

TP1: Clasificación binaria - Predicción de la comestibilidad de hongos

Micaela Oliva, Camila Bernardez

2024-09-01

Ejercicio 1: Introducción al problema

El dataset elegido tiene como origen:

<https://www.kaggle.com/datasets/vishalpnaik/mushroom-classification-edible-or-poisonous?resource=download&select=mushroom.csv>

Es un dataset que busca clasificar si un hongo es comestible (edible) o no (poisonous). Para considerarlo, el dataset esta compuesto de 16 variables, de las cuáles:

- **class** (variable categórica binaria): indica si un hongo es comestible o no, y es lo que buscamos predecir.
[edible, poisonous]
- **cap-diameter** (variable numérica): indica el diametro del sombrero del hongo en cm.
- **cap-shape** (variable categórica): indica el forma del sombrero del hongo.
['bell', 'conical', 'convex', 'flat', 'sunken', 'spherical', 'others']
- **cap-surface** (variable categórica): indica la textura de la superficie del sombrero del hongo.
['fibrous', 'grooves', 'scaly', 'smooth', 'dry', 'shiny', 'leathery', 'silky', 'sticky', 'wrinkled', 'fleshy']
- **cap-color** (variable categórica): indica el color del sombrero del hongo.
['brown', 'orange', 'buff', 'gray', 'green', 'pink', 'purple', 'red', 'white', 'yellow', 'blue', 'black']
- **does-bruise-or-bleed** (variable categórica binaria -> true/false): indica si el hongo al lesionarse presenta moratones o sangrado.
['true', 'false']
- **gill-attachment** (variable categórica): indica cómo las láminas del hongo se adhieren al pie.
['adnate', 'adnexed', 'decurrent', 'free', 'sinuate', 'pores', 'none']
- **gill-spacing** (variable categórica): indica la separación entre las láminas del hongo.
['close', 'distant', 'none']
- **stem-height** (variable numérica): indica la altura del pie del hongo en cm.
- **stem-width** (variable numérica): indica el ancho del pie del hongo en mm.
- **stem-root** (variable categórica): indica la estructura de la raíz del pie del hongo.
['bulbous', 'swollen', 'club', 'cup', 'equal', 'rhizomorphs', 'rooted'] #funnel

- **veil-type** (variable categórica): indica el tipo de velo que cubre las láminas del hongo.
[- 'partial', 'universal']
- **has-ring** (variable categórica binaria -> true/false): indica si esta presente un anillo en el hongo o no.
['true', 'false']
- **ring-type** (variable categórica): indica el tipo del anillo presente en el hongo.
['cobwebby', 'evanescent', 'flaring', 'grooved', 'large', 'pendant', 'sheathing', 'zone', 'scaly', 'movable', 'none', 'unknown'] #unknown?
- **habitat** (variable categórica): indica el ambiente en el cual el hongo fue encontrado.
['grasses', 'leaves', 'meadows', 'paths', 'heaths', 'urban', 'waste', 'woods']
- **season** (variable categórica): indica la estación en la cual el hongo es comunmente observado.
['spring', 'summer', 'autumn', 'winter']

El conjunto de datos de mushrooms es ideal para utilizar árboles de decisión por:

- Las variables son diversas, ya que combina variables tanto categóricas, como numéricas, que los árboles de decisión manejan de una manera eficaz.
- La relación entre las características de un hongo (como el color, la forma, y la textura) y su clasificación como **edible** o **poisonous** no es lineal y puede ser compleja. Los árboles de decisión son particularmente buenos para captar estas relaciones no lineales, ya que dividen los datos en múltiples ramas basadas en diferentes combinaciones de características.
- Los árboles de decisión ofrecen interpretaciones claras sobre qué características determinan la clasificación de un hongo como **poisonous**, que es valioso en este contexto, ya que puede ser crucial en aplicaciones prácticas, como la seguridad de la salud.
- Los árboles logran capturar interacciones entre variables, como color y forma del sombrero.
- El tamaño del dataset, que consiste de 46.069 observaciones, permite construir modelos robustos y generalizables, además de aplicar poda para evitar sobreajuste.

Ejercicio 2: Preparación de los datos

Carga del conjunto de datos y realizamiento del preprocesamiento

```
mushrooms <- read.csv("mushrooms_small.csv")

table(is.na(mushrooms))  #verificamos que no hay valores nulos
##
## FALSE
## 967449
nrow(mushrooms)  #verificamos que hay menos de 50.000 observaciones
## [1] 46069
```

Como los datos ya se encuentran sampleados (menos de 50.000 observaciones) y sin valores null, para preprocesarlos nos encargaremos de:

- cambiar los valores de las variables categóricas por palabras enteras en vez de letras.
- quitar columnas que dependen de otras.

Decidimos quitar de nuestro dataset las columnas 'gill-color', 'stem-surface', 'stem-color', 'veil-color' y 'spore-print-color' principalmente para no manejar tantas variables predictoras, pero también porque el dataset original aclaraba que dependían de 'cap-color' y 'cap-surface' y nos pareció que tendrían información similar.

```

mushrooms <- mushrooms[, !(names(mushrooms) %in% c("gill.color",
  "stem.surface", "stem.color", "veil.color", "spore.print.color"))]
ncol(mushrooms)
## [1] 16

```

Las variables categóricas toman valores de letras que pueden ser confusos, ya que no siempre reflejan de manera clara la palabra que representan (por ejemplo 'k' se utiliza en lugar de 'black'), y pueden repetirse entre columnas sin necesariamente significar lo mismo. Para que el manejo de los datos y las leyendas de los gráficos sean más claras decidimos reemplazarlas por las palabras correspondientes, especificadas anteriormente.

```

mushrooms_renamed <- mushrooms %>%
  mutate(class = recode(class, e = "edible", p = "poisonous"),
    cap.shape = recode(cap.shape, b = "bell", c = "conical",
      x = "convex", f = "flat", s = "sunken",
      p = "spherical", o = "others"), cap.surface = recode(cap.surface,
      i = "fibrous", f = "fibrous", g = "grooves",
      y = "scaly", s = "smooth", d = "dry", h = "shiny",
      l = "leathery", k = "silky", t = "sticky",
      w = "wrinkled", e = "fleshy", ), cap.color = recode(cap.color,
      n = "brown", b = "buff", g = "gray", r = "green",
      p = "pink", u = "purple", e = "red", w = "white",
      y = "yellow", l = "blue", o = "orange",
      k = "black"), does.bruise.or.bleed = recode(does.bruise.or.bleed,
      t = "true", f = "false"), gill.attachment = recode(gill.attachment,
      a = "adnate", x = "adnexed", d = "decurrent",
      e = "free", s = "sinuate", p = "pores",
      f = "none", `?` = "unknown"), gill.spacing = recode(gill.spacing,
      c = "close", d = "distant", f = "none"),
    stem.root = recode(stem.root, b = "bulbous",
      s = "swollen", c = "club", u = "cup", e = "equal",
      z = "rhizomorphs", r = "rooted"), veil.type = recode(veil.type,
      p = "partial", u = "universal"), has.ring = recode(has.ring,
      t = "true", f = "false"), ring.type = recode(ring.type,
      c = "cobwebby", e = "evanescent", r = "flaring",
      g = "grooved", l = "large", p = "pendant",
      s = "sheathing", z = "zone", y = "scaly",
      m = "movable", f = "none", `?` = "unknown"),
    habitat = recode(habitat, g = "grasses", l = "leaves",
      m = "meadows", p = "paths", h = "heaths",
      u = "urban", w = "waste", d = "woods"),
    season = recode(season, s = "spring", u = "summer",
      a = "autumn", w = "winter"))

```

```

head(mushrooms_renamed) #verificamos que se hayan cambiado los nombres de las variables
##      class cap.diameter cap.shape cap.surface
## 1 poisonous      15.26    convex    grooves
## 2 poisonous      16.60    convex    grooves
## 3 poisonous      14.07    convex    grooves
## 4 poisonous      14.17     flat     shiny
## 5 poisonous      15.34    convex    grooves
## 6 poisonous      12.85     flat    grooves
##   cap.color does.bruise.or.bleed gill.attachment
## 1    orange           false           free

