

TP1: Clasificación binaria - Predicción de la comestibilidad de hongos

Micaela Oliva, Camila Bernardez

```
knitr::opts_chunk$set(echo = TRUE)
```

Ejercicio 1: Introducción al problema

El dataset elegido tiene como origen:

<https://www.kaggle.com/datasets/vishalpnaiik/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’ #extra
 - ‘equal’ #extra
 - ‘rhizomorphs’ #extra
 - ‘rooted’
 - ‘’ #falta es f, supongo que es none
 - ‘’ #
- **veil-type** (variable categórica): indica el tipo de velo que cubre las láminas del hongo.
 - ‘partial’ #extra
 - ‘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’ #extra
 - ‘evanescent’
 - ‘flaring’
 - ‘grooved’
 - ‘large’
 - ‘pendant’
 - ‘sheathing’, #extra
 - ‘zone’
 - ‘scaly’ #extra
 - ‘movable’
 - ‘none’
 - ‘unknown’ #extra
 - ‘,’
- **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’

Para resumir, la variable **class** es una variable categórica binaria que indica si un hongo es comestible (**edible**) o venenoso (**poisonous**) pra poder hacer una clasificación binaria.

Asimismo, el conjunto de datos tiene varias variables categóricas (cap-shape, cap-surface, cap-color, entre otras) y variables numéricas como cap-diameter, stem-heigh, y stem-width, cumpliendo así la inclusión de ambos tipos de atributos.

Las variables categóricas están representadas como palabras (e.g. cap-shape con valores como ‘bell’, ‘conical’, etc.). Además, hicimos un preprocesamiento de los datos tal que tengamos menos de 50.000 observaciones, pues el dataset original consistía de más de 60K de datos, y que además no tengan valores nulos. ##
HOWWW

Usamos árboles de decisión porque son capaces de manejar tanto variables categóricas como numéricas, permitiendo identificar de manera eficaz patrones complejos y no lineales en los datos. Además, los árboles de decisión pueden dividir los datos en función de múltiples atributos, lo que es útil cuando tenemos varias características relevantes, como en este caso, para clasificar hongos entre comestibles y venenosos.

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.

La fuente del dataset especifica que las columnas 'gill-color', 'stem-surface', 'stem-color', 'veil-color' y 'spore-print-color' dependen de 'cap-color' y 'cap-surface', así que decidimos eliminar estas columnas correlacionadas para disminuir al máximo posible el ruido, que podría distorsionar nuestra clasificación binaria.

```
mushrooms <- mushrooms[, !(names(mushrooms) %in% c("gill.color", "stem.surface", "stem.color", "veil.co"
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.

```
#install.packages("dplyr")
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##   filter, lag

## The following objects are masked from 'package:base':
##
##   intersect, setdiff, setequal, union
```

```
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' = 's',
    cap.surface = recode(cap.surface, 'i' = 'fibrous', 'f' = 'fibrous', 'g' = 'grooves', 'y' = 'scaly',
    cap.color = recode(cap.color, 'n' = 'brown', 'b' = 'buff', 'g' = 'gray', 'r' = 'green', 'p' = 'pink',
    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' = 'e',
    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' = 'e')
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'),
habitat = recode(habitat, 'g' = 'grasses', 'l' = 'leaves', 'm' = 'meadows', 'p' = 'paths', 'h' = 'h'),
season = recode(season, 's' = 'spring', 'u' = 'summer', 'a' = 'autumn', 'w' = 'winter')
)

```

