrounds = c(50, 100, 150),
+   gamma = c(0, 1),
+   min_child_weight = 1
+ )
```

La rejilla resultante contiene 2 * 3 * 2 * 3 * 3 * 2 * 1 = 216 combinaciones diferentes de valores del hiperparámetro xgboost. Evaluaremos cada uno de estos modelos potenciales en caret utilizando un coeficiente de variación (CV) de 10-fold, como hemos hecho con otros

modelos. Ten en cuenta que el parámetro de verbosidad se establece en cero para que la salida de la función xgboost() se suprima en las iteraciones:

```
> library(caret)
> ctrl <- trainControl(method = "cv", number = 10,
+   selectionFunction = "best")
> credit <- read.csv("credit.csv", stringsAsFactors = TRUE)
> set.seed(300)
> m_xgb <- train(default ~ ., data = credit, method = "xgbTree",
+   trControl = ctrl, tuneGrid = grid_xgb,
+   metric = "Kappa", verbosity = 0)
```

Dependiendo de la capacidad de su computadora, el experimento puede tardar unos minutos en completarse, pero una vez finalizado, al escribir m_xgb se obtendrán los resultados de los 216 modelos probados. También podemos obtener el mejor modelo directamente de la siguiente manera:

```
> m_xgb$bestTune
```

	nrounds	max_depth	eta	gamma	colsample_bytree	min_child_weight	subsample
42	150	2	0.3	0	0.6	1	0.75

El valor kappa de este modelo se puede calcular utilizando la función max() para encontrar el valor más alto, como se indica a continuación:

```
> max(m_xgb$results["Kappa"])
[1] 0.4230198
```

El valor kappa de 0.423 es nuestro modelo con mejor rendimiento hasta la fecha, superando el 0.394 del modelo GBM y el 0.381 del bosque aleatorio. El hecho de que XGBoost requiriera tan poco esfuerzo de entrenamiento (con algunos ajustes) y, aun así, superara a otras técnicas potentes, es un ejemplo de por qué siempre parece ganar las competiciones de aprendizaje automático.

Sin embargo, con un ajuste aún mayor, ¡podría ser posible alcanzar valores aún mayores!

Dejando esto como ejercicio para el lector, ahora nos centraremos en la pregunta de por qué todos estos conjuntos populares parecen centrarse exclusivamente en métodos basados en árboles de decisión.

¿Por qué son tan populares los conjuntos basados en árboles?

Después de leer las secciones anteriores, no serás el primero en preguntarse por qué los algoritmos de conjuntos parecen basarse siempre en árboles de decisión. Aunque los árboles no son necesarios para construir un conjunto, existen varias razones por las que son especialmente adecuados para este proceso. Quizás ya haya notado algunos detalles:

- Los conjuntos funcionan mejor con diversidad, y dado que los árboles de decisión no son robustos a pequeños cambios en los datos, el muestreo aleatorio de los mismos datos de entrenamiento puede crear fácilmente un conjunto diverso de modelos basados en árboles.
- Gracias al algoritmo voraz basado en "divide y vencerás", los árboles de decisión son computacionalmente eficientes y, a pesar de ello, tienen un rendimiento relativamente bueno.
- Los árboles de decisión pueden aumentar o reducirse a propósito para sobreajustar o subajustar según sea necesario.
- Los árboles de decisión pueden ignorar automáticamente las características irrelevantes, lo que reduce el impacto negativo de la "maldición de la dimensionalidad".
- Los árboles de decisión pueden utilizarse tanto para la predicción numérica como para la clasificación.

Con base en estas características, no es difícil entender por qué hemos desarrollado una gran cantidad de enfoques de conjuntos basados en árboles, como bagging, boosting y los bosques aleatorios. Las distinciones entre ellos son sutiles pero importantes.

La siguiente tabla puede ayudar a contrastar los algoritmos de ensamble basados en árboles que se tratan en este documento:

Algoritmo de ensamble	**Función de asignación**	**Función de combinación**	**Otras notas**
Bagging	Proporciona a cada aprendiz una muestra de bootstrap de los datos de entrenamiento.	Los aprendices se combinan mediante una votación para la clasificación o un promedio ponderado para la predicción numérica.	Utiliza un conjunto independiente: - los apredices pueden ejecutarse en paralelo.