```

```
## 2 orange false free
## 3 orange false free
## 4 red false free
## 5 orange false free
## 6 orange false free
## gill.spacing stem.height stem.width stem.root
## 1 none 16.95 17.09 swollen
## 2 none 17.99 18.19 swollen
## 3 close 17.80 17.74 swollen
## 4 none 15.77 15.98 swollen
## 5 distant 17.84 18.79 swollen
## 6 none 17.27 18.69 swollen
## veil.type has.ring ring.type habitat season
## 1 universal true grooved woods winter
## 2 universal true grooved woods summer
## 3 universal true grooved woods winter
## 4 universal true pendant woods winter
## 5 universal true pendant woods summer
## 6 universal true pendant woods autumn
```

Análisis exploratorio de datos: Estadísticas descriptivas y visualizaciones de las variables principales

Primero veamos de qué tipos de datos son nuestras variables. Encontramos solo dos tipos de datos en nuestro dataset: 3 variables de tipo `numeric` ('cap.diameter', 'stem.height' y 'stem.width') y 13 categóricas de tipo `character`. Además, 3 de las 13 categóricas son binarias, incluyendo a nuestra variable a predecir 'class', que especifica si el hongo en cuestión es o no comestible ('edible' o 'poisonous').

```
str(mushrooms_renamed)
## 'data.frame': 46069 obs. of 16 variables:
## $ class : chr "poisonous" "poisonous" "poisonous" "poisonous" ...
## $ cap.diameter : num 15.3 16.6 14.1 14.2 15.3 ...
## $ cap.shape : chr "convex" "convex" "convex" "flat" ...
## $ cap.surface : chr "grooves" "grooves" "grooves" "shiny" ...
## $ cap.color : chr "orange" "orange" "orange" "red" ...
## $ does.bruise.or.bleed: chr "false" "false" "false" "false" ...
## $ gill.attachment : chr "free" "free" "free" "free" ...
## $ gill.spacing : chr "none" "none" "close" "none" ...
## $ stem.height : num 16.9 18 17.8 15.8 17.8 ...
## $ stem.width : num 17.1 18.2 17.7 16 18.8 ...
## $ stem.root : chr "swollen" "swollen" "swollen" "swollen" ...
## $ veil.type : chr "universal" "universal" "universal" "universal" ...
## $ has.ring : chr "true" "true" "true" "true" ...
## $ ring.type : chr "grooved" "grooved" "grooved" "pendant" ...
## $ habitat : chr "woods" "woods" "woods" "woods" ...
## $ season : chr "winter" "summer" "winter" "winter" ...
```

Comenzamos por analizar la distribución de algunas variables. En primer lugar nos interesa analizar si tenemos una proporción más o menos pareja de los valores que toma `class`. Siendo que hay aproximadamente 44 observaciones de `edible` para cada 55 observaciones de `poisonous` nos aseguramos una buena distribución de valores para que el modelo tenga suficientes ejemplos de ambas clases para aprender y generalizar adecuadamente.

```
num_edible <- sum(mushrooms_renamed$class == "edible")
num_poisonous <- sum(mushrooms_renamed$class == "poisonous")
```

```
prop_edible <- num_edible/(nrow(mushrooms_renamed))
prop_poisonous <- num_poisonous/(nrow(mushrooms_renamed))

print(paste("Edible:", num_edible, "(", round(prop_edible *
  100, 2), "%)"))
## [1] "Edible: 20516 ( 44.53 %)"
print(paste("Poisonous:", num_poisonous, "(", round(prop_poisonous *
  100, 2), "%)"))
## [1] "Poisonous: 25553 ( 55.47 %)"
```

Veamos la distribución de algunas otras variables categóricas. En el gráfico inferior observamos que para algunas variables la distribución de los valores que pueden tomar es más o menos pareja, por ejemplo para 'stem root', 'gill attachment' o - en menor medida - 'cap surface'. Mientras que en otras categorías hay valores sobre representados: para 'cap shape' casi la mitad de los datos son de tipo 'convex' mientras que la otra mitad se distribuye en 6 tipos restantes, y otras categorías como 'cap color', 'ring type' o 'habitat' tienen una diferencia incluso mayor.

Sabemos que una distribución pareja de datos para cada variable permite que el modelo generalice mejor por tener información variada de cada categoría, por lo que la existencia de variables con una distribución desbalanceada podría suponer un problema para el entrenamiento. Puede que ocurra un *overfit* a la clase mayoritaria, lo cual dificulta la predicción cuando los datos pertenecen a las clases minoritarias. En el árbol esto se podría ver representado con cortes que separen a la categoría mayoritaria de las demás, y en consecuencia esto puede disminuir algunas métricas como el F1-score.

```
color <- "steelblue"

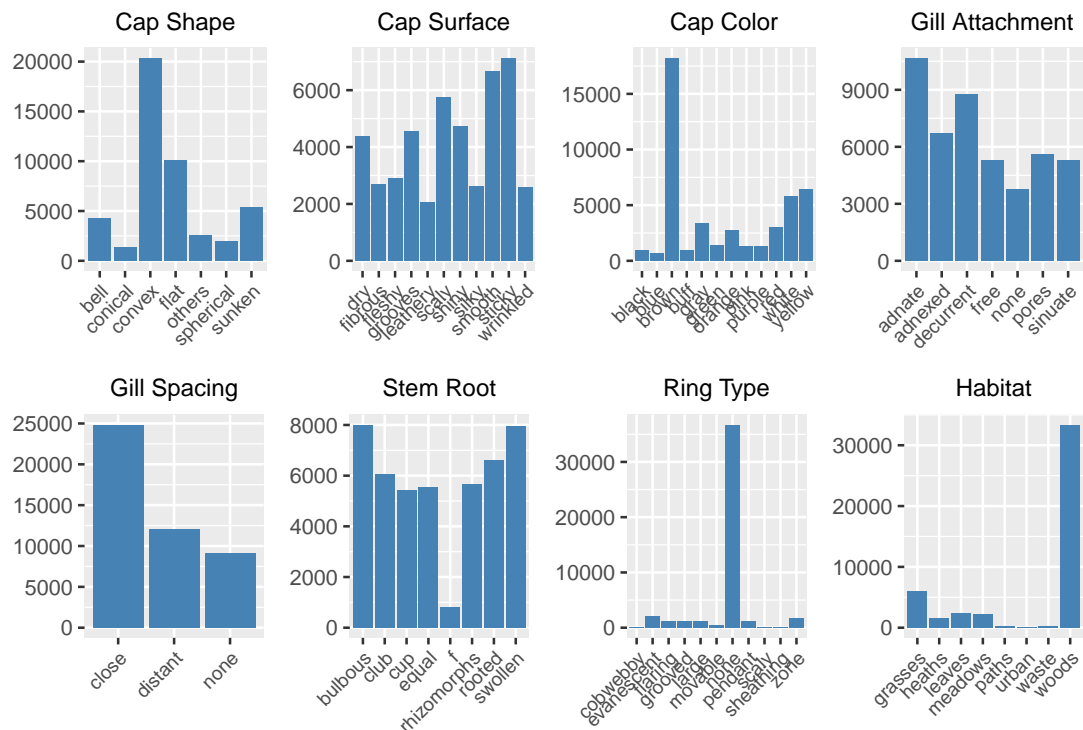
create_plot <- function(data, variable, title) {
  ggplot(data, aes_string(x = variable)) + geom_bar(fill = color) +
    ggtitle(title) + theme(axis.text.x = element_text(angle = 45,
    hjust = 1, size = 8), axis.title.x = element_blank(),
    axis.title.y = element_blank(), plot.title = element_text(size = 10,
    hjust = 0.5))
}

variables <- list(cap.shape = "Cap Shape", cap.surface = "Cap Surface",
  cap.color = "Cap Color", gill.attachment = "Gill Attachment",
  gill.spacing = "Gill Spacing", stem.root = "Stem Root",
  ring.type = "Ring Type", habitat = "Habitat")

plots <- lapply(names(variables), function(var) {
  create_plot(mushrooms_renamed, var, variables[[var]])
})

combined_plot <- (plots[[1]] | plots[[2]] | plots[[3]] |
  plots[[4]])/(plots[[5]] | plots[[6]] | plots[[7]] |
  plots[[8]]) + plot_layout(widths = c(4, 4))

print(combined_plot)
```