```
head(mushrooms_renamed)
```

```

##      class cap.diameter cap.shape cap.surface cap.color does.bruise.or.bleed
## 1 poisonous      15.26    convex    grooves    orange             false
## 2 poisonous      16.60    convex    grooves    orange             false
## 3 poisonous      14.07    convex    grooves    orange             false
## 4 poisonous      14.17    flat      shiny      red              false
## 5 poisonous      15.34    convex    grooves    orange             false
## 6 poisonous      12.85    flat      grooves    orange             false
##  gill.attachment gill.spacing stem.height stem.width stem.root veil.type
## 1             free         none      16.95     17.09    swollen universal
## 2             free         none      17.99     18.19    swollen universal
## 3             free         close     17.80     17.74    swollen universal
## 4             free         none      15.77     15.98    swollen universal
## 5             free         distant   17.84     18.79    swollen universal
## 6             free         none      17.27     18.69    swollen universal
##  has.ring ring.type habitat season
## 1      true  grooved  woods winter
## 2      true  grooved  woods summer
## 3      true  grooved  woods winter
## 4      true  pendant  woods winter
## 5      true  pendant  woods summer
## 6      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.

```
library(ggplot2)
library(patchwork)

color = "steelblue"

# Define individual plots
p1 <- ggplot(mushrooms_renamed, aes(x = cap.shape)) +
  geom_bar(fill = color) +
  ggtitle("Cap Shape") +
  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))

p2 <- ggplot(mushrooms_renamed, aes(x = cap.surface)) +
  geom_bar(fill = color) +
  ggtitle("Cap Surface") +
  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))

p3 <- ggplot(mushrooms_renamed, aes(x = cap.color)) +
  geom_bar(fill = color) +
  ggtitle("Cap Color") +
  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))

p4 <- ggplot(mushrooms_renamed, aes(x = gill.attachment)) +
  geom_bar(fill = color) +
  ggtitle("Gill Attachment") +
  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))

p5 <- ggplot(mushrooms_renamed, aes(x = gill.spacing)) +
  geom_bar(fill = color) +
  ggtitle("Gill Spacing") +
  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))

p6 <- ggplot(mushrooms_renamed, aes(x = stem.root)) +
  geom_bar(fill = color) +
  ggtitle("Stem Root") +
  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))

p8 <- ggplot(mushrooms_renamed, aes(x = ring.type)) +
  geom_bar(fill = color) +
  ggtitle("Ring Type") +
  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))

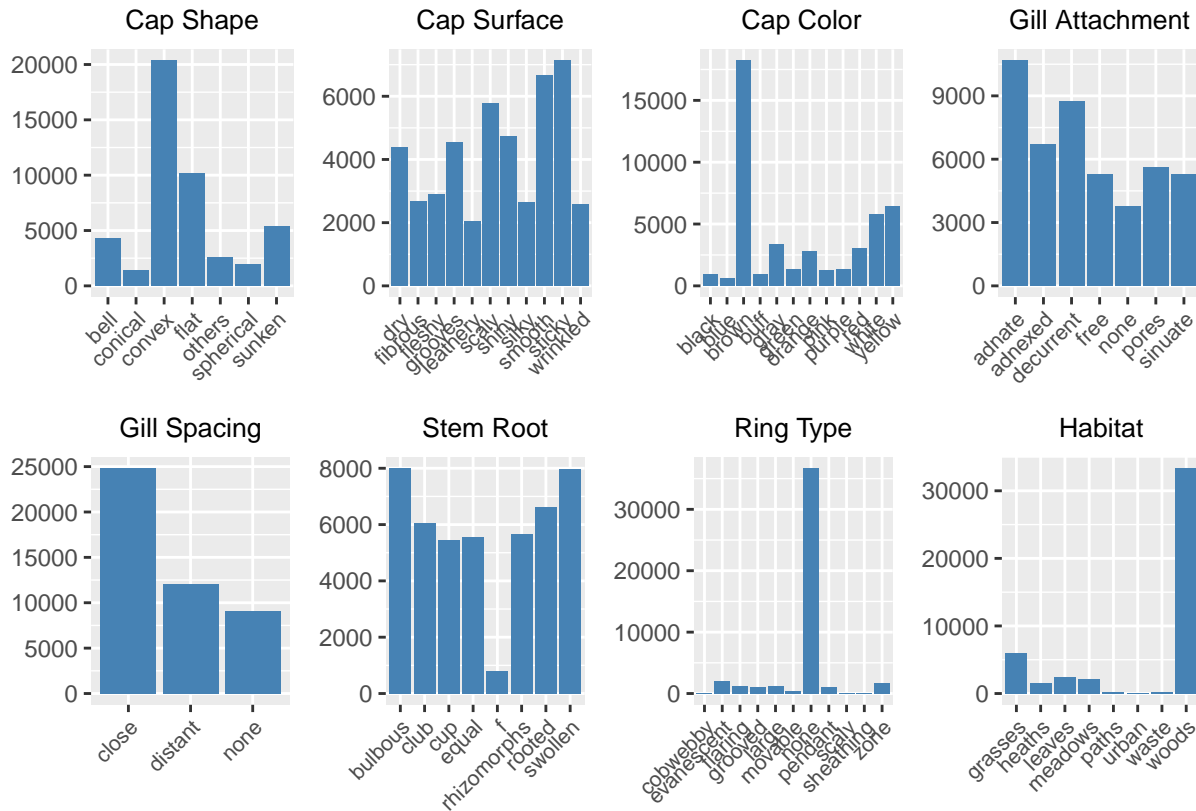
p9 <- ggplot(mushrooms_renamed, aes(x = habitat)) +
  geom_bar(fill = color) +
  ggtitle("Habitat") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1, size = 8),
        axis.title.x = element_blank(),
        axis.title.y = element_blank(),
        plot.title = element_text(size = 10, hjust = 0.5))

# Combine all the plots into a wider grid layout

