Mejora del rendimiento del modelo (Construyendo mejores aprendices), parte I

Cuando un equipo deportivo no cumple con su objetivo - ya sea que el objetivo es obtener una medalla de oro olímpica, un campeonato de liga o un tiempo récord mundial - debe buscar posibles mejoras. Imagina que eres el entrenador del equipo. ¿Cómo pasarías tus sesiones de práctica? Tal vez dirigirías a los atletas que entrenen más duro o entrenen de manera diferente para maximizar cada parte de su potencial. O bien, puedes enfatizar un mejor trabajo en equipo, utilizando las fortalezas y debilidades de los atletas de manera más inteligente.

Ahora imagina que estás entrenando un algoritmo de aprendizaje automático de campeonato. Quizás esperas competir en competiciones de aprendizaje automático o simplemente necesitas superar a la competencia empresarial. ¿Por dónde empezar? A pesar del contexto diferente, las estrategias para mejorar el rendimiento de un equipo deportivo son similares a las que se utilizan para mejorar el rendimiento de los estudiantes de estadística.

Como entrenador, es tu trabajo encontrar la combinación de técnicas de entrenamiento y habilidades de trabajo en equipo que permitan que el proyecto de aprendizaje automático alcance tus objetivos de rendimiento.

Este documento se basa en el material tratado en este curso para presentar técnicas que mejoran la capacidad predictiva de los algoritmos de aprendizaje. Aprenderás:

- Técnicas para automatizar el ajuste del rendimiento de los modelos mediante la búsqueda sistemática del conjunto óptimo de condiciones de entrenamiento.
- Métodos para combinar modelos en grupos que utilizan el trabajo en equipo para abordar tareas de aprendizaje complejas.
- Cómo usar y diferenciar las variantes populares de árboles de decisión que se han popularizado gracias a su impresionante rendimiento.

Ninguno de estos métodos funcionará correctamente en todos los problemas. Sin embargo, al observar los proyectos ganadores de las competiciones de aprendizaje automático, probablemente encontrarás que al menos uno de ellos se ha empleado.

Para ser competitivo, tú también necesitarás incorporar estas habilidades a su repertorio.

Optimización de modelos de stock para un mejor rendimiento

Algunas tareas de aprendizaje automático se adaptan bien a los modelos de stock presentados en temas anteriores. Para estas tareas, puede que no sea necesario dedicar mucho tiempo a iterar y refinar el modelo, ya que puede funcionar bastante bien sin esfuerzo adicional. Por otro lado, muchas tareas del mundo real son inherentemente más difíciles. En estas tareas, los conceptos subyacentes que se deben aprender tienden a ser extremadamente complejos, lo que requiere la comprensión de muchas relaciones sutiles, o el problema puede verse afectado por una variabilidad aleatoria considerable, lo que dificulta encontrar la señal dentro del ruido.

Desarrollar modelos que rindan excepcionalmente bien en este tipo de problemas desafiantes es tanto un arte como una ciencia. A veces, un poco de intuición es útil para identificar áreas donde se puede mejorar el rendimiento. En otros casos, encontrar mejoras requerirá un enfoque de fuerza bruta y ensayo y error. Por supuesto, esta es una de las ventajas de usar máquinas que nunca se cansan ni se aburren. La búsqueda de numerosas mejoras potenciales puede facilitarse mediante programas automatizados.

Sin embargo, como veremos, el esfuerzo humano y el tiempo de computación no siempre son intercambiables, y crear un algoritmo de aprendizaje preciso puede conllevar sus propios costos.

En el documento, Divide y vencerás - Clasificación mediante árboles de decisión y reglas, intentamos resolver un problema complejo de aprendizaje automático al intentar predecir préstamos bancarios con probabilidad de impago. Si bien logramos una respetable precisión de clasificación del 82 %, tras un análisis más detallado en el documento, Evaluación del rendimiento del modelo, nos dimos cuenta de que el estadístico de precisión era algo engañoso. El estadístico kappa (una mejor medida del rendimiento para resultados desequilibrados) fue de tan solo 0.294, medido mediante validación cruzada (CV) de 10 pasos, lo que sugirió que el modelo tenía un rendimiento algo deficiente, a pesar de su alta precisión.

En esta sección, revisaremos el modelo de calificación crediticia para ver si podemos mejorar los resultados.

Tal vez recuerden que primero usamos un árbol de decisión C5.0 estándar para construir el clasificador de los datos crediticios y, posteriormente, intentamos mejorar su rendimiento ajustando la opción de ensayos para aumentar el número de iteraciones de refuerzo.

Al cambiar el número de iteraciones del valor predeterminado de 1 a 10, logramos aumentar la precisión del modelo. Como se define en documentos relativos al tema, estas opciones del

modelo, conocidas como hiperparámetros, no se aprenden automáticamente de los datos, sino que se configuran antes del entrenamiento. El proceso de probar diversas configuraciones de hiperparámetros para lograr un mejor ajuste del modelo se denomina **ajuste de hiperparámetros**, y las estrategias para el ajuste van desde el simple ensayo y error ad hoc hasta iteraciones más rigurosas y sistemáticas.

El ajuste de hiperparámetros no se limita a los árboles de decisión. Por ejemplo, ajustamos modelos k-NN al buscar el mejor valor de k. También ajustamos máquinas de soporte vectorial al elegir diferentes funciones kernel. La mayoría de los algoritmos de aprendizaje automático permiten ajustar al menos un hiperparámetro, y los modelos más sofisticados ofrecen diversas maneras de ajustar el modelo. Si bien esto permite que el modelo se adapte estrechamente a la tarea de aprendizaje, la complejidad de las múltiples opciones puede resultar abrumadora. Se justifica un enfoque más sistemático.

Determinación del alcance del ajuste de hiperparámetros

Al realizar el ajuste de hiperparámetros, es importante limitar el alcance para evitar que la búsqueda se prolongue indefinidamente. La computadora proporciona la fuerza, pero es el ser humano quien debe determinar dónde buscar y durante cuánto tiempo. A pesar del aumento de la potencia computacional y la reducción de los costos de la computación en la nube, la búsqueda puede fácilmente descontrolarse al examinar combinaciones de valores casi infinitas. Un alcance de ajuste estrecho o superficial puede durar lo suficiente para tomar un café, mientras que uno amplio o profundo puede dar tiempo para dormir bien por la noche, ¡o incluso más!

El tiempo y el dinero suelen ser intercambiables, ya que se puede ganar tiempo con recursos computacionales adicionales o contratando a más miembros del equipo para construir modelos más rápido o en paralelo. Aun así, dar esto por sentado puede llevar a la ruina en forma de sobrecostos o incumplimientos de plazos, ya que es fácil que el alcance se dispare rápidamente cuando el trabajo avanza por innumerables tangentes y callejones sin salida sin un plan. Para evitar estos inconvenientes, conviene planificar con antelación la amplitud y profundidad del proceso de ajuste.

Puedes empezar por considerar el ajuste como un proceso similar al clásico juego de mesa “Batalla Naval”. En este juego, tu oponente ha colocado una flota de acorazados en una cuadrícula bidimensional, oculta a tu vista. Tu objetivo es destruir la flota del oponente adivinando las coordenadas de todos sus barcos antes de que ellos hagan lo mismo con los tuyos. Dado que los barcos tienen tamaños y formas conocidos, un jugador inteligente comenzará explorando ampliamente la cuadrícula (rejilla) de búsqueda en un patrón de tablero de ajedrez, pero se centrará rápidamente en un objetivo específico una vez alcanzado.

Esta es una mejor estrategia que adivinar coordenadas al azar o iterar sistemáticamente sobre cada coordenada, dos métodos ineficientes en comparación.

Figura 1: La búsqueda de los hiperparámetros óptimos para el aprendizaje automático puede ser muy similar a jugar al clásico juego de mesa "Batalla Naval".

De igual forma, existen métodos de ajuste más eficientes que la iteración sistemática sobre un sinfín de valores y combinaciones de valores. Con la experiencia, desarrollarás una intuición sobre cómo proceder, pero para los primeros intentos, puede ser útil pensar detenidamente en el proceso.

La siguiente estrategia general, presentada en una serie de pasos, puede adaptarse a tu proyecto de aprendizaje automático, tus recursos computacionales y de personal, y tu estilo de trabajo:

1. **Replicar los criterios de evaluación del mundo real**: Para encontrar el mejor conjunto de hiperparámetros del modelo, es importante que los modelos se evalúen utilizando los mismos criterios que se utilizarán en la implementación. Esto puede implicar elegir una métrica de evaluación que refleje la métrica final del mundo real o escribir una función que simule el entorno de implementación.

2. **Considerar el uso de recursos para una iteración**: Dado que el ajuste implicará iterar varias veces el mismo algoritmo, es necesario tener una estimación del tiempo y los recursos computacionales necesarios para una sola iteración. Si se tarda una hora en entrenar un solo modelo, se necesitarán 100 horas o más para 100 iteraciones. Si la memoria de la computadora ya está al límite, es probable que se supere durante el ajuste. Si esto supone un problema, será necesario invertir en potencia computacional adicional, ejecutar el experimento en paralelo o reducir el tamaño del conjunto de datos mediante un muestreo aleatorio.