Pasemos ahora a inspeccionar las variables numéricas. Visualizamos los datos de una forma más intuitiva con los histogramas, y al superponerles las curvas normales correspondientes observamos que siguen aproximadamente esta distribución (o más bien una distribución lognormal las primeras dos).

```
numeric_columns <- mushrooms_renamed[, c("cap.diameter",
      "stem.height", "stem.width")]
summary(numeric_columns)
##   cap.diameter      stem.height
##   Min.   : 0.380    Min.   : 0.00
##   1st Qu.: 3.480    1st Qu.: 4.63
##   Median : 5.860    Median : 5.95
##   Mean   : 6.709    Mean   : 6.57
##   3rd Qu.: 8.540    3rd Qu.: 7.73
##   Max.   :62.340    Max.   :33.92
##   stem.width
##   Min.   : 0.00
##   1st Qu.: 5.19
##   Median :10.16
##   Mean   :12.14
##   3rd Qu.:16.53
##   Max.   :103.91

create_hist <- function(data, col_name, color) {
  mu <- mean(data[[col_name]])
  sigma <- sd(data[[col_name]])

  ggplot(data, aes_string(x = col_name)) + geom_histogram(aes(y = after_stat(density)),
    fill = color, bins = 30, alpha = 0.6) + stat_function(fun = dnorm,
    args = list(mean = mu, sd = sigma), color = "red",
    size = 0.5) + ggtitle(paste(col_name)) + xlim(0,
```

```

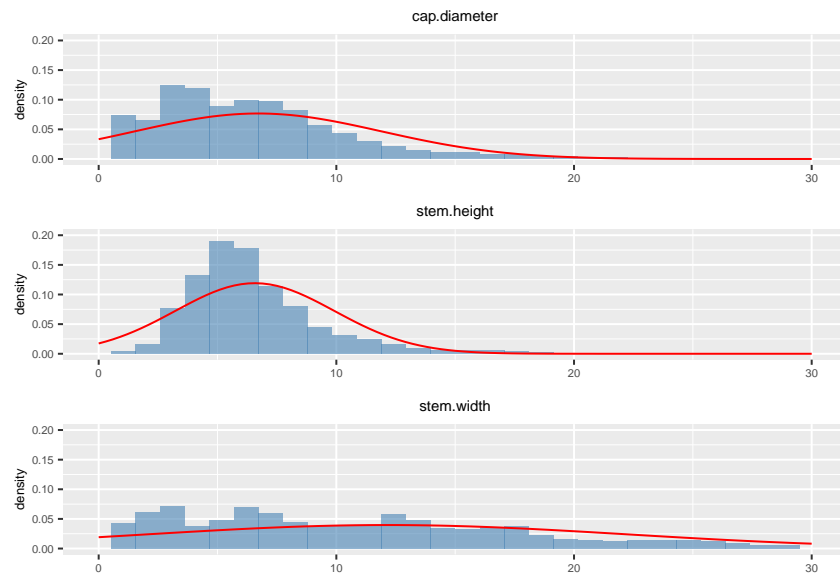
    30) + ylim(0, 0.2) + theme(plot.title = element_text(hjust = 0.5,
    size = 8), axis.text = element_text(size = 6),
    axis.title = element_text(size = 7), axis.title.x = element_blank())
}

p1 <- create_hist(mushrooms_renamed, "cap.diameter",
  "steelblue")
p2 <- create_hist(mushrooms_renamed, "stem.height",
  "steelblue")
p3 <- create_hist(mushrooms_renamed, "stem.width",
  "steelblue")

combined_plot <- p1/p2/p3

print(combined_plot)

```



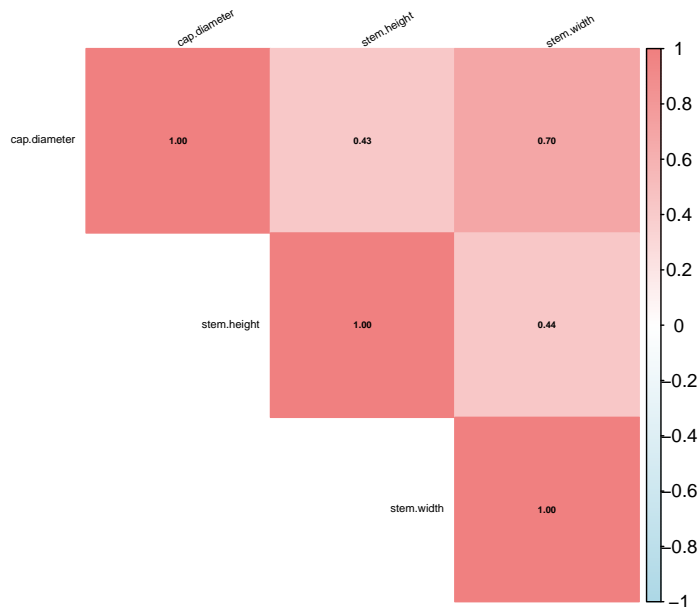
Por último nos interesa analizar la correlación entre las variables. Si miramos únicamente las variables numéricas observamos una correlación alta (0.7) entre 'stem width' y 'cap.diameter', y una correlación considerable (0.43 y 0.44) 'stem height' y las otras dos mencionadas. Esto podría significar que si una de ellas es utilizada en un corte de alto nivel, las demás posiblemente también ya que puede que provean información similar.

```

cor_matrix <- cor(numeric_columns, use = "complete.obs")

corrplot(cor_matrix, method = "color", type = "upper",
  tl.col = "black", tl.srt = 30, tl.cex = 0.5, addCoef.col = "black",
  number.cex = 0.4, col = colorRampPalette(c("lightblue",
  "white", "lightcoral"))(200))

```



También nos interesa saber si hay algunas variables más correlacionadas con nuestra variable objetivo que otras. Esto significaría que es posible que un modelo menos complejo (de menos variables) sea suficiente para predecirla. Vemos que para las variables binarias:

- ‘does bruise or bleed’ no parece tener correlación con la variable a predecir, ya que tanto para las instancias de ‘true’ como para las ‘false’ la proporción que es comestible es alrededor del 50%.
- cuando ‘has ring’ es falsa también no hay una relación aparente con `class`, pero cuando sí tiene anillo parece haber una correlación positiva con la posibilidad de que sea venenoso (0.605)

```
prop_bruise <- round(prop.table(table(mushrooms_renamed$class,
mushrooms_renamed$does.bruise.or.bleed), margin = 2),
3)
prop_ring <- round(prop.table(table(mushrooms_renamed$class,
mushrooms_renamed$has.ring), margin = 2), 3)

cat("Proportion Table for 'does.bruise.or.bleed':\n")
## Proportion Table for 'does.bruise.or.bleed':
print(prop_bruise)
##
##           false true
## edible      0.441 0.464
## poisonous  0.559 0.536

cat("\nProportion Table for 'has.ring':\n")
##
## Proportion Table for 'has.ring':
print(prop_ring)
##
##           false true
## edible      0.462 0.395
## poisonous  0.538 0.605
```


Ejercicio 3: Construcción de un árbol de decisión básico

Comenzamos convirtiendo las variables a tipo factor.

```
convert_to_factors <- function(data, columns) {  
  data[columns] <- lapply(data[columns], as.factor)  
  return(data)  
}  
  
cols <- c("cap.shape", "cap.surface", "cap.color",  
  "gill.attachment", "gill.spacing", "stem.root",  
  "ring.type", "veil.type", "habitat", "season",  
  "class", "does.bruise.or.bleed", "has.ring")  
  
mushrooms_renamed <- convert_to_factors(mushrooms_renamed,  
  cols)
```