```

```
combined_plot <- (p1 | p2 | p3 | p4) /
  (p5 | p6 | p8 | p9) +
  plot_layout(widths = c(4, 4)) # Increase the widths for more space

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.

```
numeric_columns <- mushrooms_renamed[, c("cap.diameter", "stem.height", "stem.width")]
summary(numeric_columns)
```

```
##   cap.diameter   stem.height   stem.width
##   Min.   : 0.380   Min.   : 0.00   Min.   : 0.00
##   1st Qu.: 3.480   1st Qu.: 4.63   1st Qu.: 5.19
##   Median : 5.860   Median : 5.95   Median : 10.16
##   Mean   : 6.709   Mean   : 6.57   Mean   : 12.14
##   3rd Qu.: 8.540   3rd Qu.: 7.73   3rd Qu.: 16.53
##   Max.   :62.340   Max.   :33.92   Max.   :103.91
```

```
create_histogram_with_normal <- function(data, column_name, color) {
  mu <- mean(data[[column_name]], na.rm = TRUE)
  sigma <- sd(data[[column_name]], na.rm = TRUE)

  ggplot(data, aes_string(x = column_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(column_name)) +
theme(plot.title = element_text(hjust = 0.5, size = 8), # Título más pequeño
      axis.text = element_text(size = 6), # Texto del eje más pequeño
      axis.title = element_text(size = 7),
      axis.title.x = element_blank()) # Título del eje más pequeño
}

# Crear los histogramas
p1 <- create_histogram_with_normal(mushrooms_renamed, "cap.diameter", "steelblue")

## Warning: `aes_string()` was deprecated in ggplot2 3.0.0.
## i Please use tidy evaluation idioms with `aes()`.
## i See also `vignette("ggplot2-in-packages")` for more information.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.

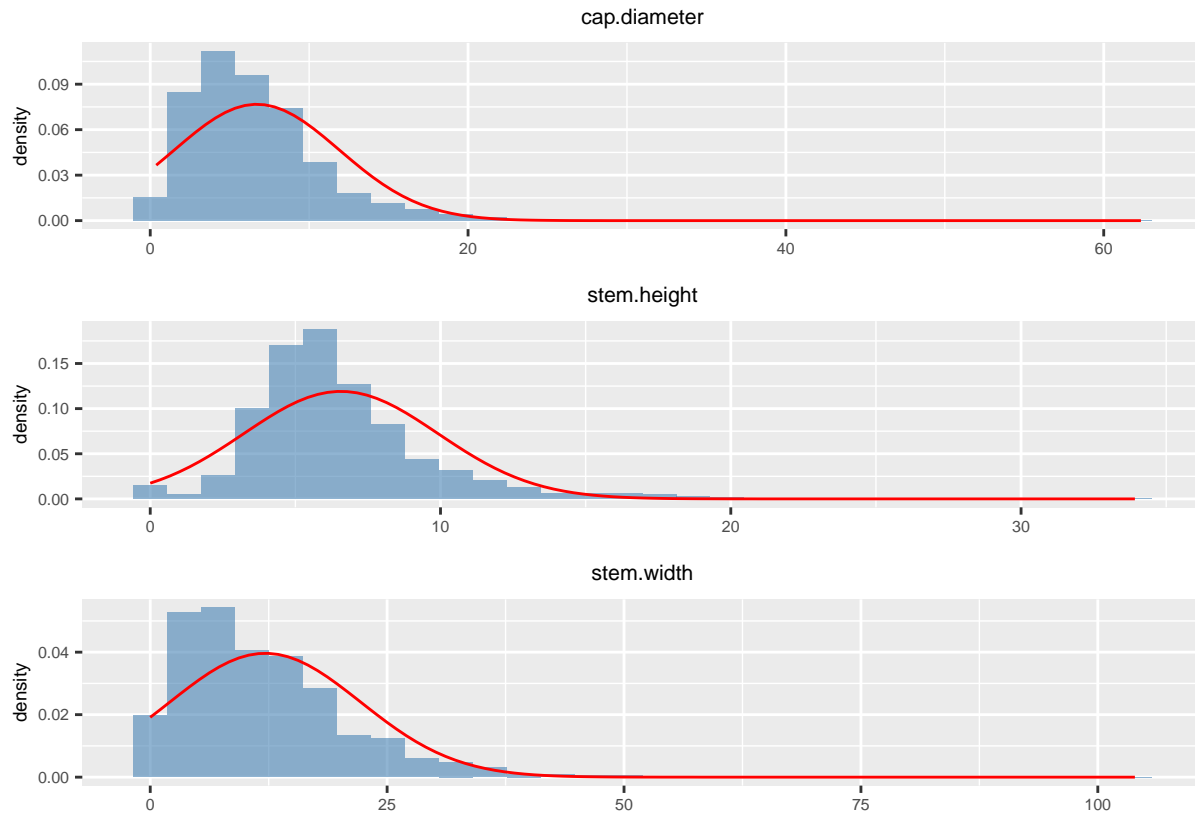
## Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
## i Please use `linewidth` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.