3. **Comenzar con una búsqueda superficial para detectar patrones**: El proceso de ajuste inicial debe ser interactivo y superficial. El objetivo es desarrollar tu propia

comprensión de qué opciones y valores son importantes. Al probar un solo hiperparámetro, siga aumentando o disminuyendo tu configuración en incrementos razonables hasta que el rendimiento deje de mejorar (o comience a disminuir). Dependiendo de la opción, estos pueden ser incrementos de uno, múltiplos de cinco o diez, o fracciones incrementales pequeñas, como 0.1, 0.01, 0.001, etc. Al ajustar dos o más hiperparámetros, puede ser útil centrarse en uno a la vez y mantener los demás valores estáticos. Este es un enfoque más eficiente que probar todas las combinaciones posibles de configuraciones, pero puede pasar por alto combinaciones importantes que se habrían descubierto si se probaran todas las combinaciones.

4. **Limitar al conjunto óptimo de valores de hiperparámetros**: Una vez que tengas una idea de un rango de valores que sospeches que contiene las configuraciones óptimas, puedes reducir los incrementos entre los valores probados y probar un rango más estrecho con mayor precisión o probar un mayor número de combinaciones de valores.
   El paso anterior ya debería haber generado un conjunto razonable de hiperparámetros, por lo que este paso solo debería mejorar y nunca perjudicar el rendimiento del modelo; puede detenerse en cualquier momento.

5. **Determinar un punto de parada razonable**: Decidir cuándo detener el proceso de ajuste es más fácil decirlo que hacerlo: la emoción de la búsqueda y la posibilidad de un modelo ligeramente mejor pueden generar un deseo obstinado de continuar. A veces, el punto de parada es una fecha límite del proyecto cuando el tiempo se agota. En otros casos, el trabajo solo puede detenerse una vez alcanzado el nivel de rendimiento deseado.
   En cualquier caso, dado que la única manera de garantizar la búsqueda de los valores óptimos de los hiperparámetros es probar un número infinito de posibilidades, en lugar de trabajar hasta el agotamiento, deberá definir el punto en el que el rendimiento es “suficientemente bueno” para detener el proceso.

La figura 2 ilustra el proceso de búsqueda de valores de hiperparámetros para el ajuste de un solo parámetro. Se evaluaron cinco valores potenciales (1, 2, 3, 4 y 5), representados por círculos sólidos, en la primera pasada. La precisión fue máxima cuando el hiperparámetro se estableció en 3.

Para comprobar si existía una configuración aún mejor, se probaron ocho valores adicionales (de 2.2 a 3.8 en incrementos de 0.2, representados por marcas verticales) dentro del rango de 2 a 4. Esto permitió descubrir una mayor precisión con el hiperparámetro establecido en 3.2. Si el tiempo lo permite, se podrían probar más valores en un rango más estrecho alrededor de este valor para encontrar una configuración aún mejor.

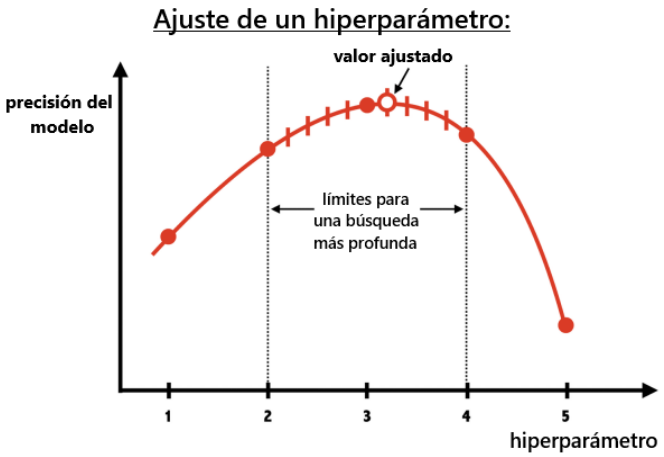

Figura 2: Estrategias para el ajuste de parámetros para encontrar el valor óptimo mediante una búsqueda amplia y luego restringida.

Ajustar dos o más hiperparámetros es más complicado, ya que el valor óptimo de un parámetro puede depender del valor de los demás. Construir una visualización como la que se muestra en la figura 3 puede ayudar a comprender cómo encontrar las mejores combinaciones de parámetros; en los puntos críticos donde ciertas combinaciones de valores resultan en un mejor rendimiento del modelo, se pueden probar más valores en rangos cada vez más estrechos:

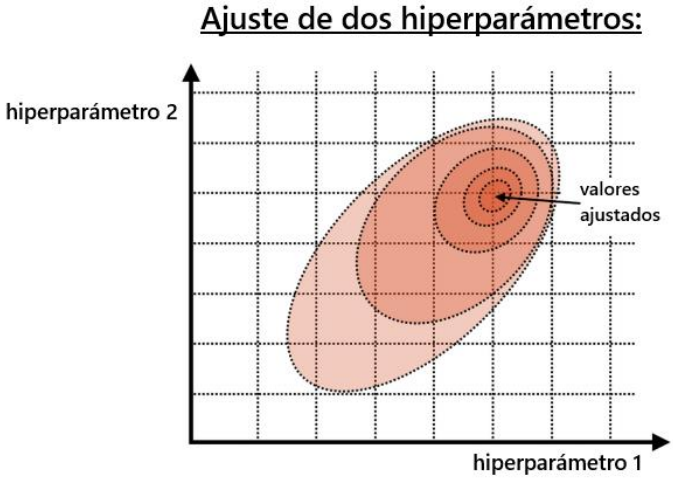

Figura 3: Las estrategias de ajuste se vuelven más complejas a medida que se añaden más hiperparámetros, ya que el mejor rendimiento del modelo depende de las combinaciones de valores.

Esta búsqueda en rejilla (***grid search***) estilo Battleship, en la que los hiperparámetros y sus combinaciones se prueban sistemáticamente, no es el único enfoque de ajuste, aunque podría ser el más utilizado. Un enfoque más inteligente, denominado optimización bayesiana, considera el proceso de ajuste como un problema de aprendizaje que puede resolverse

mediante modelado. Este enfoque se incluye en algunos programas de aprendizaje automático, pero queda fuera del alcance de este curso. En su lugar, en el resto de esta sección, nos centraremos en aplicar la idea de la búsqueda en rejilla a nuestro conjunto de datos del mundo real.

Ejemplo - uso de caret para el ajuste automático

Afortunadamente, podemos usar R para realizar la búsqueda iterativa entre múltiples valores de hiperparámetros y combinaciones de valores posibles para encontrar el mejor conjunto. Este enfoque es un método de 'fuerza bruta' relativamente sencillo, aunque a veces computacionalmente costoso, para optimizar el rendimiento de un algoritmo de aprendizaje.

El paquete caret, utilizado previamente en el tema de Evaluación del Rendimiento del Modelo, proporciona herramientas para facilitar este tipo de ajuste automático. La funcionalidad principal de ajuste la proporciona la función train(), que sirve como interfaz estandarizada para más de 200 modelos de aprendizaje automático diferentes, tanto para tareas de clasificación como de predicción numérica. Con esta función, es posible automatizar la búsqueda de modelos óptimos mediante una selección de métodos de evaluación y métricas.

El ajuste automático de parámetros con caret requiere considerar tres preguntas:

- ¿Qué tipo de algoritmo de aprendizaje automático (y su implementación específica en R) se debe entrenar con los datos?
- ¿Qué hiperparámetros se pueden ajustar para este algoritmo y con qué intensidad se deben ajustar para encontrar la configuración óptima?
- ¿Qué criterio se debe utilizar para evaluar los modelos candidatos e identificar el mejor conjunto general de valores de ajuste?

Responder a la primera pregunta implica encontrar una coincidencia entre la tarea de aprendizaje automático y uno de los muchos modelos disponibles en el paquete caret. Esto requiere una comprensión general de los tipos de modelos de aprendizaje automático, que probablemente ya tengas si has estado trabajando en los temas del curso cronológicamente. También puede ser útil realizar un proceso de eliminación.

Casi la mitad de los modelos se pueden eliminar dependiendo de si la tarea es de clasificación o predicción numérica; otros se pueden excluir según el formato de los datos de entrenamiento o la necesidad de evitar modelos de caja negra, etc. En cualquier caso, no hay razón para no crear varios modelos altamente ajustados y compararlos en todo el conjunto.

La respuesta a la segunda pregunta depende en gran medida de la elección del modelo, ya que cada algoritmo utiliza su propio conjunto de hiperparámetros. Las opciones de ajuste disponibles para los modelos predictivos que se tratan en este curso se enumeran en la siguiente tabla. Ten en cuenta que, aunque algunos modelos tienen opciones adicionales que no se muestran, solo las que aparecen en la tabla son compatibles con el ajuste automático.

Modelo	Tarea de aprendizaje	Nombre del método	Hiperparámetros
k-Nearest Neighbors	Clasificación	knn	k
Naïve Bayes	Clasificación	nb	fL, usekernel
Decision Trees	Clasificación	C5.0	model, trials, winnow
OneR Rule Learner	Clasificación	OneR	None
RIPPER Rule Learner	Clasificación	JRip	NumOpt
Linear Regression	Regresión	lm	None
Regression Trees	Regresión	rpart	cp
Model Trees	Regresión	M5	pruned, smoothed, rules
Neural Networks	Doble uso	nnet	size, decay
Support Vector Machines (Linear Kernel)	Doble uso	svmLinear	C
Support Vector Machines (Radial Basis Kernel)	Doble uso	svmRadial	C, sigma
Random Forests	Doble uso	rf	mtry
Gradient Boosting Machines (GBM)	Doble uso	gbm	n.trees, interaction.depth, shrinkage, n.minobsinnode
XGBoost (XGB)	Doble uso	xgboost	eta, max_depth, colsample_bytree, subsample, nrounds, gamma,min_child_wight

Para obtener una lista completa de los modelos y las opciones de ajuste correspondientes que cubre caret, consulta la tabla proporcionada por Max Kuhn, autor del paquete, en http://topepo.github.io/caret/available-models.html.