Boosting	Al primer aprendiz se le da una muestra aleatoria; las subsiguientes muestras se ponderan para tener casos más difíciles de predecir.	Las predicciones de los aprendices se combinan como se indicó anteriormente, pero se ponderan según su desempeño en los datos de entrenamiento.	Utiliza un conjunto dependiente - cada árbol de la secuencia recibe datos que los árboles anteriores consideraron desafiantes.

Random Forest	Al igual que el bagging, cada árbol recibe una muestra de bootstrap de datos de entrenamiento; sin embargo, las características también se seleccionan aleatoriamente para cada división del árbol.	Similar al bagging.	Similar al bagging, pero la diversidad añadida mediante la selección aleatoria de características permite beneficios adicionales para conjuntos más grandes.
Gradient Boosting Machine (GBM)	Conceptualmente similar al boosting.	Similar al boosting, pero hay muchos más aprendices y comprenden una función matemática compleja.	Utiliza el descenso de gradiente para crear un algoritmo de boosting más eficiente; los árboles generalmente no son muy profundos (árboles de decisión "stumps"), pero hay muchos más; requiere más ajustes.
eXtreme Gradient Boosting (XGB)	Similar a GBM.	Similar a GBM.	Similar a GBM, pero más extremo; utiliza estructuras de datos optimizadas, procesamiento paralelo y heurísticas para crear un algoritmo de boosting de alto rendimiento; el ajuste es esencial.

Poder distinguir entre estos enfoques revela un profundo conocimiento de varios aspectos del ensamble. Además, las técnicas más recientes, como los bosques aleatorios y el gradient boosting, se encuentran entre los algoritmos de aprendizaje de mayor rendimiento y se utilizan como soluciones estándar para resolver algunos de los problemas empresariales más complejos.

Esto puede explicar por qué las empresas que contratan científicos de datos e ingenieros de aprendizaje automático suelen pedir a los candidatos que describan o comparen estos algoritmos durante el proceso de entrevista. Por lo tanto, aunque los algoritmos de ensamble basados en árboles no son el único enfoque para el aprendizaje automático, es importante conocer sus posibles usos. Sin embargo, como se describe en la siguiente sección, los árboles no son el único enfoque para construir un ensamble diverso.

Apilamiento (*stacking*) de modelos para meta-aprendizaje

En lugar de utilizar un método de ensamble predefinido como bagging, boosting o bosques aleatorios, existen situaciones en las que se justifica un enfoque de ensamble personalizado. Si bien estas técnicas de ensamble basadas en árboles combinan cientos o incluso miles de aprendices en un único aprendiz más potente, el proceso no difiere mucho del entrenamiento de un algoritmo tradicional de aprendizaje automático y presenta algunas de las mismas limitaciones, aunque en menor grado.

Basarse en árboles de decisión con un entrenamiento débil y un ajuste mínimo puede, en algunos casos, limitar el rendimiento del ensamble en comparación con uno compuesto por un conjunto más diverso de algoritmos de aprendizaje que se han ajustado exhaustivamente con la ayuda de la inteligencia humana.

Además, si bien es posible paralelizar ensambles basados en árboles como bosques aleatorios y XGB, esto solo paraleliza el esfuerzo de la computadora, no el esfuerzo humano en la construcción del modelo. De hecho, es posible aumentar la diversidad de un conjunto no solo añadiendo algoritmos de aprendizaje adicionales, sino también distribuyendo el trabajo de construcción de modelos a otros equipos que trabajan en paralelo.

De hecho, muchos de los modelos ganadores de competiciones mundiales se construyeron combinando los mejores modelos de otros equipos.

Este tipo de conjunto es conceptualmente bastante simple y ofrece mejoras de rendimiento que de otro modo serían inalcanzables, pero puede volverse complejo en la práctica. Acertar con los detalles de implementación es crucial para evitar niveles desastrosos de sobreajuste. Si se realiza correctamente, el conjunto tendrá un rendimiento al menos tan bueno como el modelo más potente del conjunto, e incluso considerablemente mejor.