A continuación, creamos los subsets de datos con un 70% para entrenamiento, 15% para validación y 15% para testing, tomando observaciones aleatorias.

```
set.seed(42) #semilla para replicabilidad  
  
n <- nrow(mushrooms_renamed)  
indices <- sample(1:n) #tomamos indices aleatorios, es decir filas aleatorias, para luego solo tomar d  
  
train_size <- floor(0.7 * n)  
valid_size <- floor(0.15 * n)  
  
train_indices <- indices[1:train_size]  
valid_indices <- indices[(train_size + 1):(train_size +  
  valid_size)]  
test_indices <- indices[(train_size + valid_size +  
  1):n]  
  
# creamos los subsets de datos  
train_set <- mushrooms_renamed[train_indices, ]  
valid_set <- mushrooms_renamed[valid_indices, ]  
test_set <- mushrooms_renamed[test_indices, ]
```

Hiperparámetros

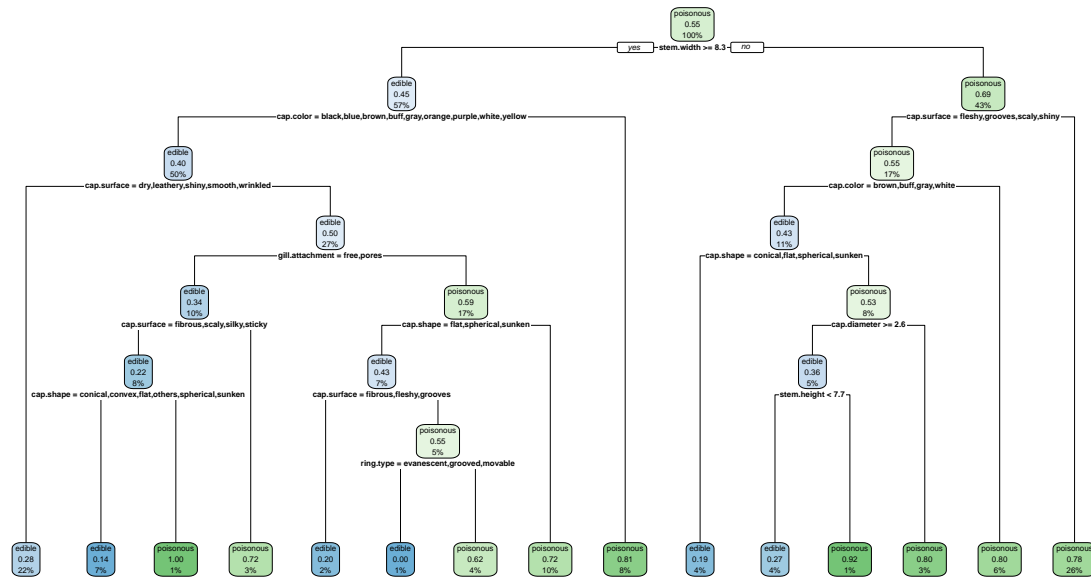
Los parámetros e hiperparámetros que utilizamos son los que vienen por defecto en **rpart**. Los principales hiperparámetros son:

- **minsplit** = define el número mínimo de observaciones que debe haber en un nodo para realizar un corte (split). Por defecto el valor de este hiperparámetro es 20. Un valor elevado puede producir un árbol con menor profundidad, pero menos riesgo de sufrir de overfitting. Aunque, por el otro lado, si es demasiado elevado puede que no realicemos los suficientes cortes para realizar una predicción útil.
- **minbucket** = define el número mínimo de observaciones que debe haber por hoja. Por defecto su valor es $\text{minsplit}/3$ (si **minsplit** también se deja por defecto, sería casi 7). Un nivel muy bajo cercano a 1 puede correr el riesgo de realizar overfitting sobre el modelo.
- **maxdepth** = define la profundidad máxima del árbol. Por defecto toma un valor de 30. Similar a los parámetros anteriores, un valor muy alto puede causar overfitting y un valor muy bajo puede resultar en predicciones malas.

Otros hiperparámetros son **cp**, **xval**, **maxcompete**, **maxsurrogate**, **usersurrogate** y **surrogatestyle**.

Estructura del árbol binario

```
tree_model <- rpart(class ~ ., data = train_set, method = "class") #dejamos los parametros por defecto
rpart.plot(tree_model)
```



El primer corte utiliza la variable ‘stem.width’ y con esta se logra un primer split bastante parejo: un 57% de los tallos tienen un ancho mayor a los 8.3 centímetros y se clasifican como venenosos, mientras que el 43% restante van a la categoría de comestibles. Luego, otros dos criterios de decisión importantes son la superficie y el color de la parte superior del hongo.

En el segundo nivel ya se comienzan a tomar algunas decisiones finales sobre la clasificación. Por ejemplo, si ya se había pre-clasificado como venenoso por tener un tallo ancho y luego no tiene uno de los colores listados (por defecto deja como opciones el rosa, rojo o verde), ya se determina que un 8% del total de hongos que cumple con esas características es venenoso. Por el lado derecho del árbol un 26% también se clasifica como venenoso si no tiene un cierto tipo de superficie.

Las características sobre la superficie reaparecen en otro niveles para afinar aun mas la clasificación. Otras variables utilizadas son ‘cap.shape’ (varias veces), ‘gill.attachment’, ‘stem.height’ y ‘cap.diameter’. En el último nivel antes de las hojas la última clasificación se realiza en base al ‘ring.type’.

Las variables ‘habitat’, ‘season’, ‘does-bruise-or-bleed’, ‘gill.spacing’, ‘stem.root’, ‘veil.type’ y ‘has.ring’ no fueron usadas.

Ejercicio 4: Evaluación del árbol de decisión básico

Comenzamos evaluando las predicciones sobre el conjunto de *testing*, tanto para las probabilidades como la clase predicha. A partir de estos datos podemos comenzar a calcular las métricas que nos interesan.

```
pred_prob <- predict(tree_model, test_set, type = "prob") # Probabilidades predichas
pred_class <- predict(tree_model, test_set, type = "class") # Clases predichas
```

Matriz de confusión

Hacemos la Matriz de Confusión:

```
conf_matrix <- confusionMatrix(pred_class, test_set$class)
print("Matriz de Confusión:")
## [1] "Matriz de Confusión:"
print(conf_matrix$table)
##               Reference
## Prediction edible poisonous
## edible      2075      613
## poisonous   1008     3215
```

Trasponemos para verla como visto en clase (las predicciones en las columnas y el valor verdadero en las filas):

```
print(t(conf_matrix$table))
##               Prediction
## Reference edible poisonous
## edible      2075      1008
## poisonous   613      3215
```

Lo que obtenemos es:

- $TP = 2075$: El True Positive indica la cantidad de instancias donde el modelo predijo **edible** y la clase verdaderamente tomaba ese valor.
- $FP = 613$: El False Positive indica la cantidad de instancias donde el modelo predijo **edible**, cuando en realidad la clase tomaba el valor **poisonous**.
- $FN = 1008$: El False Negative indica la cantidad de instancias donde el modelo predijo **poisonous**, cuando en realidad la clase tomaba el valor **edible**.
- $TN = 3215$: El True Negative Indica la cantidad de instancias donde el modelo predijo **poisonous** y la clase verdaderamente tomaba ese valor.

Necesitamos de otras métricas para evaluar si el resultado fue bueno, por lo que a continuación calculamos la **accuracy**.