Si olvidas los parámetros de ajuste de un modelo en particular, puedes usar la función modelLookup() para encontrarlos. Simplemente proporciona el nombre del método, como se ilustra para el modelo C5.0:

```
> library(caret)
> modelLookup("C5.0")
```

```
  model parameter                label forReg forClass probModel
1  C5.0    trials # Boosting Iterations  FALSE     TRUE      TRUE
2  C5.0     model            Model Type  FALSE     TRUE      TRUE
3  C5.0    winnow                Winnow  FALSE     TRUE      TRUE
```

El objetivo del ajuste automático es iterar sobre el conjunto de modelos candidatos que componen la rejilla de búsqueda de posibles combinaciones de parámetros. Dado que no es

práctico buscar todas las combinaciones posibles, solo se utiliza un subconjunto de posibilidades para construir la rejilla. Por defecto, caret busca, como máximo, tres valores para cada uno de los p hiperparámetros del modelo, lo que significa que se probarán, como máximo, 3p modelos candidatos.

Por ejemplo, por defecto, el ajuste automático de los k vecinos más cercanos comparará 3^1 = 3 modelos candidatos con k = 5, k = 7 y k = 9. De forma similar, ajustar un árbol de decisión resultará en una comparación de hasta 27 modelos candidatos diferentes, que comprenden la rejilla de 3^3 = 27 combinaciones de configuraciones de modelo, ensayos y winnow. Sin embargo, en la práctica, solo se prueban 12 modelos. Esto se debe a que el modelo y el winnow solo pueden tomar dos valores (árbol versus reglas y TRUE versus FALSE, respectivamente), lo que resulta en un tamaño de rejilla de 3*2*2 = 12.

Dado que la rejilla de búsqueda predeterminada puede no ser la ideal para tu problema de aprendizaje, caret te permite proporcionar una rejilla de búsqueda personalizada definida con un simple comando, que abordaremos más adelante.

El tercer y último paso en el ajuste automático de modelos consiste en identificar el mejor modelo entre los candidatos. Esto utiliza los métodos descritos en el tema, Evaluación del Rendimiento del Modelo, incluyendo la elección de la estrategia de remuestreo (resampling) para crear conjuntos de datos de entrenamiento y prueba, y el uso de estadísticas de rendimiento del modelo para medir la precisión predictiva.

Todas las estrategias de remuestreo y muchas de las estadísticas de rendimiento que hemos aprendido son compatibles con caret. Estos incluyen estadísticas como la precisión y kappa para clasificadores, y R cuadrada o el error cuadrático medio (RMSE, ***root-mean-square-error***) para modelos numéricos. Si se desea, también se pueden utilizar medidas sensibles al costa, como la sensibilidad, la especificidad y el AUC.

Por defecto, caret seleccionará el modelo candidato con el mejor valor para la medida de rendimiento deseada. Dado que esta práctica a veces resulta en la selección de modelos que logran pequeñas mejoras de rendimiento mediante grandes aumentos en la complejidad del modelo, se proporcionan funciones de selección de modelos alternativas. Estas alternativas permiten elegir modelos más simples que se acerquen razonablemente al mejor modelo, lo cual puede ser conveniente cuando conviene sacrificar algo de rendimiento predictivo para mejorar la eficiencia computacional.

Dada la amplia variedad de opciones en el proceso de ajuste de caret, resulta útil que muchos de los valores predeterminados de la función sean razonables. Por ejemplo, sin especificar la configuración manualmente, caret utiliza la precisión de predicción o el RMSE en una muestra de bootstrap para elegir el modelo con mejor rendimiento para los modelos de

clasificación y predicción numérica, respectivamente. De igual forma, definirá automáticamente una rejilla limitada para la búsqueda. Estos valores predeterminados nos permiten comenzar con un proceso de ajuste sencillo y aprender a ajustar la función train() para diseñar una amplia variedad de experimentos a nuestra elección.

Creación de un modelo ajustado sencillo

Para ilustrar el proceso de ajuste de un modelo, comencemos observando lo que sucede al intentar ajustar el modelo de calificación crediticia utilizando la configuración predeterminada del paquete caret. La forma más sencilla de ajustar un aprendiz solo requiere especificar un tipo de modelo mediante el parámetro *method*.

Dado que ya utilizamos árboles de decisión C5.0 con el modelo de crédito, continuaremos nuestro trabajo optimizando este aprendiz. El comando train() básico para ajustar un árbol de decisión C5.0 utilizando la configuración predeterminada es el siguiente (utilizamos el dataset del ejemplo de árboles con C5):

```
> library(caret)
> credit <- read.csv("credit.csv", stringsAsFactors = TRUE)
> set.seed(300)
> m <- train(default ~ ., data = credit, method = "C5.0")
```

Primero, la función set.seed() se usa para inicializar el generador de números aleatorios de R a una posición inicial establecida. Recordarás que usamos esta función en varios temas anteriores. Al establecer el parámetro semilla (en este caso, el número arbitrario 300), los números aleatorios seguirán una secuencia predefinida. Esto permite repetir simulaciones que utilizan muestreo aleatorio con resultados idénticos, una función muy útil si comparte código o intentas replicar un resultado anterior.

A continuación, definimos un árbol como default ~ . utilizando la interfaz de fórmulas de R. Esto modela el estado de impago de un préstamo (sí o no) utilizando todas las demás características del conjunto de datos de crédito. El parámetro method="C5.0" indica a la función que utilice el algoritmo de árbol de decisión C5.0.

Después de introducir el comando anterior, dependiendo de la capacidad de tu computadora, puede haber un retraso significativo durante el proceso de ajuste. Aunque se trata de un conjunto de datos pequeño, se requiere una cantidad considerable de cálculo. R debe generar repetidamente muestras de datos de bootstrap aleatorias, construir árboles de decisión, calcular estadísticas de rendimiento y evaluar el resultado.

Dado que hay 12 modelos candidatos con diferentes valores de hiperparámetros para evaluar y 25 muestras de bootstrap por modelo candidato para calcular una medida de rendimiento promedio, se construyen 25*12 = 300 modelos de árboles de decisión con C5.0, ¡y esto sin contar los árboles de decisión adicionales que se construyen al configurar las pruebas de refuerzo!

Una lista llamada m almacena el resultado del experimento train(), y el comando str(m) mostrará los resultados asociados, pero el contenido puede ser considerable. En su lugar, simplemente escribe el nombre del objeto para obtener un resumen condensado de los resultados. Por ejemplo, escribir m produce el siguiente resultado (ten en cuenta que se han añadido etiquetas numeradas para mayor claridad):

```
> m
1000 samples
  16 predictor
   2 classes: 'no', 'yes'                                          (1)

No pre-processing
Resampling: Bootstrapped (25 reps)
Summary of sample sizes: 1000, 1000, 1000, 1000, 1000, 1000, ...  (2)
Resampling results across tuning parameters:

  model  winnow  trials  Accuracy   Kappa                          (3)
  rules  FALSE    1      0.6918852  0.2749843
  rules  FALSE   10      0.7144478  0.3112184
  rules  FALSE   20      0.7270463  0.3344054
  rules   TRUE    1      0.6947856  0.2752201
  rules   TRUE   10      0.7144334  0.3150388
  rules   TRUE   20      0.7268154  0.3393965
  tree   FALSE    1      0.6909682  0.2569822
  tree   FALSE   10      0.7256597  0.2996244
  tree   FALSE   20      0.7322016  0.3157605
  tree    TRUE    1      0.6920217  0.2604578
  tree    TRUE   10      0.7279264  0.3099733
  tree    TRUE   20      0.7299631  0.3131074

Accuracy was used to select the optimal model using the largest value.
The final values used for the model were trials = 20, model = tree and winnow = FALSE.   (4)
```

Figura 4: Los resultados de un experimento con caret se dividen en cuatro componentes, como se indica en esta figura.

Las etiquetas resaltan cuatro componentes principales en la salida:

1. **Una breve descripción del conjunto de datos de entrada**: Si estás familiarizado con tus datos y has aplicado la función train() correctamente, esta información no debería sorprenderte.
2. **Un informe de los métodos de preprocesamiento y remuestreo aplicados**: Aquí vemos que se utilizaron 25 muestras de bootstrap, cada una con 1000 ejemplos, para entrenar los modelos.
3. **Una lista de los modelos candidatos evaluados**: En esta sección, podemos confirmar que se probaron 12 modelos diferentes, basados en las combinaciones de

tres hiperparámetros C5.0: model, trials y winnow. También se muestran la precisión promedio y las estadísticas kappa de cada modelo candidato.

4. **La elección del mejor modelo**: Como se describe en la nota al pie, se seleccionó el modelo con la mayor precisión (en otras palabras, el "largest" (*mayor*)). Este fue el modelo C5.0 que utilizó un árbol de decisión con la configuración winnow = FALSE y trials = 20.