El análisis de las curvas ROC, como se presenta en el tema, Evaluación del Rendimiento del Modelo, proporciona un método sencillo para determinar si dos o más modelos se beneficiarían del ensamblaje. Si dos modelos tienen curvas ROC que se intersecan, su envoltura convexa (el límite exterior que se obtendría estirando una banda elástica imaginaria alrededor de las curvas) representa un modelo hipotético que puede obtenerse interpolando o combinando las predicciones de estos modelos.

Como se muestra en la figura 9, dos curvas ROC con valores idénticos de área bajo la curva (AUC) de 0.70 podrían crear un nuevo modelo con un AUC de 0.72 al combinarse en un conjunto:

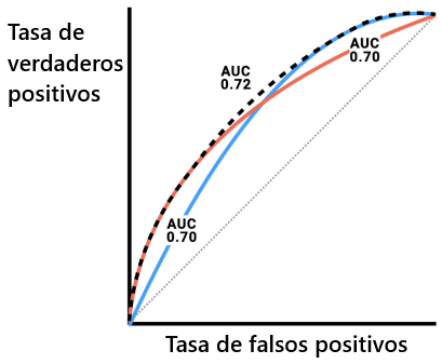

Figura 9: Cuando dos o más curvas ROC se intersecan, su envoltura convexa representa un clasificador potencialmente mejor que puede generarse combinando sus predicciones en un conjunto.

Dado que esta forma de agrupamiento se realiza en gran medida manualmente, es necesario que un usuario proporcione las funciones de asignación y combinación para los modelos del conjunto. En su forma más simple, estas pueden implementarse de forma bastante pragmática. Por ejemplo, supongamos que el mismo conjunto de datos de entrenamiento se ha asignado a tres equipos diferentes. Esta es la función de asignación. Estos equipos pueden utilizar este conjunto de datos como consideren oportuno para construir el mejor modelo posible utilizando criterios de evaluación de su elección.

A continuación, cada equipo recibe el conjunto de prueba y sus modelos se utilizan para realizar predicciones, que deben combinarse en una única predicción final. La función de combinación puede adoptar diversas formas: los grupos podrían votar, las predicciones podrían promediarse o las predicciones podrían ponderarse según el rendimiento previo de cada grupo. Incluso el simple enfoque de elegir un grupo al azar es una estrategia viable, suponiendo que cada grupo tenga un rendimiento superior al de los demás al menos ocasionalmente. Por supuesto, existen enfoques aún más inteligentes, como pronto aprenderás.

Comprensión del apilamiento y la combinación de modelos

Algunos de los conjuntos personalizados más sofisticados aplican aprendizaje automático para aprender una función de combinación para la predicción final. En esencia, intenta aprender en qué modelos se puede confiar y en cuáles no.

Este aprendiz árbitro puede darse cuenta de que un modelo del conjunto tiene un rendimiento deficiente y no se debe confiar en él, o de que otro merece más peso en el conjunto. La función árbitro también puede aprender patrones más complejos. Por ejemplo, supongamos que cuando los modelos M1 y M2 coinciden en el resultado, la predicción es casi siempre precisa; sin embargo, por lo demás, M3 suele ser más preciso que cualquiera de los dos.

En este caso, un modelo árbitro adicional podría aprender a ignorar el voto de M1 y M2, excepto cuando coincidan. Este proceso de usar las predicciones de varios modelos para entrenar un modelo final se denomina apilamiento.

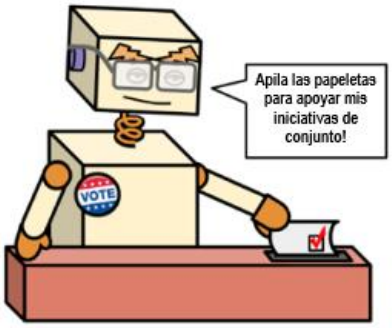

Figura 10: El apilamiento es un conjunto sofisticado que utiliza un algoritmo de aprendizaje árbitro para combinar las predicciones de un conjunto de aprendices y generar una predicción final.