Accuracy

Buscamos ver la proporción de predicciones correctas (tanto positivas como negativas) de las predicciones totales.

$$\begin{aligned} Accuracy &= \frac{TP + TN}{P + N} = \\ &= \frac{TP + TN}{TP + FN + FP + TN} = \\ &= \frac{TP + TN}{TP + FN + FP + TN} = \\ &= \frac{2075 + 3215}{2075 + 1008 + 613 + 3215} = \end{aligned}$$

$$= \frac{5290}{6911} = 0.7654464$$

Verificamos el número obtenido con R y con el método de ['Accuracy'] de la misma matriz de confusión.

```
# indexamos la matriz original antes de
# trasponerla
TP <- conf_matrix$table[1, 1]
FP <- conf_matrix$table[1, 2]
FN <- conf_matrix$table[2, 1]
TN <- conf_matrix$table[2, 2]

# Calculamos el accuracy
Accuracy <- (TP + TN)/(TP + FN + FP + TN)
print(paste("Accuracy: ", Accuracy))
## [1] "Accuracy: 0.765446389813341"

# Validacion del cálculo manual
print(paste("Accuracy (conf matrix): ", conf_matrix$overall["Accuracy"]))
## [1] "Accuracy (conf matrix): 0.765446389813341"
```

Esta métrica indica que el modelo clasifica correctamente alrededor de 76.5% de las instancias, lo cual significa que en este caso $Error\ rate = 1 - Accuracy = 0.2345536$.

Pero para considerar que las clases pueden llegar a estar desbalanceadas, usaremos Precision-Recall:

Precision and Recall

Precision

Buscamos ver la proporción de las instancias predichas como **edible** (positivas) que verdaderamente toman ese valor.

$$Precision_{edible} = \frac{TP}{TP + FP} =$$

$$= \frac{2075}{2075 + 613} = 0.7719494$$

```
Precision <- TP/(TP + FP)
Precision
## [1] 0.7719494

# Validacion del cálculo manual
print(paste("Precision (conf matrix): ", conf_matrix$byClass["Pos Pred Value"]))
## [1] "Precision (conf matrix): 0.771949404761905"
```

Esto significa que alrededor de 77.2% del tiempo cuando el modelo predice la clase **edible**, acierta.

Recall

Buscamos ver la proporción de las instancias que verdaderamente valen **edible** que fueron correctamente predichas como tal.

$$Recall_{edible} = \frac{TP}{TP + FN} =$$

$$= \frac{2075}{2075 + 1008} = 0.6730457$$

En R:

```

Recall <- TP/(TP + FN)
print(paste("Recall: ", Recall))
## [1] "Recall: 0.673045734674019"

# Validacion del cálculo manual
print(paste("Recall: ", conf_matrix$byClass["Sensitivity"]))
## [1] "Recall: 0.673045734674019"

```

Esto significa que alrededor de 67.3% el modelo identifica correctamente si los hongos son **edible**, es decir, comestibles.

Queremos también una ponderación conjunta de Precision y de Recall, por lo cual calculamos el F1-Score.

F1-Score

$$\begin{aligned}
 F1 - Score &= \frac{2 * Precision * Recall}{Precision + Recall} = \\
 &= \frac{2 * 0.7719494 * 0.6730457}{0.7719494 + 0.6730457} = 0.7191128
 \end{aligned}$$

En R:

```

F1_score <- (2 * Precision * Recall)/(Precision + Recall)
print(paste("F1_score: ", F1_score))
## [1] "F1_score: 0.719112805406342"

```

Esto significa que el accuracy del modelo en predecir la clase **edible** es igual a 0.7191128.

ROC-AUC Score

Buscamos ver que tan bien el modelo tiene la capacidad de separar o clasificar las clases. Esto se obtiene graficando la relación entre el *True Positive Rate* (o Recall) y el *False Positive Rate*. Es decir, de los que efectivamente son **edible**, en que proporción acierta y en cuánto erra. En la situación ideal, el TPR es 1.0 mientras que el FPR es 0.0, es decir que acierta siempre (y por ende nunca erra en la clasificación). Mientras más se acerque la curva ROC a esta forma, mejor será nuestro modelo. Medimos el área debajo de esta curva para calcular la métrica AUC-ROC.

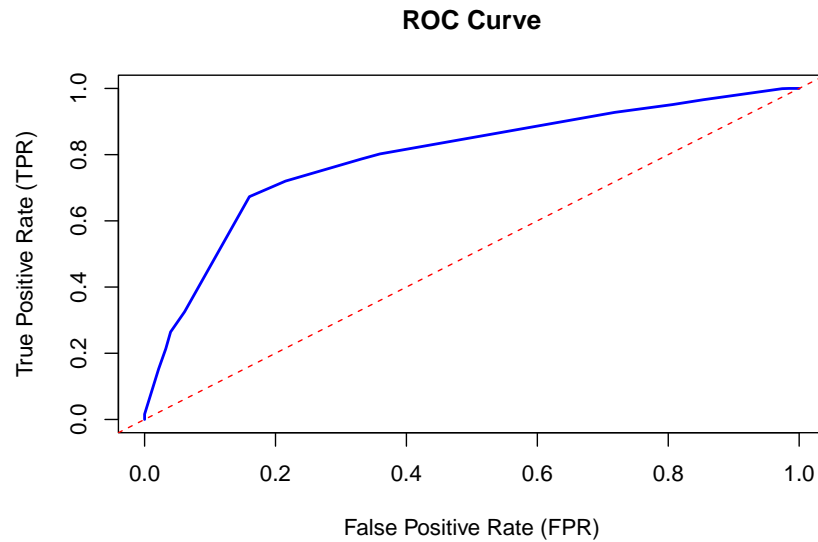
```

roc_obj <- roc.curve(scores.class0 = pred_prob[, "edible"],
  weights.class0 = test_set$class == "edible", curve = TRUE)

plot(roc_obj$curve[, 1], roc_obj$curve[, 2], type = "l",
  col = "blue", lwd = 2, xlab = "False Positive Rate (FPR)",
  ylab = "True Positive Rate (TPR)", main = "ROC Curve")

# Línea diagonal = resultado de una clasificación
# random
abline(a = 0, b = 1, lty = 2, col = "red")

```



```
auc_value <- roc_obj$auc
print(paste("AUC-ROC Score:", auc_value))
## [1] "AUC-ROC Score: 0.79219154760779"
```

$$AUC - ROC = 0.7921915$$

Esto significa que como $0.7921915 > 0.5$, el modelo se ejecuta mejor que hacer random guessing.

Por lo tanto, evaluamos el modelo de árbol de decisión sobre el conjunto de testeo, calculando varias métricas claves. La matriz de confusión mostró un balance aceptable entre verdaderos positivos y negativos, aunque con un número considerable de falsos positivos y negativos. El accuracy fue del 76.5%, lo que indica que el modelo clasifica correctamente la mayoría de las instancias.

La precisión fue del 77.2% y el recall del 67.3%, lo que sugiere que el modelo es más confiable en predecir instancias positivas que en capturar todas las positivas existentes. El F1-score combinó estos aspectos y alcanzó un valor de 0.719.

Finalmente, el AUC-ROC de 0.792 indica que el modelo tiene una buena capacidad para distinguir entre las clases, superando el azar. En resumen, el modelo tiene un desempeño razonable.

Ejercicio 5: Optimización del modelo

En esta sección, optimizaremos el modelo de árbol de decisión ajustando los hiperparámetros *maxdepth*, *minsplit*, y *minbucket*. La optimización se realizará evaluando el rendimiento del modelo en términos de AUC-ROC sobre un conjunto de validación. Exploraremos varias combinaciones de estos hiperparámetros para identificar la configuración que maximice esta métrica.