Tras identificar el mejor modelo, la función train() utiliza los hiperparámetros ajustados para construir un modelo con el conjunto de datos de entrada completo, que se almacena en m como m$finalModel. En la mayoría de los casos, no será necesario trabajar directamente con el subobjeto finalModel. En su lugar, simplemente utiliza la función predict() con el objeto m de la siguiente manera:

```
> p <- predict(m, credit)
```

El vector de predicciones resultante funciona como se esperaba, lo que nos permite crear una matriz de confusión que compara los valores predichos y reales:

```
> table(p, credit$default)
```

```
p       no yes
  no   700   2
  yes    0 298
```

De los 1000 ejemplos utilizados para entrenar el modelo final, solo dos se clasificaron erróneamente, lo que representa una precisión del 99.8 %. Sin embargo, es muy importante tener en cuenta que, dado que el modelo se construyó con los datos de entrenamiento y de prueba, esta precisión es optimista y, por lo tanto, no debe interpretarse como un indicador del rendimiento con datos no analizados. La estimación de precisión bootstrap del 72.996 %, que se encuentra en la última fila de la sección tres de la salida de train() en la figura 4, es una estimación mucho más realista de la precisión futura.

Además del ajuste automático de hiperparámetros, el uso de las funciones train() y predict() del paquete caret también ofrece dos ventajas que van más allá de las funciones incluidas en los paquetes estándar.

En primer lugar, cualquier paso de preparación de datos aplicado por la función train() se aplicará de forma similar a los datos utilizados para generar predicciones. Esto incluye transformaciones como el centrado y el escalado, así como la imputación de valores faltantes. Permitir que caret gestione la preparación de datos garantizará que los pasos que contribuyeron al mejor rendimiento del modelo se mantengan al implementarlo.

En segundo lugar, la función predict() proporciona una interfaz estandarizada para obtener valores y probabilidades de clase predichos, incluso para tipos de modelos que normalmente requerirían pasos adicionales para obtener esta información. Para un modelo de clasificación, las clases predichas se proporcionan por defecto:

```
> head(predict(m, credit))
[1] no  yes no  no  yes no
Levels: no yes
```

Para obtener las probabilidades estimadas de cada clase, utilice el parámetro type = "prob":

```
> head(predict(m, credit, type = "prob"))
```

	no	yes
1	0.9606970	0.03930299
2	0.1388444	0.86115560
3	1.0000000	0.00000000
4	0.7720279	0.22797207
5	0.2948061	0.70519387
6	0.8583715	0.14162853

Incluso en los casos en que el modelo subyacente se refiere a las probabilidades de predicción mediante una cadena diferente (por ejemplo, "raw" para un modelo naiveBayes), la función predict() traducirá type = "prob" a la configuración de parámetro adecuada automáticamente.

Personalización del proceso de ajuste

El árbol de decisión que creamos anteriormente demuestra la capacidad del paquete caret para producir un modelo optimizado con mínima intervención. La configuración predeterminada permite crear modelos optimizados fácilmente. Sin embargo, también es posible modificarla según se desee, lo que puede ayudar a alcanzar el máximo rendimiento.

Antes de comenzar el proceso de ajuste, conviene responder a una serie de preguntas que ayudarán a configurar el experimento de caret:

- ¿Cuánto tiempo tarda una iteración? En otras palabras, ¿cuánto tiempo se tarda en entrenar una sola instancia del modelo que se está ajustando?

- Dado el tiempo que se tarda en entrenar una sola instancia, ¿cuánto tiempo se tardará en realizar la evaluación del modelo utilizando el método de remuestreo elegido? Por ejemplo, un CV de 10-fold requerirá 10 veces más tiempo que entrenar un solo modelo.

- ¿Cuánto tiempo está dispuesto a dedicar al ajuste? Con base en esta cifra, se puede determinar el número total de valores de hiperparámetros que se pueden probar. Por ejemplo, si se tarda un minuto en evaluar un modelo utilizando un CV de 10-fold, se pueden probar 60 configuraciones de hiperparámetros por hora.

Usar el tiempo como factor limitante clave ayudará a limitar el proceso de ajuste y evitará que se busque constantemente un rendimiento cada vez mejor.

Una vez que haya decidido cuánto tiempo dedicar a las pruebas, es fácil personalizar el proceso a tu gusto. Para ilustrar esta flexibilidad, modifiquemos nuestro trabajo en el árbol de decisión de crédito para reflejar el proceso que utilizamos en el documento, Evaluación del Rendimiento del Modelo.

En ese tema, estimamos el estadístico kappa utilizando un CV de 10-fold. Haremos lo mismo aquí, utilizando kappa para ajustar las pruebas de refuerzo para el algoritmo del árbol de decisión C5.0 y encontrar la configuración óptima para nuestros datos. Ten en cuenta que el boosting del árbol de decisión se trató por primera vez en el tema, Divide y vencerás: Clasificación mediante árboles de decisión y reglas, y también se tratará con mayor detalle más adelante en este documento.

La función trainControl() se utiliza para crear un conjunto de opciones de configuración conocido como objeto de control. Este objeto guía la función train() y permite la selección de criterios de evaluación del modelo, como la estrategia de remuestreo y la medida utilizada para elegir el mejor modelo.

Aunque esta función puede usarse para modificar prácticamente cualquier aspecto de un experimento de ajuste de caret, nos centraremos en dos parámetros importantes: method y SelectionFunction.

Si deseas obtener más detalles sobre el objeto de control, puedes usar el comando ?trainControl para obtener una lista de todos los parámetros.

Al usar la función trainControl(), el parámetro método establece el método de remuestreo, como el muestreo de retención (holdout) o el -kfold CV. La siguiente tabla enumera los posibles valores del método, así como los parámetros adicionales para ajustar el tamaño de la muestra y el número de iteraciones.

Aunque las opciones predeterminadas para estos métodos de remuestreo siguen las convenciones habituales, puedes ajustarlas según el tamaño de tu conjunto de datos y la complejidad de su modelo.

Método de remuestreo	Nombre del método	Opciones adicionales y valores predeterminados
Holdout sampling	LGOCV	p = 0.75 (proporción de datos de entrenamiento)
k-fold CV	cv	number = 10 (número de pliegues (*folds*))
Repeated k-fold CV	repeatedcv	number = 10 (número de pliegues (*folds*)) repeats = 10 (número de iteraciones)
Bootstrap sampling	boot	number = 25 (iteraciones de remuestreo)
0.632 bootstrap	boot632	number = 25 (iteraciones de remuestreo)
Leave-one-out CV	LOOCV	ninguna

El parámetro selectionFunction se utiliza para especificar la función que seleccionará el modelo óptimo entre los candidatos. Se incluyen tres de estas funciones. La función best simplemente selecciona el candidato con el mejor valor en la medida de rendimiento especificada. Esta función se utiliza por defecto. Las otras dos funciones se utilizan para seleccionar el modelo más simple o parsimonioso que se encuentre dentro de un umbral determinado de rendimiento del mejor modelo. La función oneSE selecciona el candidato más simple con un margen de error estándar del mejor rendimiento, y la función de tolerancia utiliza el candidato más simple dentro de un porcentaje especificado por el usuario.

La clasificación de modelos por simplicidad del paquete caret implica cierta subjetividad. Para obtener información sobre cómo se clasifican los modelos, consulta la página de ayuda de las funciones de selección escribiendo ?best en el símbolo del sistema de R.

Para crear un objeto de control llamado ctrl que utilice un CV de 10-fold y la función de selección oneSE, utiliza el siguiente comando; ten en cuenta que number = 10 se incluye solo para mayor claridad. Dado que este es el valor predeterminado para method="cv", podría haberse omitido:

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

Usaremos el resultado de esta función en breve.

Mientras tanto, el siguiente paso para configurar nuestro experimento es crear la rejilla de búsqueda para el ajuste de hiperparámetros. La rejilla debe incluir una columna con el nombre de cada hiperparámetro del modelo deseado, independientemente de si se ajustará.

También debes incluir una fila para cada combinación de valores que se desee probar. Dado que utilizamos un árbol de decisión C5.0, necesitaremos columnas con los nombres model, trials y winnow, correspondientes a las tres opciones que se pueden ajustar. Para otros modelos de aprendizaje automático, consulta la tabla presentada anteriormente en este documento o utiliza la función modelLookup() para encontrar los hiperparámetros como se describió anteriormente.

En lugar de llenar el frame de datos de la rejilla celda por celda (una tarea tediosa si hay muchas combinaciones posibles de valores), podemos utilizar la función expand.grid(), que crea frames de datos a partir de las combinaciones de todos los valores proporcionados. Por ejemplo, supongamos que queremos mantener constantes model = "tree" y winnow = FALSE mientras buscamos ocho valores diferentes de ensayos.

Esto se puede crear como:

```
> grid <- expand.grid(model = "tree",
+   trials = c(1, 5, 10, 15, 20, 25, 30, 35),
+   winnow = FALSE)
```

El frame de datos de la cuadrícula resultante contiene 1*8*1 = 8 filas:

```
> grid
```

```
  model trials winnow
1  tree      1  FALSE
2  tree      5  FALSE
3  tree     10  FALSE
4  tree     15  FALSE
5  tree     20  FALSE
6  tree     25  FALSE
7  tree     30  FALSE
8  tree     35  FALSE
```

La función train() generará un modelo candidato para la evaluación utilizando la combinación de parámetros del modelo de cada fila de la rejilla.