En términos más generales, el apilamiento se enmarca en una metodología conocida como **generalización apilada**. Según su definición formal, la pila se construye utilizando modelos de primer nivel entrenados mediante CV y un modelo de segundo nivel o metamodelo que se entrena utilizando las predicciones de las muestras fuera de pliegue (*fold*) (los ejemplos que el modelo no ve durante el entrenamiento, pero que se prueban durante el proceso de CV).

Por ejemplo, supongamos que se incluyen tres modelos de primer nivel en la pila y que cada uno se entrena utilizando CV de 10-pliegues. Si el conjunto de datos de entrenamiento incluye 1000 filas, cada uno de los tres modelos de la primera etapa se entrena con 900 filas y se prueba con 100 filas diez veces. Los conjuntos de prueba de 100 filas, al combinarse, conforman el conjunto de datos de entrenamiento completo.

Como los tres modelos han realizado una predicción para cada fila de los datos de entrenamiento, se puede construir una nueva tabla con cuatro columnas y 1000 filas: las tres primeras columnas representan las predicciones para los tres modelos y la cuarta columna representa el valor real del objetivo.

Ten en cuenta que, dado que las predicciones realizadas para cada una de estas 100 filas se realizaron en las otras 900 filas, las 1000 filas son predicciones sobre datos no vistos. Esto permite que el metamodelo de la segunda etapa, que suele ser un modelo de regresión o regresión logística, aprenda qué modelos de la primera etapa funcionan mejor entrenándolos con los valores predichos como predictores del valor real. Este proceso de encontrar la combinación óptima de aprendices se denomina a veces **super aprendizaje**, y el modelo

resultante puede denominarse **super aprendiz**. Este proceso suele realizarse mediante software o paquetes de aprendizaje automático, que entrenan numerosos algoritmos de aprendizaje en paralelo y los apilan automáticamente.

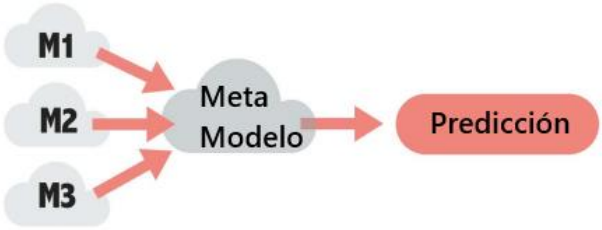

Figura 11: En un conjunto apilado, el metamodelo de segunda etapa o "super aprendiz" aprende de las predicciones de los modelos de primera etapa en muestras fuera del plegado (*out-of-fold*).

Para un enfoque más práctico, un caso especial de generalización apilada, denominado **apilamiento de mezcla** (*blending*) o **de reserva** (*holdout*), ofrece una forma simplificada de implementar el apilamiento al reemplazar el CV por una muestra de reserva.

Esto permite distribuir el trabajo entre los equipos con mayor facilidad, simplemente dividiendo los datos de entrenamiento en un conjunto de entrenamiento para los modelos de primer nivel y utilizando un conjunto de reserva para el meta-aprendiz de segundo nivel.

También puede ser menos propenso al sobreajuste de la "fuga de información" del CV. Por lo tanto, aunque es un enfoque simple, puede ser bastante eficaz. La mezcla es a menudo lo que hacen los equipos ganadores de la competición cuando toman otros modelos y los combinan para obtener mejores resultados.

La terminología en torno al apilamiento, la combinación y el super aprendizaje es algo imprecisa, y muchos los usan indistintamente.

Métodos prácticos para la combinación y el apilamiento en R

Realizar la combinación en R requiere una estrategia meticulosa, ya que un error en los detalles puede provocar un sobreajuste extremo y modelos cuyo rendimiento sea inferior al de una suposición aleatoria. La siguiente figura ilustra el proceso.