Primero, definimos una función `opt.hyperparams` que ajusta el modelo de árbol de decisión para cada combinación de hiperparámetros y calcula el AUC-ROC en el conjunto de validación.

```
opt_hyperparams <- function(train_data, valid_data,
                             maxdepth_set, minsplit_set, minbucket_set) {

  results <- data.frame(maxdepth = integer(), minsplit_val = integer(),
                        minbucket_val = integer(), auc = numeric())
```

```

# grid search de hiperparametros
for (depth in maxdepth_set) {
  for (split in minsplit_set) {
    for (bucket in minbucket_set) {
      tree_model_modified <- rpart(class ~
        ., data = train_data, method = "class",
        control = rpart.control(minbucket = bucket,
          minsplit = split, maxdepth = depth,
          cp = 0, xval = 0))

      # predecir en el set de
      # validacion
      pred_probs <- predict(tree_model_modified,
        newdata = valid_data, type = "prob")[,
        2]

      # AUC-ROC
      roc_curve <- roc(valid_data$class,
        pred_probs)
      auc_value <- auc(roc_curve)

      results <- rbind(results, data.frame(maxdepth = depth,
        minsplit_val = split, minbucket_val = bucket,
        auc = auc_value))
    }
  }
}

return(results)
}

```

Ahora, definimos los conjuntos de valores que probaremos para los hiperparámetros. Ejecutamos la función `opt.hiperparams` para encontrar la combinación óptima.

```

# definimos las opciones de hiperparametros
# teniendo en cuenta que nuestro modelo tiene +46
# mil observaciones. por ejemplo, un minsplit muy
# chico podria resultar en un arbol muy grande ya
# que con tantas observaciones lo mas probable es
# que haya suficientes para hacer un corte.
maxdepth_options <- c(3, 5, 7, 9, 11)
minsplit_options <- c(1000, 3000, 6000, 10000, 20000)
minbucket_options <- c(50, 100, 500, 1000, 5000)

results <- opt_hiperparams(train_set, valid_set, maxdepth_options,
  minsplit_options, minbucket_options)

best_model_params <- results[which.max(results$auc),
]

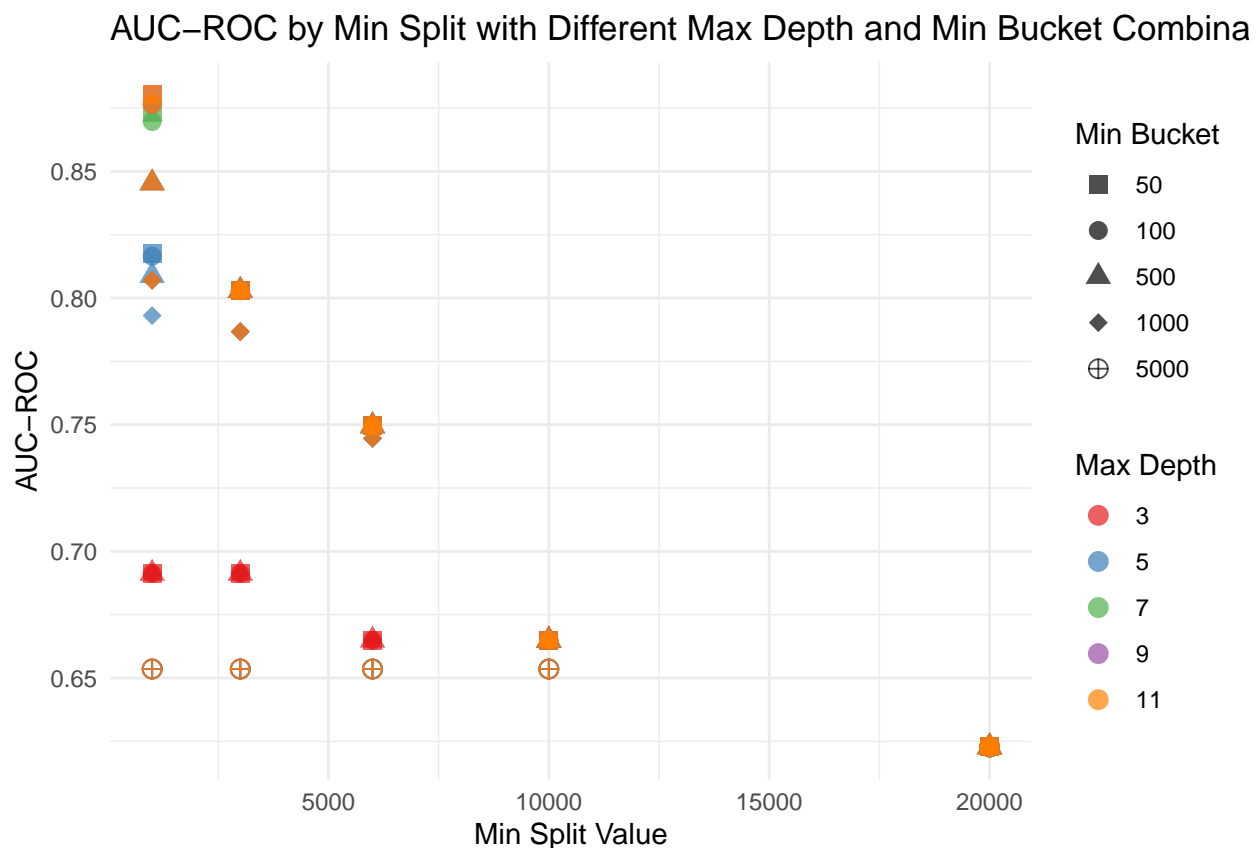
best_maxdepth <- best_model_params$maxdepth
best_maxdepth <- as.integer(as.character(best_maxdepth))
best_minsplit <- best_model_params$minsplit_val
best_minbucket <- best_model_params$minbucket_val

```

Vemos el rendimiento para cada combinación de hiperparámetros. Observamos que para valores bajos de *minsplit* hay mucha diferencia en el AUC-ROC según la combinación de *maxdepth* y *minbucket* que tomemos. Si tomamos como ejemplo *minsplit* = 1000, mientras mayor sea el *maxdepth* mejor será el rendimiento, aunque *minbucket* también influye (por ejemplo con una profundidad de 11 y *minbucket* 1000 obtenemos pero AUC ROC que con profundidad de 5 y *minbucket* 50).

Sin embargo, a medida que aumenta el *minsplit* los otros hiperparámetros dejan de ser tan relevantes. Incluso para el *minsplit* de 20 000 (que en nuestro caso es casi la mitad del dataset) el AUC ROC es casi igual para todas las combinaciones de *minsplit* y *minbucket* (están superpuestos y por eso no se observan en el gráfico)

```
ggplot(results, aes(x = minsplit_val, y = auc)) + geom_point(aes(color = factor(maxdepth),
  shape = factor(minbucket_val)), size = 3, alpha = 0.7) +
  scale_shape_manual(values = c(15, 16, 17, 18, 10)) +
  scale_color_brewer(palette = "Set1") + scale_y_continuous(breaks = seq(0.45,
  0.9, by = 0.05), labels = function(x) format(x,
  nsmall = 2)) + labs(x = "Min Split Value", y = "AUC-ROC",
  title = "AUC-ROC by Min Split with Different Max Depth and Min Bucket Combinations",
  color = "Max Depth", shape = "Min Bucket") + theme_minimal()
```

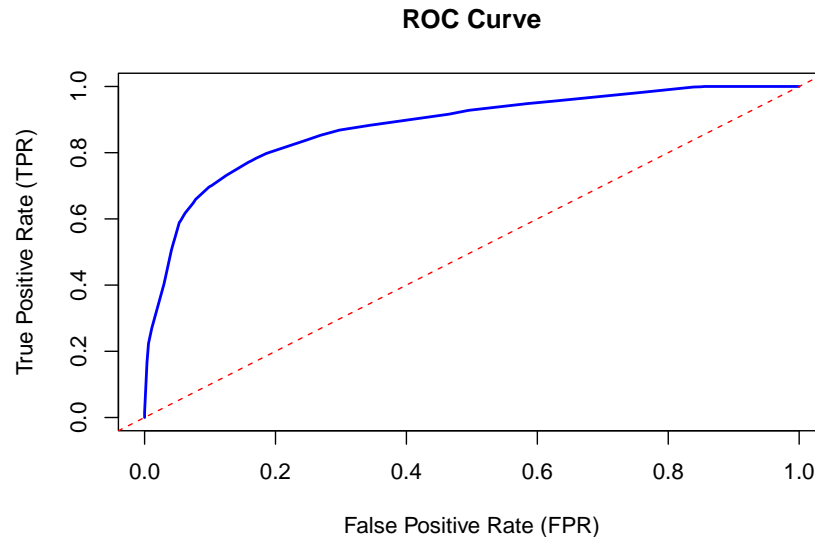


```
print(paste("Max depth *: ", best_maxdepth))
## [1] "Max depth *: 9"
print(paste("Min bucket *: ", best_minbucket))
## [1] "Min bucket *: 50"
print(paste("Min split *: ", best_minsplit))
## [1] "Min split *: 1000"
```