Dadas la rejilla de búsqueda y el objeto de control creado previamente, estamos listos para ejecutar un experimento train() completamente personalizado. Como antes, estableceremos la semilla aleatoria en el número arbitrario 300 para garantizar resultados repetibles. Pero esta vez, pasaremos nuestro objeto de control y la rejilla de ajuste, añadiendo el parámetro metric = "Kappa", que indica la estadística que utilizará la función de evaluación del modelo; en este caso, "oneSE". El conjunto completo de comandos es el siguiente:

```
> set.seed(300)
> m <- train(default ~ ., data = credit, method = "C5.0",
+   metric = "Kappa",
+   trControl = ctrl,
+   tuneGrid = grid)
```

Esto genera un objeto que podemos ver escribiendo su nombre:

> m

```
C5.0

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:

  trials  Accuracy  Kappa
   1      0.710     0.2859380
   5      0.726     0.3256082
  10      0.725     0.3054657
  15      0.726     0.3204938
  20      0.733     0.3292403
  25      0.732     0.3308708
  30      0.733     0.3298968
  35      0.738     0.3449912

Tuning parameter 'model' was held constant at a value of tree
Tuning parameter 'winnow' was held constant at a
 value of FALSE
Kappa was used to select the optimal model using  the one SE rule.
The final values used for the model were trials = 5, model = tree and winnow = FALSE.
```

Aunque el resultado es similar al del modelo ajustado automáticamente, existen algunas diferencias notables. Debido a que se utilizó un CV de 10-fold, el tamaño de la muestra para construir cada modelo candidato se redujo a 900 en lugar de los 1000 utilizados en el bootstrap. Además, se probaron ocho modelos candidatos en lugar de los 12 del experimento anterior. Finalmente, dado que el modelo y el winnow se mantuvieron constantes, sus valores ya no se muestran en los resultados; en su lugar, se listan como nota al pie.

El mejor modelo aquí difiere bastante del experimento anterior. Antes, el mejor modelo usaba trials = 20, mientras que aquí, usaba trials = 1. Este cambio se debe a que usamos la función oneSE en lugar de la mejor función para seleccionar el modelo óptimo. Si bien el modelo con trials = 35 obtuvo el mejor kappa, el modelo de un solo ensayo ofrece un rendimiento razonablemente cercano con un algoritmo mucho más simple.

Debido a la gran cantidad de parámetros de configuración, caret puede resultar abrumador al principio. No dejes que esto te desanime: no hay una manera más sencilla de probar el rendimiento de los modelos utilizando CV de 10-fold.

En su lugar, piensa en el experimento como definido por dos partes: un objeto trainControl() que dicta los criterios de prueba y una rejilla de ajuste que determina qué parámetros del modelo evaluar. Suminístralos a la función train() y, con un poco de tiempo de cálculo, ¡tu experimento estará completo!

Por supuesto, el ajuste es solo una posibilidad para desarrollar mejores aprendices. En la siguiente sección, descubrirás que, además de potenciar un solo aprendiz para fortalecerlo, también es posible combinar varios modelos más débiles para formar un equipo más potente.

Mejora del rendimiento del modelo con conjuntos

Así como los mejores equipos deportivos tienen jugadores con habilidades complementarias en lugar de superpuestas, algunos de los mejores algoritmos de aprendizaje automático utilizan equipos de modelos complementarios. Dado que un modelo aporta un sesgo único a una tarea de aprendizaje, puede aprender fácilmente un subconjunto de ejemplos, pero tener problemas con otro. Por lo tanto, al utilizar inteligentemente el talento de varios miembros diversos del equipo, es posible crear un equipo sólido con múltiples aprendices débiles.

Esta técnica de combinar y gestionar las predicciones de múltiples modelos se enmarca en un conjunto más amplio de **métodos de meta-aprendizaje**, que consisten en aprender a aprender. Esto incluye desde algoritmos sencillos que mejoran gradualmente el rendimiento iterando sobre las decisiones de diseño (por ejemplo, el ajuste automático de parámetros utilizado anteriormente en este documento) hasta algoritmos altamente complejos que utilizan conceptos tomados de la biología evolutiva y la genética para auto modificarse y adaptarse a las tareas de aprendizaje.

Supongamos que participas en un concurso de televisión que te permite elegir un panel de cinco amigos para que te ayuden a responder la pregunta final del premio de un millón de dólares. La mayoría de la gente intentaría conformar el panel con un grupo diverso de expertos en la materia. Un panel compuesto por profesores de literatura, ciencias, historia y arte, junto con un experto actual en cultura pop, sería, sin duda, muy completo. Dada tu amplitud de conocimientos, sería improbable que una pregunta dejara perplejo al grupo.

El enfoque de meta-aprendizaje que utiliza un principio similar al de crear un equipo diverso de expertos se conoce como **conjunto**. En el resto de este documento, nos centraremos únicamente en el meta-aprendizaje en lo que respecta al conjunto - la tarea de modelar una relación entre las predicciones de varios modelos y el resultado deseado. Los métodos basados en el trabajo en equipo que se tratan aquí son muy eficaces y se utilizan a menudo para crear clasificadores más eficaces.

Entendiendo el aprendizaje por conjuntos

Todos los métodos de conjunto se basan en la idea de que al combinar varios aprendices menos eficaces, se crea uno más fuerte. Los conjuntos contienen dos o más modelos de aprendizaje automático, que pueden ser del mismo tipo, como varios árboles de decisión, o de diferentes tipos, como un árbol de decisión y una red neuronal. Si bien existen

innumerables maneras de construir un conjunto, estas tienden a clasificarse en varias categorías generales, que se pueden distinguir, en gran medida, por las respuestas a dos preguntas:

- ¿Cómo se eligen y entrenan los modelos del conjunto?
- ¿Cómo se combinan las predicciones de los modelos para obtener una única predicción final?

Para responder a estas preguntas, puede ser útil imaginar el conjunto en términos del siguiente diagrama de proceso, que abarca casi todos los enfoques de conjunto:

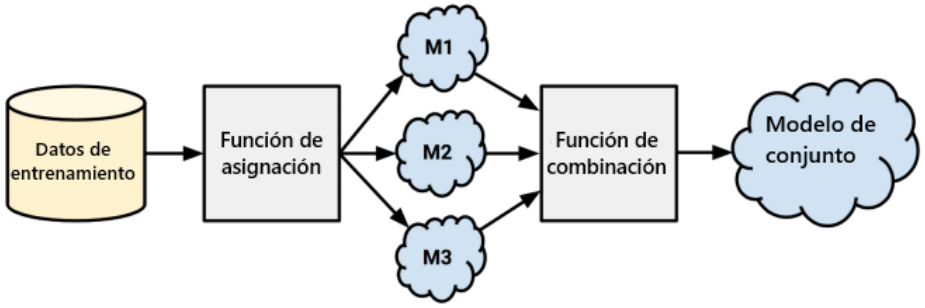

Figura 5: Los conjuntos combinan múltiples modelos más débiles en un único modelo más fuerte.

En este patrón de diseño, los datos de entrenamiento de entrada se utilizan para construir varios modelos. La función de asignación determina la cantidad y los subconjuntos de datos de entrenamiento que recibe cada modelo. ¿Recibe cada uno el conjunto completo de datos de entrenamiento o solo una muestra? ¿Recibe cada uno todas las características o un subconjunto de ellas? Las decisiones que se tomen aquí determinarán el entrenamiento de los aprendices más débiles que componen el conjunto más fuerte.

Así como desearía que diversos expertos asesoraran tu participación en un concurso televisivo de preguntas y respuestas, los conjuntos dependen de un conjunto **diverso** de clasificadores, lo que significa que tienen clasificaciones no correlacionadas, pero aun así funcionan mejor que el azar. En otras palabras, cada clasificador debe realizar una predicción independiente, pero también debe ir más allá de las simples suposiciones.

Se puede añadir diversidad al conjunto mediante la inclusión de diversas técnicas de aprendizaje automático, como un conjunto que agrupa un árbol de decisión, una red neuronal y un modelo de regresión logística.

Como alternativa, la función de asignación en sí misma también puede ser una fuente de diversidad al actuar como **manipulador de datos** y variar artificialmente los datos de entrada

para sesgar a los aprendices resultantes, incluso si utilizan el mismo algoritmo de aprendizaje. Como veremos en la práctica más adelante, los procesos de asignación y manipulación de datos pueden automatizarse o incluirse como parte del propio algoritmo de ensamblaje, o pueden realizarse manualmente como parte del proceso de ingeniería de datos y construcción de modelos.

En general, los modos de aumentar la diversidad del ensamblaje se clasifican generalmente en cinco categorías:

- Utilizar diversos algoritmos de aprendizaje base
- Manipular la muestra de entrenamiento tomando diferentes muestras aleatoriamente, a menudo mediante bootstrapping
- Manipular un único algoritmo de aprendizaje utilizando diferentes configuraciones de hiperparámetros
- Cambiar la forma en que se representa la característica objetivo, como representar un resultado como binario, categórico o numérico
- Dividir los datos de entrenamiento en subgrupos que representan diferentes patrones a aprender; Por ejemplo, se podrían estratificar los ejemplos por características clave y permitir que los modelos del conjunto se conviertan en expertos en diferentes subconjuntos de los datos de entrenamiento.

Por ejemplo, en un conjunto de árboles de decisión, la función de asignación podría usar muestreo bootstrap para construir conjuntos de datos de entrenamiento únicos para cada árbol, o podría pasar a cada uno un subconjunto diferente de características. Por otro lado, si el conjunto ya incluye un conjunto diverso de algoritmos (como una red neuronal, un árbol de decisión y un clasificador k-NN), la función de asignación podría pasar los datos de entrenamiento a cada algoritmo prácticamente sin cambios.