Imagina que debes predecir impagos de préstamos y tienes acceso a un millón de filas de datos históricos. Inmediatamente, debes dividir el conjunto de datos en conjuntos de entrenamiento y de prueba; por supuesto, el conjunto de prueba debe guardarse en un almacén para evaluar el conjunto posteriormente. Suponga que el conjunto de entrenamiento tiene 750,000 filas y el de prueba, 250,000. A continuación, el conjunto de entrenamiento debe

dividirse de nuevo para crear conjuntos de datos para entrenar los modelos de nivel uno y el meta-aprendiz de nivel dos. Las proporciones exactas son algo arbitrarias, pero se suele utilizar un conjunto menor para el modelo de segunda etapa, a veces tan bajo como el diez por ciento. Como se muestra en la figura 12, podríamos usar 500,000 filas para el nivel uno y 250,000 filas para el nivel dos:

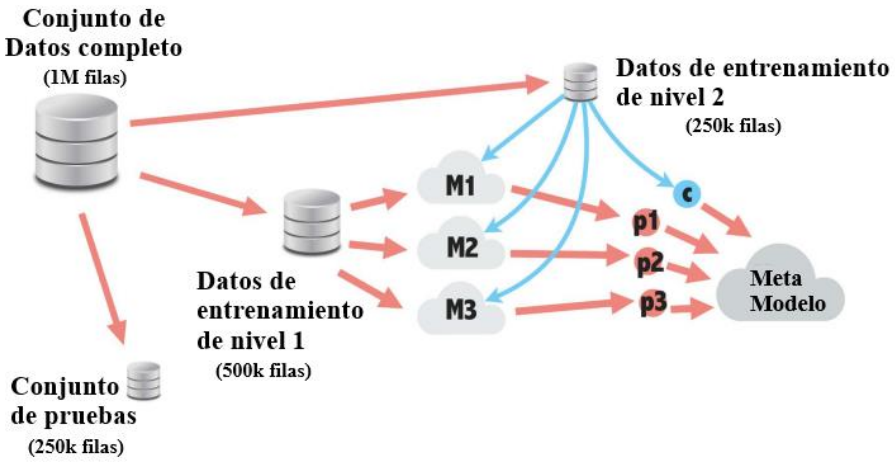

Figura 12: El conjunto de datos de entrenamiento completo debe dividirse en subconjuntos distintos para entrenar los modelos de nivel uno y nivel dos.

El conjunto de datos de entrenamiento de nivel uno de 500,000 filas se utiliza para entrenar los modelos de primer nivel exactamente como lo hemos hecho en repetidas ocasiones a lo largo de este curso. Los modelos M1, M2 y M3 pueden usar cualquier algoritmo de aprendizaje, y el trabajo de construcción de estos modelos puede incluso distribuirse entre diferentes equipos que trabajen de forma independiente.

No es necesario que los modelos o equipos utilicen el mismo conjunto de características de los datos de entrenamiento ni el mismo método de ingeniería de características, suponiendo que el proceso de ingeniería de características de cada equipo pueda replicarse o automatizarse en el futuro cuando se implemente el conjunto. Lo importante es que M1, M2 y M3 puedan tomar un conjunto de datos con características idénticas y generar una predicción para cada fila.

El conjunto de datos de entrenamiento de nivel dos, con 250,000 filas, se incorpora a los modelos M1, M2 y M3 tras ser procesado a través de sus canales de ingeniería de características asociados, obteniéndose así tres vectores con 250,000 predicciones. Estos vectores se etiquetan como p1, p2 y p3 en el diagrama. Al combinarse con los 250,000 valores reales del objetivo (etiquetados como c en el diagrama) obtenidos del conjunto de datos de entrenamiento de nivel dos, se genera un frame de datos de cuatro columnas, como se muestra en la figura 13:

M_1 Prediction	M_2 Prediction	M_3 Prediction	Actual Value
Yes	Yes	Yes	Yes
No	No	Yes	Yes
No	Yes	Yes	Yes
Yes	No	No	No

Figura 13: El conjunto de datos utilizado para entrenar el metamodelo se compone de las predicciones de los modelos de primer nivel y el valor objetivo real de los datos de entrenamiento de nivel dos.

Este tipo de frame de datos se utiliza para crear un metamodelo, generalmente mediante regresión o regresión logística, que predice el valor objetivo real (c en la figura 12) utilizando las predicciones de M1, M2 y M3 (p1, p2 y p3 en la figura 12) como predictores. En una fórmula de R, esto podría especificarse como c ~ p1 + p2 + p3, lo que resulta en un modelo que pondera la entrada de tres predicciones diferentes para generar su propia predicción final.