p2 <- create_histogram_with_normal(mushrooms_renamed, "stem.height", "steelblue")
p3 <- create_histogram_with_normal(mushrooms_renamed, "stem.width", "steelblue")

# Combinar los histogramas en una sola visualización
combined_plot <- p1 / p2 / p3

print(combined_plot)

```



Por último nos interesa analizar la correlación entre las variables, principalmente para 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.

```
#install.packages("PerformanceAnalytics")
library("PerformanceAnalytics")

## Loading required package: xts
## Loading required package: zoo

##
## Attaching package: 'zoo'

## The following objects are masked from 'package:base':
##
##   as.Date, as.Date.numeric

##
## ##### Warning from 'xts' package #####
## #
## # The dplyr lag() function breaks how base R's lag() function is supposed to #
## # work, which breaks lag(my_xts). Calls to lag(my_xts) that you type or #
## # source() into this session won't work correctly. #
## #
## # Use stats::lag() to make sure you're not using dplyr::lag(), or you can add #
## # conflictRules('dplyr', exclude = 'lag') to your .Rprofile to stop #
## # dplyr from breaking base R's lag() function. #
## #
```

```
## # Code in packages is not affected. It's protected by R's namespace mechanism #
## # Set `options(xts.warn_dplyr_breaks_lag = FALSE)` to suppress this warning. #
## #
## #####
##
## Attaching package: 'xts'
##
## The following objects are masked from 'package:dplyr':
##
##     first, last
##
## Attaching package: 'PerformanceAnalytics'
##
## The following object is masked from 'package:graphics':
##
##     legend
#chart.Correlation(numeric_columns, pch=19)
```

Ejercicio 3: Construcción de un árbol de decisión básico

```
mushrooms_renamed$cap.shape <- as.factor(mushrooms_renamed$cap.shape)
mushrooms_renamed$cap.surface <- as.factor(mushrooms_renamed$cap.surface)
mushrooms_renamed$cap.color <- as.factor(mushrooms_renamed$cap.color)
mushrooms_renamed$gill.attachment <- as.factor(mushrooms_renamed$gill.attachment)
mushrooms_renamed$gill.spacing <- as.factor(mushrooms_renamed$gill.spacing)
mushrooms_renamed$stem.root <- as.factor(mushrooms_renamed$stem.root)
mushrooms_renamed$ring.type <- as.factor(mushrooms_renamed$ring.type)
mushrooms_renamed$veil.type <- as.factor(mushrooms_renamed$veil.type)
mushrooms_renamed$habitat <- as.factor(mushrooms_renamed$habitat)
mushrooms_renamed$season <- as.factor(mushrooms_renamed$season)
mushrooms_renamed$class <- as.factor(mushrooms_renamed$class)
mushrooms_renamed$does.bruise.or.bleed <- as.factor(mushrooms_renamed$does.bruise.or.bleed)
mushrooms_renamed$has.ring <- as.factor(mushrooms_renamed$has.ring)

library(dplyr)
set.seed(42) #semilla para replicabilidad

n <- nrow(mushrooms_renamed)
indices <- sample(1:n) #'mezclamos' primero el orden de las filas, para luego solo tomar de la 1:70%, 70%
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 `minsplit/3` (si `minsplit` también se deja por defecto, sería casi 7). Mientras más bajo sea mayor será la specificity, es decir la proporción de negativos identificados correctamente como tal. Sin embargo, 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`.