Una vez entrenados los modelos del conjunto, pueden usarse para generar predicciones sobre datos futuros, pero este conjunto de múltiples predicciones debe conciliarse de alguna manera para generar una única predicción final. La **función de combinación** es el paso del proceso de ensamblaje que toma cada una de estas predicciones y las combina en una única predicción fiable para el conjunto.

Por supuesto, dado que algunos modelos pueden discrepar en el valor predicho, la función debe combinar o unificar de alguna manera la información de los aprendices. La función de combinación también se conoce como **compositor** debido a su función de sintetizar la predicción final. Existen dos estrategias principales para fusionar o componer predicciones finales. La más sencilla de las dos implica métodos de ponderación, que asignan una puntuación a cada predicción que determina su peso en la predicción final. Estos métodos

van desde una votación por mayoría simple, en la que cada clasificador recibe la misma ponderación, hasta métodos más complejos basados en el rendimiento, que otorgan mayor autoridad a algunos modelos que a otros si han demostrado ser más fiables con datos históricos.

El segundo enfoque utiliza métodos de meta-aprendizaje más complejos, como la técnica de apilamiento de modelos, que se abordará en profundidad más adelante en este documento. Estos utilizan el conjunto inicial de predicciones de los estudiantes con dificultades para entrenar un algoritmo secundario de aprendizaje automático y realizar la predicción final; un proceso similar al de un comité que formula recomendaciones a un líder que toma la decisión final.

Los métodos de ensamble se utilizan para obtener un mejor rendimiento que el que se puede obtener con un solo algoritmo de aprendizaje; el objetivo principal del ensamble es convertir a un grupo de estudiantes con dificultades en un equipo más fuerte y unificado. Sin embargo, existen muchos beneficios adicionales, algunos de los cuales pueden resultar sorprendentes.

Estos sugieren razones adicionales por las que se podría recurrir a un ensamble, incluso fuera de un entorno competitivo de aprendizaje automático:

- **El uso de ensambles independientes permite el trabajo en paralelo**: Entrenar clasificadores independientes por separado significa que el trabajo puede dividirse entre varias personas. Esto permite una iteración más rápida y puede aumentar la creatividad. Cada miembro del equipo construye su mejor modelo y los resultados pueden combinarse fácilmente en un ensamble al final.
- **Rendimiento mejorado en conjuntos de datos masivos o minúsculos**: Muchos algoritmos alcanzan límites de memoria o complejidad al utilizar un conjunto extremadamente grande de características o ejemplos.
  Un conjunto de modelos independientes puede alimentarse con subconjuntos de características o ejemplos, que son más eficientes computacionalmente para entrenar que un solo modelo completo y, lo que es más importante, a menudo pueden ejecutarse en paralelo utilizando métodos de computación distribuida. Por otro lado, los conjuntos también funcionan bien con los conjuntos de datos más pequeños, ya que los métodos de remuestreo, como el bootstrapping, son parte inherente de la función de asignación de muchos diseños de conjuntos.
- **Capacidad para sintetizar datos de distintos dominios**: Dado que no existe un algoritmo de aprendizaje universal, y cada algoritmo de aprendizaje tiene sus propios sesgos y heurísticas, la capacidad del conjunto para incorporar evidencia de múltiples tipos de aprendices es cada vez más importante para modelar las tareas de aprendizaje más desafiantes, basadas en datos extraídos de diversos dominios.

- **Una comprensión más matizada de las tareas de aprendizaje difíciles**: Los fenómenos del mundo real suelen ser extremadamente complejos, con muchas complejidades que interactúan. Métodos como los conjuntos, que dividen la tarea en porciones modeladas más pequeñas, son más capaces de capturar patrones sutiles que un solo modelo podría pasar por alto. Algunos aprendices del conjunto pueden profundizar y acotar el aprendizaje para un subconjunto específico de los casos más complejos.

Ninguno de estos beneficios sería muy útil si no se pudieran aplicar fácilmente métodos de conjunto en R, y existen muchos paquetes disponibles para ello. Analicemos varios de los métodos de conjunto más populares y cómo se pueden utilizar para mejorar el rendimiento del modelo de crédito en el que hemos estado trabajando.

Algoritmos populares basados en conjuntos

Afortunadamente, utilizar equipos de aprendizaje automático para mejorar el rendimiento predictivo no implica tener que entrenar manualmente a cada miembro del conjunto por separado, aunque esta opción existe, como se aprenderá más adelante en este documento. En su lugar, existen algoritmos basados en conjuntos que manipulan la función de asignación para entrenar automáticamente una gran cantidad de modelos más simples en un solo paso.

De esta manera, un conjunto que incluya cien aprendices o más puede entrenarse sin mayor inversión de tiempo ni intervención humana que entrenar a un solo aprendiz. Tan fácil como construir un único modelo de árbol de decisión, es posible construir un conjunto con cientos de estos árboles y aprovechar el poder del trabajo en equipo. Aunque sería tentador suponer que se trata de una solución mágica, dicho poder, por supuesto, conlleva desventajas como la pérdida de interpretabilidad y un conjunto menos diverso de algoritmos base entre los que elegir. Esto se hará evidente en las secciones siguientes, que abarcan la evolución de dos décadas de algoritmos de ensamblaje populares, todos los cuales, no por casualidad, se basan en árboles de decisión.

Bagging

Uno de los primeros métodos de conjunto en obtener una amplia aceptación utilizó una técnica denominada **agregación bootstrap** o **bagging**. Como lo describió Leo Breiman a mediados de la década de 1990, el bagging comienza generando varios conjuntos de datos de entrenamiento nuevos mediante muestreo bootstrap sobre los datos de entrenamiento originales. Estos conjuntos de datos se utilizan para generar un conjunto de modelos con un único algoritmo de aprendizaje. Las predicciones de los modelos se combinan mediante votación para la clasificación y promediación para la predicción numérica.

Para obtener más información sobre bagging, consulta: Bagging predictors. Breiman L., Machine Learning, 1996, vol. 24, pp. 123-140.

Aunque el bagging es un conjunto relativamente simple, puede funcionar bastante bien si se utiliza con aprendices relativamente **inestables**, es decir, aquellos que generan modelos que tienden a cambiar sustancialmente cuando los datos de entrada cambian solo ligeramente. Los modelos inestables son esenciales para garantizar la diversidad del conjunto a pesar de las pequeñas variaciones entre los conjuntos de datos de entrenamiento bootstrap.

Por esta razón, el bagging se utiliza con mayor frecuencia con árboles de decisión, que tienden a variar drásticamente ante pequeños cambios en los datos de entrada.

El paquete ipred ofrece una implementación clásica de árboles de decisión bagging. Para entrenar el modelo, la función bagging() funciona de forma similar a muchos de los modelos utilizados anteriormente. El parámetro nbagg se utiliza para controlar el número de árboles de decisión que votan en el conjunto, con un valor predeterminado de 25.

Dependiendo de la dificultad de la tarea de aprendizaje y la cantidad de datos de entrenamiento, aumentar este número puede mejorar el rendimiento del modelo, hasta cierto punto. La desventaja es que esto genera un gasto computacional adicional y un gran número de árboles puede tardar algún tiempo en entrenarse.

Después de instalar el paquete ipred, podemos crear el conjunto de la siguiente manera. Nos ceñiremos al valor predeterminado de 25 árboles de decisión:

```
> library(ipred)
> credit <- read.csv("credit.csv", stringsAsFactors = TRUE)
> set.seed(123)
> mybag <- bagging(default ~ ., data = credit, nbagg = 25)
```

El modelo mybag resultante funciona como se esperaba en conjunto con la función predict():

```
> credit_pred <- predict(mybag, credit)
> table(credit_pred, credit$default)

credit_pred  no yes
        no  700   2
        yes   0 298
```

Dados los resultados anteriores, el modelo parece haberse ajustado a los datos extremadamente bien; probablemente demasiado bien, ya que los resultados se basan únicamente en los datos de entrenamiento y, por lo tanto, podrían reflejar un sobreajuste en

lugar de un rendimiento real con datos futuros no observados. Para obtener una mejor estimación del rendimiento futuro, podemos utilizar el método de árbol de decisión bagged del paquete caret para obtener una estimación de CV de 10-fold la precisión y el kappa. Ten en cuenta que el nombre del método para la función de bagging ipred es treebag:

```
> library(caret)
> credit <- read.csv("credit.csv")
> set.seed(300)
> ctrl <- trainControl(method = "cv", number = 10)
> train(default ~ ., data = credit, method = "treebag",
+   trControl = ctrl)

Bagged CART

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:

  Accuracy  Kappa
  0.732     0.3319334
```

El estadístico kappa de 0.33 para este modelo sugiere que el modelo de árbol bagged funciona aproximadamente tan bien como el árbol de decisión C5.0 que ajustamos anteriormente en este documento, cuyo estadístico kappa oscilaba entre 0.32 y 0.34, dependiendo de los parámetros de ajuste. Ten presente este rendimiento al leer la siguiente sección y considera las diferencias entre la técnica simple de bagging y los métodos más complejos que la desarrollan.

Boosting

Otro método común basado en conjuntos se denomina **boosting**, ya que mejora o "boosts" el rendimiento de los aprendices con dificultades para alcanzar el rendimiento de los estudiantes con mayor rendimiento. Este método se basa principalmente en el trabajo de Robert Schapire y Yoav Freund, quienes han publicado extensamente sobre el tema desde la década de 1990.