Para estimar el rendimiento futuro de este metamodelo final, debemos utilizar el conjunto de pruebas de 250,000 filas, que, como se ilustró anteriormente en la figura 12, se mantuvo durante el proceso de entrenamiento. Como se muestra en la figura 14, el conjunto de datos de pruebas se introduce en los modelos M1, M2 y M3 y sus canales de ingeniería de características asociados. Al igual que en el paso anterior, se obtienen tres vectores de 250,000 predicciones.

Sin embargo, en lugar de usar p1, p2 y p3 para entrenar un metamodelo, ahora se utilizan como predictores del metamodelo existente para obtener una predicción final (denominada p4) para cada uno de los 250,000 casos de prueba. Este vector puede compararse con los 250,000 valores reales del objetivo en el conjunto de prueba para realizar la evaluación del rendimiento y obtener una estimación objetiva del rendimiento futuro del conjunto.

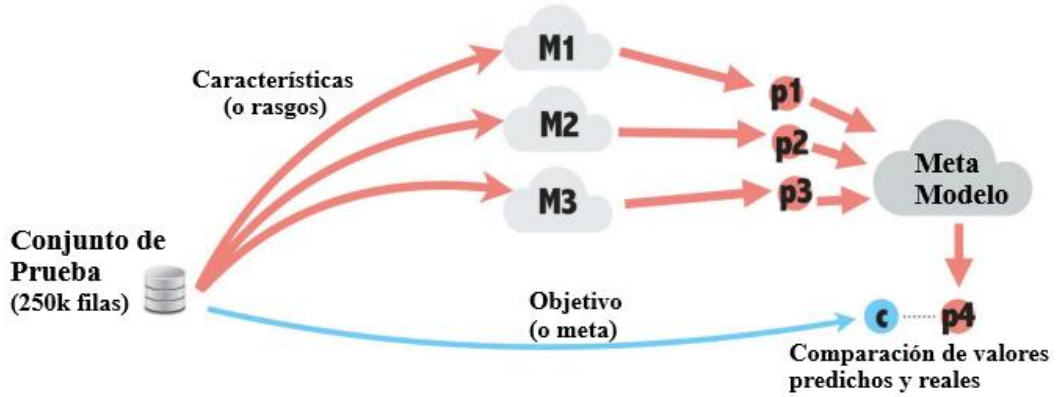

Figura 14: Para obtener una estimación imparcial del rendimiento futuro del conjunto, el conjunto de prueba se utiliza para generar predicciones para los modelos de nivel uno, que posteriormente se utilizan para obtener las predicciones finales del metamodelo.

La metodología anterior permite crear otros tipos de conjuntos interesantes. La figura 15 ilustra un conjunto combinado que combina modelos entrenados con subconjuntos de características completamente diferentes. Específicamente, prevé una tarea de aprendizaje en la que se utilizan los datos del perfil de Twitter (ahora X) para realizar una predicción sobre el usuario, como su género o si estaría interesado en comprar un producto en particular.

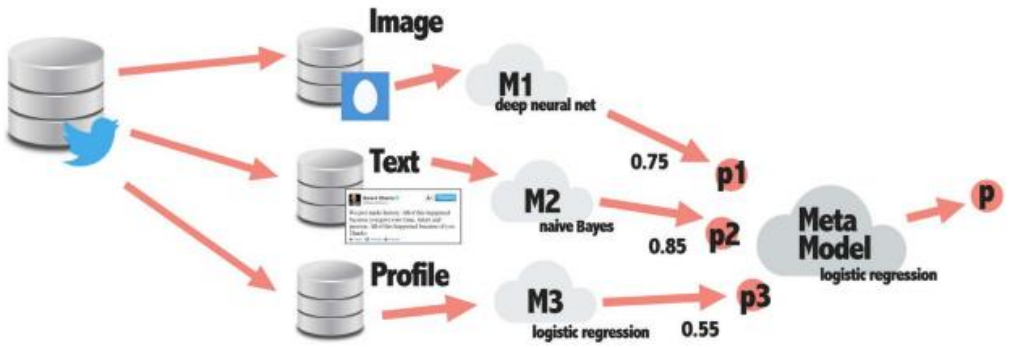

Figura 15: Los modelos de primer nivel de la pila pueden entrenarse con diferentes características del conjunto de entrenamiento, mientras que el modelo de segundo nivel se entrena con sus predicciones.