Aclaración: los valores de AUC ROC para *maxdepth* = 9 no se visualizan porque se superponen con otros. Por ejemplo, el *maxdepth* óptimo que calculamos arriba es 9, pero da el mismo AUC que con una profundidad


```
plot(roc_curve$curve[, 1], roc_curve$curve[, 2], type = "l",
     col = "blue", lwd = 2, xlab = "False Positive Rate (FPR)",
     ylab = "True Positive Rate (TPR)", main = "ROC Curve")

abline(a = 0, b = 1, lty = 2, col = "red")
```



A continuación, comparamos el rendimiento del modelo optimizado con el modelo básico evaluando la matriz de confusión, el TPR, y el FPR.

```
prediccion <- predict(new_model, test_set, type = "class") # Clases predichas
```

```
matrix <- confusionMatrix(prediccion, test_set$class)
print("Matriz de Confusión:")
## [1] "Matriz de Confusión:"
```

```
# trasponemos para que se vea como la anterior (y
# la de clase)
```

```
print(t(matrix$table))
##           Prediction
## Reference  edible poisonous
## edible    2328      755
## poisonous  555     3273
```

```
TP <- matrix$table[1, 1]
FP <- matrix$table[1, 2]
FN <- matrix$table[2, 1]
TN <- matrix$table[2, 2]
```

```
TPR <- TP/(TP + FN)
FPR <- FP/(FP + TN)
```

```
print(paste("True Positive Rate (TPR):", TPR))
## [1] "True Positive Rate (TPR): 0.755108660395718"
print(paste("False Positive Rate (FPR):", FPR))
```

```
## [1] "False Positive Rate (FPR): 0.144984326018809"
```

Habiendo optimizado los hiperparámetros obtenemos un valor de AUC-ROC de aproximadamente 0.877, 0.08 puntos mejor que con el árbol básico sin optimizar. Además, logramos aumentar el TPR un poco más de 0.10 puntos, y disminuir el FPR de 0.16 a 0.145. Este último punto es muy relevante, ya que en nuestro modelo

- El TPR indica la proporción de hongos comestibles correctamente identificados como tal.
- El FPR indica la proporción de hongos venenosos incorrectamente identificados como comestibles.

Si el modelo fuera a usarse como una manera confiable de decidir si comer un hongo salvaje, resulta más peligroso que el FPR sea alto a que el TPR sea bajo.

Ejercicio 6: Interpretación de resultados

Explicación del árbol optimizado

El árbol de decisión optimizado presenta algunas diferencias clave con respecto al árbol básico. Aunque los primeros cortes siguen siendo los mismos, reflejando la importancia central de ciertas variables como `cap.color` y `cap.shape`, el árbol optimizado introduce nuevas variables a partir de niveles más profundos, como `does.bruise.bleed`, `habitat`, y `season`.

Estas nuevas variables, aunque no fueron las más influyentes en los primeros cortes, contribuyeron a mejorar la precisión del modelo al refinar las decisiones en ramas específicas. Esto sugiere que, al permitir una mayor profundidad y ajuste en el árbol, logramos capturar relaciones más sutiles en los datos que no fueron aprovechadas por el modelo básico.

Rendimiento

El valor de AUC-ROC del árbol optimizado es de 0.877, lo que representa una mejora significativa de 0.08 puntos en comparación con el árbol básico. Esta mejora se traduce también en un mejor True Positive Rate (TPR) y una reducción del False Positive Rate (FPR). Específicamente, el TPR aumentó en 0.10 puntos, lo que implica que el modelo ahora es más efectivo en identificar correctamente los hongos comestibles. Más crucialmente, el FPR disminuyó de 0.16 a 0.145, reduciendo el riesgo de clasificar incorrectamente hongos venenosos como comestibles — aumentando así la seguridad del modelo en aplicaciones sensibles.

Variables

Aunque las variables clave del árbol básico siguen desempeñando un papel principal en el árbol optimizado, la introducción de nuevas variables en los niveles inferiores sugiere un cambio en la estructura del árbol. La aparición de variables como `habitat` y `season` en las ramas más profundas indica que estas variables, aunque no determinantes por sí solas, contribuyen a mejorar la clasificación cuando se combinan con otras. Esta combinación de variables más específicas permite al modelo optimizado tener un enfoque más refinado, mejorando la capacidad de generalización y aumentando el AUC-ROC.

Consideraciones finales

En resumen, la optimización de los hiperparámetros ha llevado a un árbol más profundo y detallado, que captura relaciones más complejas en los datos, pero manteniendo los cortes iniciales vistos inicialmente. Esto ha resultado en un modelo más preciso y seguro, especialmente en este contexto donde la minimización del FPR es crítico. Este proceso de optimización no solo ha mejorado las métricas de rendimiento, sino que también reveló la importancia de considerar variables adicionales que, en un primer análisis, podrían haber sido subestimadas.

Ejercicio 7: Análisis del impacto de los valores faltantes

En este ejercicio, exploraremos cómo la introducción de valores faltantes (NA) en diferentes proporciones afecta el rendimiento de un modelo de árbol de decisión optimizado. Para ello, se crearán tres conjuntos de

datos con 20%, 50% y 75% de valores faltantes, respectivamente, y se analizará el impacto de estos valores en la performance del modelo.

En este primer bloque de código, definimos una función `add.na.to.df` que nos permite agregar valores faltantes (NA) a un conjunto de datos en una proporción específica.

Simular la pérdida de datos en diferentes grados nos permite evaluar la robustez del modelo frente a escenarios donde la información está incompleta.

```
add_na_to_df <- function(df, na_fraction) {
  predictors <- df %>%
    select(-class)

  num_rows <- nrow(predictors)
  num_na <- round(num_rows * na_fraction)

  df_na <- df

  # Agregar NA a cada columna individualmente
  for (col in names(predictors)) {
    na_indices <- sample(1:num_rows, num_na, replace = FALSE)
    df_na[na_indices, col] <- NA
  }

  return(df_na)
}

# agregamos 20, 50 y 75% de NAs porque los datos
# originales no tenían
veinte_train <- add_na_to_df(train_set, 0.2)
cincuenta_train <- add_na_to_df(train_set, 0.5)
setentaicinco_train <- add_na_to_df(train_set, 0.75)

veinte_valid <- add_na_to_df(valid_set, 0.2)
cincuenta_valid <- add_na_to_df(valid_set, 0.5)
setentaicinco_valid <- add_na_to_df(valid_set, 0.75)

veinte_test <- add_na_to_df(test_set, 0.2)
cincuenta_test <- add_na_to_df(test_set, 0.5)
setentaicinco_test <- add_na_to_df(test_set, 0.75)
```

Sobre cada dataset nuevo de los datos de training, verificamos que haya la cantidad esperada de NAs:

```
total <- nrow(train_set)
num_20 <- ceiling(total * 0.2)
num_50 <- floor(total * 0.5)
num_75 <- ceiling(total * 0.75)
print(paste("Con 20% NAs debería haber ", num_20,
  "datos faltantes"))
## [1] "Con 20% NAs debería haber 6450 datos faltantes"
print(paste("Con 50% NAs debería haber ", num_50,
  "datos faltantes"))
## [1] "Con 50% NAs debería haber 16124 datos faltantes"
print(paste("Con 75% NAs debería haber ", num_75,
  "datos faltantes"))
## [1] "Con 75% NAs debería haber 24186 datos faltantes"
```