Para obtener más información sobre boosting, consulta: Boosting: Foundations and Algorithms, Schapire, RE, Freund, Y, Cambridge, MA: The MIT Press, 2012.

Al igual que el bagging, el boosting utiliza conjuntos de modelos entrenados con datos remuestreados y una votación para determinar la predicción final. Existen dos distinciones clave. En primer lugar, los conjuntos de datos remuestreados en el boosting se construyen

específicamente para generar aprendices complementarios. Esto significa que el trabajo no puede realizarse en paralelo, ya que los modelos del conjunto ya no son independientes entre sí.

En segundo lugar, en lugar de otorgar a cada aprendiz el mismo voto, el boosting otorga a cada aprendiz un voto ponderado en función de su rendimiento anterior. Los modelos con mejor rendimiento tienen mayor influencia en la predicción final del conjunto.

El boosting dará como resultado un rendimiento que, a menudo, es ligeramente mejor y, sin duda, no peor que el del mejor modelo del conjunto. Dado que los modelos del conjunto se construyen deliberadamente para ser complementarios, es posible aumentar el rendimiento del conjunto hasta un umbral arbitrario simplemente añadiendo clasificadores adicionales al grupo, asumiendo que cada clasificador adicional tiene un rendimiento superior al azar. Dada la evidente utilidad de este hallazgo, se considera que el boosting es uno de los descubrimientos más significativos en el aprendizaje automático.

Aunque el boosting puede crear un modelo con una tasa de error arbitrariamente baja, esto no siempre es razonable en la práctica. Una razón es que las mejoras de rendimiento son cada vez menores a medida que se obtienen más aprendices, lo que hace que algunos umbrales sean prácticamente inviables. Además, la búsqueda de la precisión pura puede provocar que el modelo se sobreajuste a los datos de entrenamiento y no sea generalizable a datos no vistos.

Freund y Schapire propusieron en 1997 un algoritmo de boosting llamado **AdaBoost** (abreviatura de **boosting adaptativo**). Este algoritmo se basa en la idea de generar aprendices débiles que aprenden iterativamente una mayor proporción de los ejemplos difíciles de clasificar en los datos de entrenamiento, prestando más atención (es decir, dando más peso) a los ejemplos que suelen clasificarse erróneamente.

A partir de un conjunto de datos no ponderado, el primer clasificador intenta modelar el resultado. Los ejemplos que el clasificador predijo correctamente tendrán menos probabilidades de aparecer en el conjunto de datos de entrenamiento del siguiente clasificador y, a la inversa, los ejemplos difíciles de clasificar aparecerán con mayor frecuencia.

A medida que se añaden rondas adicionales de aprendices débiles, se entrenan con datos de ejemplos cada vez más difíciles. El proceso continúa hasta que se alcanza la tasa de error general deseada o el rendimiento deja de mejorar. En ese momento, el voto de cada clasificador se pondera según su precisión en los datos de entrenamiento con los que se construyó.

Aunque los principios de boosting se pueden aplicar a casi cualquier tipo de modelo, se utilizan con mayor frecuencia con árboles de decisión. Ya aplicamos la técnica de boosting anteriormente en este documento, así como en el tema, Divide y vencerás: Clasificación mediante árboles de decisión y reglas, como método para mejorar el rendimiento de un árbol de decisión C5.0. Con C5.0, el boosting se puede habilitar simplemente estableciendo un parámetro de ensayos en un valor entero mayor que uno.

El algoritmo **AdaBoost.M1** proporciona una implementación independiente de AdaBoost para la clasificación con árboles. El algoritmo se encuentra en el paquete adabag.

Para más información sobre el paquete adabag, consulta: adabag: An R Package for Classification with Boosting and Bagging, Alfaro, E., Gamez, M., Garcia, N., Journal of Statistical Software, 2013, Vol. 54, pp. 1-35.

Creemos un clasificador AdaBoost.M1 para los datos crediticios. La sintaxis general de este algoritmo es similar a la de otras técnicas de modelado:

```
> library(adabag)
> credit <- read.csv("credit.csv", stringsAsFactors = TRUE)
> set.seed(300)
> m_adaboost <- boosting(default ~ ., data = credit)
```

Como es habitual, la función predict() se aplica al objeto resultante para realizar predicciones:

```
> p_adaboost <- predict(m_adaboost, credit)
```

A diferencia de lo habitual, en lugar de devolver un vector de predicciones, esto devuelve un objeto con información sobre el modelo. Las predicciones se almacenan en un sub-objeto llamado clase (class):

```
> head(p_adaboost$class)
[1] "no"  "yes" "no"  "no"  "yes" "no"
```

En el sub-objeto de confusión se encuentra una matriz de confusión:

```
> p_adaboost$confusion
```

```
               Observed Class
Predicted Class  no yes
            no  700   0
            yes   0 300
```

Antes de hacerse ilusiones sobre la precisión perfecta, ten en cuenta que la matriz de confusión anterior se basa en el rendimiento del modelo con los datos de entrenamiento.

Dado que el boosting permite reducir la tasa de error a un nivel arbitrario, el aprendiz simplemente continuó hasta que no cometió más errores.
Esto probablemente resultó en un sobreajuste en el conjunto de datos de entrenamiento.

Para una evaluación más precisa del rendimiento con datos no vistos, necesitamos usar otro método de evaluación. El paquete adabag proporciona una función sencilla para usar el CV de 10-fold:

```
> set.seed(300)
> adaboost_cv <- boosting.cv(default ~ ., data = credit)
```

Dependiendo de la capacidad de tu computadora, esto puede tardar un tiempo en ejecutarse, durante el cual registrará cada iteración en la pantalla; en una MacBook Pro reciente, tardó aproximadamente un minuto. Una vez completado, podemos ver una matriz de confusión más razonable:

```
> adaboost_cv$confusion
                Observed Class
Predicted Class  no yes
            no  598 160
            yes 102 140
```

Podemos encontrar el estadístico kappa usando el paquete vcd, como se muestra en el tema, Evaluación del Rendimiento del Modelo:

```
> library(vcd)
Loading required package: grid
> Kappa(adaboost_cv$confusion)
             value     ASE     z  Pr(>|z|)
Unweighted 0.3397 0.03255 10.44 1.676e-25
Weighted   0.3397 0.03255 10.44 1.676e-25
```

Con un kappa de 0.3397, el modelo boosted supera ligeramente el rendimiento de los árboles de decisión bagged, que tenían un kappa de alrededor de 0.3319. Veamos cómo se compara el boosting con otro método de conjunto.

Ten en cuenta que los resultados anteriores se obtuvieron con la versión R 4.2.3 en una PC con Windows y se verificaron en Mac OS. Al momento de escribir esto, se obtuvieron resultados ligeramente diferentes con R 4.3.3 para una Apple de silicio en una MacBook Pro

reciente. También ten en cuenta que el algoritmo AdaBoost.M1 se puede ajustar con caret especificando method = "AdaBoost.M1".

Bosques aleatorios

Otro método basado en conjuntos de árboles, llamado **bosques aleatorios**, se basa en los principios del bagging, pero añade diversidad a los árboles de decisión al permitir que el algoritmo solo elija entre un subconjunto de características seleccionado aleatoriamente cada vez que intenta dividir. Comenzando en el nodo raíz, el algoritmo de bosque aleatorio podría solo elegir entre un pequeño número de características seleccionadas aleatoriamente del conjunto completo de predictores; en cada división posterior, se proporciona un subconjunto aleatorio diferente. Al igual que en el caso del bagging, una vez generado el conjunto de árboles (el bosque), el algoritmo realiza una votación simple para realizar la predicción final.

Para más detalles sobre cómo se construyen los bosques aleatorios, consulta: Random Forests, Breiman L, Machine Learning, 2001, vol. 45, pp. 5-32. Cabe destacar que la frase "bosques aleatorios" es una marca registrada de Breiman y Cutler, pero se usa coloquialmente para referirse a cualquier tipo de conjunto de árboles de decisión.

El hecho de que cada árbol se construya a partir de conjuntos de características diferentes y seleccionados aleatoriamente ayuda a garantizar que cada árbol del conjunto sea único. Incluso es posible que dos árboles del bosque se hayan construido a partir de conjuntos de características completamente diferentes. La selección aleatoria de características impide que la heurística voraz del árbol de decisión seleccione las mismas opciones fáciles cada vez que se cultiva el árbol, lo que puede ayudar al algoritmo a descubrir patrones sutiles que el método estándar de cultivo de árboles podría pasar por alto. Por otro lado, el potencial de sobreajuste es limitado, dado que cada árbol tiene solo un voto de muchos en el bosque.