El primer modelo recibe la imagen del perfil y entrena una red neuronal de aprendizaje profundo con los datos de la imagen para predecir el resultado. El segundo modelo recibe un conjunto de tuits del usuario y utiliza un modelo basado en texto, como Naive Bayes, para predecir el resultado. Finalmente, el modelo tres es un modelo más convencional que utiliza un frame de datos tradicional con datos demográficos como ubicación, número total de tuits, fecha del último inicio de sesión, etc.

Los tres modelos se combinan, y el metamodelo puede determinar si los datos de imagen, texto o perfil son más útiles para predecir el género o el comportamiento de compra. Como alternativa, dado que el metamodelo es un modelo de regresión logística como M3, sería posible suministrar los datos de perfil directamente al modelo de segunda etapa y omitir por completo la construcción de M3.

Además de construir conjuntos combinados manualmente, como se describe aquí, existe un conjunto creciente de paquetes de R para facilitar este proceso. El paquete caretEnsemble puede ayudar con los modelos de conjunto entrenados con el paquete caret y garantizar que el muestreo de la pila se gestione correctamente para el apilamiento o la combinación. El paquete SuperLearner proporciona una forma sencilla de crear un superaprendiz; puede aplicar docenas de algoritmos base al mismo conjunto de datos y apilarlos automáticamente. Como algoritmo estándar, puede ser útil para construir un conjunto potente con el mínimo esfuerzo.

Resumen

Después de leer este documento (sus dos partes), ya deberías conocer los enfoques utilizados para ganar en competencias de minería de datos y aprendizaje automático. Los métodos de ajuste automatizado pueden ayudar a optimizar al máximo el rendimiento de un solo modelo.

Por otro lado, se pueden obtener enormes mejoras creando grupos de modelos de aprendizaje automático llamados conjuntos, que trabajan juntos para lograr un mayor rendimiento que el que los modelos individuales pueden lograr por sí solos. Diversos algoritmos basados en árboles, como los bosques aleatorios y el gradient boosting, ofrecen las ventajas de los conjuntos, pero se pueden entrenar con la misma facilidad que un solo modelo. Por otro lado, los aprendices se pueden apilar o combinar manualmente en conjuntos, lo que permite adaptar el enfoque cuidadosamente a un problema de aprendizaje.

Con tantas opciones para mejorar el rendimiento de un modelo, ¿por dónde empezar? No existe un único enfoque ideal, pero los profesionales tienden a clasificarse en tres grupos.

Primero, algunos comienzan con uno de los conjuntos más sofisticados, como bosques aleatorios o XGBoost, y dedican la mayor parte del tiempo a ajustar e implementar la ingeniería de características para lograr el máximo rendimiento posible para este modelo.

Un segundo grupo podría probar diversos enfoques y luego recopilar los modelos en un único conjunto apilado o combinado para crear un modelo de aprendizaje más potente.

El tercer enfoque podría describirse como “probar todo en la computadora y ver qué funciona”. Este enfoque intenta alimentar al algoritmo de aprendizaje con la mayor cantidad de datos posible y lo más rápido posible, y a veces se combina con ingeniería de características automatizada o técnicas de reducción de dimensionalidad, como las descritas en los temas anteriores. Con la práctica, es posible que algunas de estas ideas te resulten más atractivas que otras, así que siéntete libre de usar la que mejor se adapte a tus necesidades.

Aunque este documento se diseñó para ayudarte a preparar modelos listos para la competencia, ten en cuenta que tus competidores tienen acceso a las mismas técnicas. No podrás quedarte estancado; por lo tanto, continúa incorporando métodos propietarios a tu repertorio de trucos.

Quizás puedas aportar tu experiencia única en la materia, o quizás tus puntos fuertes incluyan un ojo para el detalle en la preparación de datos. En cualquier caso, la práctica hace al maestro, así que aprovecha las competencias para probar, evaluar y mejorar tus habilidades de aprendizaje automático.