Antes de seguir con la optimización, verificamos que los conjuntos de datos generados contengan el número esperado de valores faltantes:

```
# Verificar los resultados con algunas columnas
# samples
summary(veinte_train$cap.diameter)["NA's"]
## NA's
## 6450
summary(cincuenta_train$gill.attachment)["NA's"]
## NA's
## 16124
summary(setentaicinco_train$veil.type)["NA's"]
## NA's
## 24186
```

Ahora, optimizamos el modelo de árbol de decisión para cada uno de los conjuntos de datos generados con NA, utilizando la misma función de optimización de hiperparámetros utilizada en ejercicio 5. Seleccionamos los mejores valores de hiperparámetros para cada conjunto de datos con NA, basándonos en el AUC obtenido en validación.

```
results_veinte <- opt_hiperparams(veinte_train, veinte_valid,
  maxdepth_options, minsplit_options, minbucket_options)
results_cincuenta <- opt_hiperparams(cincuenta_train,
  cincuenta_valid, maxdepth_options, minsplit_options,
  minbucket_options)
results_setentaicinco <- opt_hiperparams(setentaicinco_train,
  setentaicinco_valid, maxdepth_options, minsplit_options,
  minbucket_options)

find_best_params <- function(results) {
  best_model_params <- results[which.max(results$auc),
  ]
  best_maxdepth <- as.integer(as.character(best_model_params$maxdepth))
  best_minsplit <- as.integer(as.character(best_model_params$minsplit))
  best_minbucket <- as.integer(as.character(best_model_params$minbucket))

  return(list(maxdepth = best_maxdepth, minsplit = best_minsplit,
    minbucket = best_minbucket))
}

best_params_veinte <- find_best_params(results_veinte)
best_params_cincuenta <- find_best_params(results_cincuenta)
best_params_setentaicinco <- find_best_params(results_setentaicinco)

print(paste("Veinte Train - Max depth:", best_params_veinte$maxdepth,
  "Min split:", best_params_veinte$minsplit, "Min bucket:",
  best_params_veinte$minbucket))
## [1] "Veinte Train - Max depth: 11 Min split: 1000 Min bucket: 50"
print(paste("Cincuenta Train - Max depth:", best_params_cincuenta$maxdepth,
  "Min split:", best_params_cincuenta$minsplit, "Min bucket:",
  best_params_cincuenta$minbucket))
## [1] "Cincuenta Train - Max depth: 11 Min split: 1000 Min bucket: 50"
print(paste("Setentaicinco Train - Max depth:", best_params_setentaicinco$maxdepth,
  "Min split:", best_params_setentaicinco$minsplit,
  "Min bucket:", best_params_setentaicinco$minbucket))
## [1] "Setentaicinco Train - Max depth: 11 Min split: 1000 Min bucket: 50"
```

Una vez identificados los mejores hiperparámetros, entrenamos un nuevo modelo y evaluamos su rendimiento en el conjunto de testeo.

```
train_and_evaluate <- function(train_set, test_set,
  best_params) {
  new_model <- rpart(class ~ ., data = train_set,
    method = "class", minsplit = best_params$minsplit,
    minbucket = best_params$minbucket, maxdepth = best_params$maxdepth,
    cp = 0, xval = 0)

  test_predictions <- predict(new_model, newdata = test_set,
    type = "prob")[, 2]

  roc_curve <- roc(test_set$class, test_predictions)
  auc_value <- auc(roc_curve)

  return(auc_value)
}

auc_veinte <- train_and_evaluate(veinte_train, veinte_test,
  best_params_veinte)
auc_cincuenta <- train_and_evaluate(cincuenta_train,
  cincuenta_test, best_params_cincuenta)
auc_setentaicinco <- train_and_evaluate(setentaicinco_train,
  setentaicinco_test, best_params_setentaicinco)
```

Esto nos permite medir cómo el rendimiento del modelo se ve afectado cuando se enfrenta a datos incompletos.

```
print(paste("Veinte Train AUC:", auc_veinte))
## [1] "Veinte Train AUC: 0.816215834228965"
print(paste("Cincuenta Train AUC:", auc_cincuenta))
## [1] "Cincuenta Train AUC: 0.71802839991852"
print(paste("Setentaicinco Train AUC:", auc_setentaicinco))
## [1] "Setentaicinco Train AUC: 0.632216276198291"
```

El árbol con mejor performance en testeo es el de menor cantidad de NAs (20%), lo cual tiene sentido porque una mayor cantidad de datos evita la distorsión de datos. Los hiperparámetros que lo optimizan son los mismos que los que optimizaban al árbol del punto 5, pero la performance es 0.07 puntos menor.

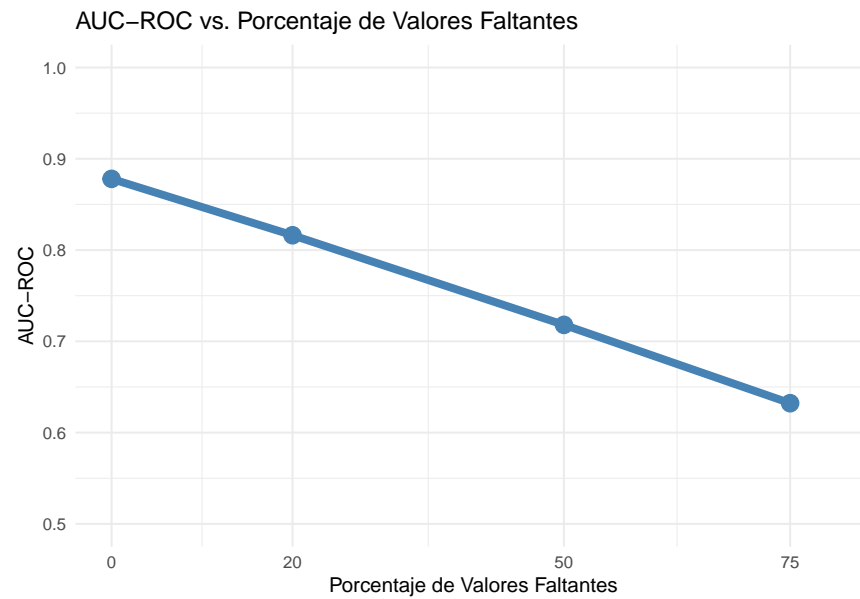
Los valores muestran cómo el AUC del modelo disminuye a medida que aumenta la proporción de valores faltantes. Esto demuestra claramente que los modelos entrenados con datos más completos (menos NA) tienden a tener un mejor rendimiento, ya que tienen más información disponible para hacer predicciones.

Los resultados obtenidos indican que la presencia de valores faltantes tiene un impacto negativo en el rendimiento del modelo. Con un 20% de NA, el modelo todavía mantiene una buena capacidad predictiva, pero a medida que aumentamos el porcentaje de NA a 50% y 75%, el rendimiento disminuye significativamente, lo que subraya la importancia de manejar adecuadamente los datos incompletos en la práctica.

```
auc_results <- data.frame(Percentage_NA = c(0, 20,
  50, 75), AUC_ROC = c(auc_opt, auc_veinte, auc_cincuenta,
  auc_setentaicinco))

ggplot(auc_results, aes(x = Percentage_NA, y = AUC_ROC)) +
  geom_line(color = "steelblue", lwd = 2) + geom_point(color = "steelblue",
  size = 4) + labs(title = "AUC-ROC vs. Porcentaje de Valores Faltantes",
```

```
x = "Porcentaje de Valores Faltantes", y = "AUC-ROC") +  
scale_y_continuous(limits = c(0.5, 1), breaks = seq(0,  
1, by = 0.1)) + scale_x_continuous(limits = c(0,  
80), breaks = c(0, 20, 50, 75)) + theme_minimal()
```



Ejercicio 8: Conclusiones y discusión