Dadas estas fortalezas, no sorprende que el algoritmo de bosque aleatorio se haya convertido rápidamente en uno de los algoritmos de aprendizaje más populares. Solo recientemente, su popularidad ha sido superada por un nuevo método de conjunto, que conocerás en breve.
Los bosques aleatorios combinan versatilidad y potencia en un único enfoque de aprendizaje automático y no son especialmente propensos al sobreajuste ni al subajuste. Dado que el algoritmo de crecimiento de árboles utiliza solo una pequeña porción aleatoria del conjunto completo de características, los bosques aleatorios pueden gestionar conjuntos de datos extremadamente grandes, donde la llamada maldición de la dimensionalidad podría provocar el fallo de otros modelos. Al mismo tiempo, su rendimiento predictivo en la mayoría de las tareas de aprendizaje es tan bueno, si no mejor, que el de todos los métodos, salvo los más sofisticados. La siguiente tabla resume las fortalezas y debilidades de los modelos de bosques aleatorios:

Fortalezas	Debilidades
• Un modelo multipropósito con un buen rendimiento en la mayoría de los problemas, incluyendo clasificación y predicción numérica. • Puede manejar datos ruidosos o faltantes, así como características categóricas o continuas. • Selecciona solo las características más importantes. • Puede utilizarse con datos con un gran número de características o ejemplos.	• A diferencia de un árbol de decisión, el modelo no es fácilmente interpretable. • Puede presentar dificultades con características categóricas con un gran número de niveles. • No se puede ajustar exhaustivamente si se desea un mayor rendimiento.

Su excelente rendimiento, combinado con su facilidad de uso, convierte a los bosques aleatorios en un excelente punto de partida para la mayoría de los proyectos de aprendizaje automático del mundo real. El algoritmo también proporciona un punto de referencia sólido para otras comparaciones con modelos altamente ajustados, así como con otros enfoques más complejos que aprenderás más adelante.

Para una demostración práctica de los bosques aleatorios, aplicaremos la técnica a los datos de calificación crediticia que hemos utilizado en este documento. Aunque existen varios paquetes con implementaciones de bosques aleatorios en R, el paquete randomForest, con su nombre acertado, es quizás el más sencillo, mientras que el paquete ranger ofrece un rendimiento mucho mejor en conjuntos de datos grandes. Ambos son compatibles con el paquete caret para la experimentación y el ajuste automático de parámetros. La sintaxis para entrenar un modelo con randomForest es la siguiente:

Random forest syntax

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

Building the classifier:

```
m <- randomForest(train, class, ntree = 500, mtry = sqrt(p))
```

- `train` is a data frame containing training data
- `class` is a factor vector with the class for each row in the training data
- `ntree` is an integer specifying the number of trees to grow
- `mtry` is an optional integer specifying the number of features to randomly select at each split (uses `sqrt(p)` by default, where `p` is the number of features in the data)

The function will return a random forest object that can be used to make predictions.

Making predictions:

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

- `m` is a model trained by the `randomForest()` 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 `"response"`, `"prob"`, or `"votes"` and is used to indicate whether the predictions vector should contain the predicted class, the predicted probabilities, or a matrix of vote counts, respectively

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

Example:

```
credit_model <- randomForest(credit_train, loan_default)
credit_prediction <- predict(credit_model, credit_test)
```

Figura 6: Sintaxis de bosques aleatorios.

Por defecto, la función randomForest() crea un conjunto de 500 árboles de decisión que consideran sqrt(p) características aleatorias en cada división, donde p es el número de características en el conjunto de datos de entrenamiento y sqrt() se refiere a la función raíz cuadrada de R. Por ejemplo, dado que los datos de crédito tienen 16 características, cada uno de los 500 árboles de decisión solo podría considerar sqrt(16) = 4 predictores cada vez que el algoritmo intenta dividir.

La idoneidad de estos parámetros predeterminados ntree y mtry depende de la naturaleza de la tarea de aprendizaje y de los datos de entrenamiento. Generalmente, los problemas de aprendizaje más complejos y los conjuntos de datos más grandes (tanto con más características como con más ejemplos) justifican un mayor número de árboles, aunque esto debe equilibrarse con el gasto computacional que supone entrenar más árboles. Una vez que el parámetro ntree se establece en un valor suficientemente grande, el parámetro mtry se puede ajustar para determinar la mejor configuración; sin embargo, el valor predeterminado suele funcionar bien en la práctica.

Suponiendo que el número de árboles sea suficientemente grande, el número de características seleccionadas aleatoriamente puede ser sorprendentemente bajo antes de que el rendimiento se degrade; sin embargo, probar algunos valores sigue siendo una buena práctica. Idealmente, el número de árboles debería ser lo suficientemente grande como para que cada característica tenga la posibilidad de aparecer en varios modelos.

Veamos cómo funcionan los parámetros predeterminados de randomForest() con los datos de crédito. Entrenaremos el modelo como lo hemos hecho con otros aprendices. Como es habitual, la función set.seed() garantiza que el resultado pueda replicarse:

```
> library(randomForest
> set.seed(300)
> rf <- randomForest(default ~ ., data = credit)
```

Para obtener un resumen del rendimiento del modelo, simplemente escriba el nombre del objeto resultante:

```
> rf
Call:
 randomForest(formula = default ~ ., data = credit)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 4

        OOB estimate of  error rate: 23.3%
Confusion matrix:
     no yes class.error
no  638  62  0.08857143
yes 171 129  0.57000000
```

El resultado muestra que el bosque aleatorio incluyó 500 árboles y probó cuatro variables en cada división, como se esperaba. A primera vista, podría resultar alarmante el rendimiento aparentemente bajo según la matriz de confusión: la tasa de error del 23.3 % es mucho peor que el error de resustitución de cualquiera de los otros métodos de conjunto hasta la fecha.

Sin embargo, esta matriz de confusión no muestra un error de resustitución. En cambio, refleja la tasa de error fuera de bolsa, '**out-of-bag error rate**' (indicada en el resultado como estimación fuera de bolsa de la tasa de error, *OOB estimate of error rate*), que, a diferencia del error de resustitución, es una estimación imparcial del error del conjunto de prueba. Esto significa que debería ser una estimación justa del rendimiento futuro.

La estimación fuera de bolsa se calcula mediante una técnica inteligente durante la construcción del bosque aleatorio. En esencia, cualquier ejemplo no seleccionado para la muestra de bootstrap de un solo árbol puede utilizarse para probar el rendimiento del modelo con datos no analizados. Al finalizar la construcción del bosque, para cada uno de los 1000 ejemplos del conjunto de datos, cualquier árbol que no haya utilizado el ejemplo en el entrenamiento puede realizar una predicción.

Estas predicciones se contabilizan y se realiza una votación para determinar la predicción final única para el ejemplo. La tasa de error total de dichas predicciones en los 1000 ejemplos se convierte en la tasa de error out-of-bag. Dado que cada predicción utiliza solo un subconjunto del bosque, no equivale a una validación real ni a una estimación del conjunto de prueba, pero es un sustituto razonable.

En el tema, Evaluación del Rendimiento del Modelo, se indicó que cualquier ejemplo dado tiene un 63.2 % de probabilidad de ser incluido en una muestra bootstrap. Esto implica que un promedio del 36.8 % de los 500 árboles del bosque aleatorio votaron por cada uno de los 1000 ejemplos en la estimación out-of-bag.

Para calcular el estadístico kappa de las predicciones out-of-bag, podemos utilizar la función del paquete vcd de la siguiente manera. El código aplica la función Kappa() a las dos primeras filas y columnas del objeto de confusión, que almacena la matriz de confusión de las predicciones out-of-bag para el objeto del modelo de bosque aleatorio rf:

```
> library(vcd)
> Kappa(rf$confusion[1:2,1:2])

            value     ASE     z  Pr(>|z|)
Unweighted 0.381 0.03215 11.85 2.197e-32
Weighted   0.381 0.03215 11.85 2.197e-32
```

Con un estadístico kappa de 0.381, el bosque aleatorio es nuestro modelo con mejor rendimiento hasta la fecha. Su rendimiento fue superior al del conjunto de árboles de decisión bagged, cuyo kappa fue de aproximadamente 0.332, así como al del modelo AdaBoost.M1, cuyo kappa fue de aproximadamente 0.340.

El paquete ranger, como se mencionó anteriormente, es una implementación sustancialmente más rápida del algoritmo de bosque aleatorio. Para un conjunto de datos tan pequeño como el de crédito, optimizar la eficiencia computacional puede ser menos importante que la facilidad de uso, y por defecto, ranger sacrifica algunas ventajas para aumentar la velocidad y reducir el consumo de memoria. Por consiguiente, aunque la función ranger es casi idéntica a randomForest() en sintaxis, en la práctica, podrías encontrarte con que rompe el código existente o requiere investigar un poco las páginas de ayuda.

Para recrear el modelo anterior usando ranger, simplemente cambiamos el nombre de la función:

```
> library(ranger)
> set.seed(300)
> m_ranger <- ranger(default ~ ., data = credit)
```

El modelo resultante tiene un error de predicción out-of-bag bastante similar:

```
> m_ranger

Ranger result

Call:
 ranger(default ~ ., data = credit)

Type:                             Classification
Number of trees:                  500
Sample size:                      1000
Number of independent variables:  16
Mtry:                             4
Target node size:                 1
Variable importance mode:         none
Splitrule:                        gini
OOB prediction error:             23.10 %
```

Podemos calcular kappa de forma similar a la anterior, pero observando la ligera diferencia en el nombre del sub objeto de la matriz de confusión del modelo:

```
> Kappa(m_ranger$confusion.matrix)

            value    ASE     z  Pr(>|z|)
Unweighted 0.381 0.0321 11.87 1.676e-32
Weighted   0.381 0.0321 11.87 1.676e-32
```

El valor kappa es 0.381, que es el mismo que el resultado del modelo de bosque aleatorio anterior.
Cabe destacar que esto es una coincidencia, ya que no se garantiza que ambos algoritmos produzcan resultados idénticos.

Al igual que con AdaBoost, los resultados anteriores se obtuvieron con R versión 4.3.3 en un PC con Windows y se verificaron en Mac OS.