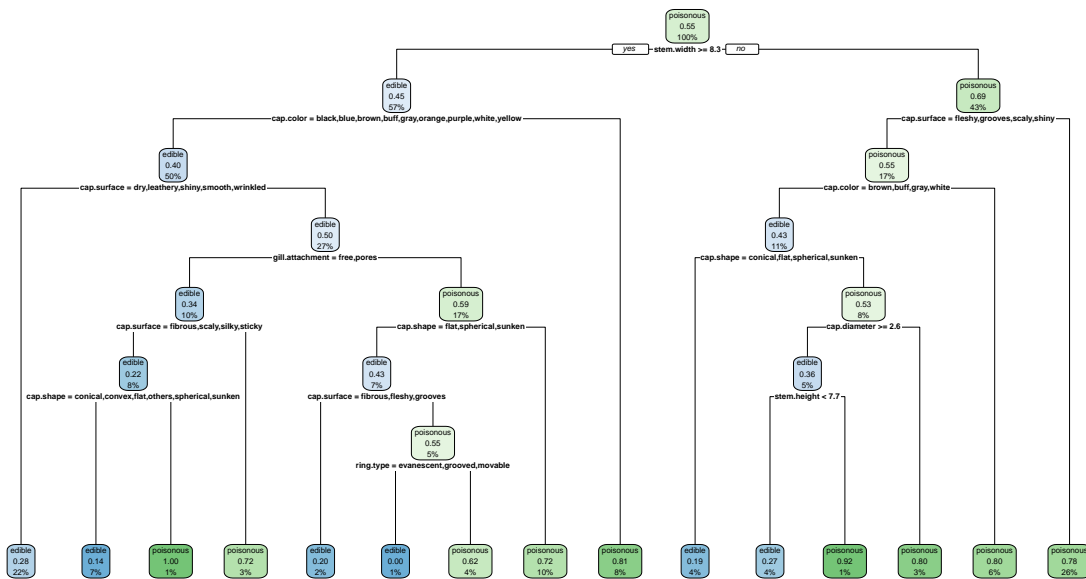
Estructura del árbol binario

```
library(rpart)

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

library(rpart.plot)

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 otros niveles para afinar aun más 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. #REVISAR CORR

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.

```
# Instalacion y/o importe de las librerias
# install.packages('caret')
library(caret)
```

```
## Loading required package: lattice
```

```
# Predicciones de probabilidades y de clases
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
```

```
print(t(conf_matrix$table)) # traspongo para que sea como visto en clase
```

```
##           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.

$$\begin{aligned}
 Accuracy &= \frac{TP + TN}{P + N} = \\
 &= \frac{TP + TN}{TP + FN + FP + TN} = \\
 &= \frac{TP + TN}{TP + FN + FP + TN} = \\
 &= \frac{2075 + 3215}{2075 + 1008 + 613 + 3215} = \\
 &= \frac{5290}{6911} = 0.7654464
 \end{aligned}$$

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

```
#
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
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.

$$\begin{aligned} Precision_{edible} &= \frac{TP}{TP + FP} = \\ &= \frac{2075}{2075 + 613} = 0.7719494 \end{aligned}$$

```
Precision <- TP/(TP + FP)
Precision
```

```
## [1] 0.7719494

# Validacion
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.

$$\begin{aligned} Recall_{edible} &= \frac{TP}{TP + FN} = \\ &= \frac{2075}{2075 + 1008} = 0.6730457 \end{aligned}$$

En R:

```
Recall <- TP/(TP + FN)
print(paste("Recall: ", Recall))
```

```
## [1] "Recall: 0.673045734674019"
```

```
# Validation
```

```
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

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

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 TRP 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.

```
# Install PRROC package if not already installed
```

```
# install.packages("PRROC")
```

```
library(PRROC)
```

```
# Calculate ROC curve using PRROC
```

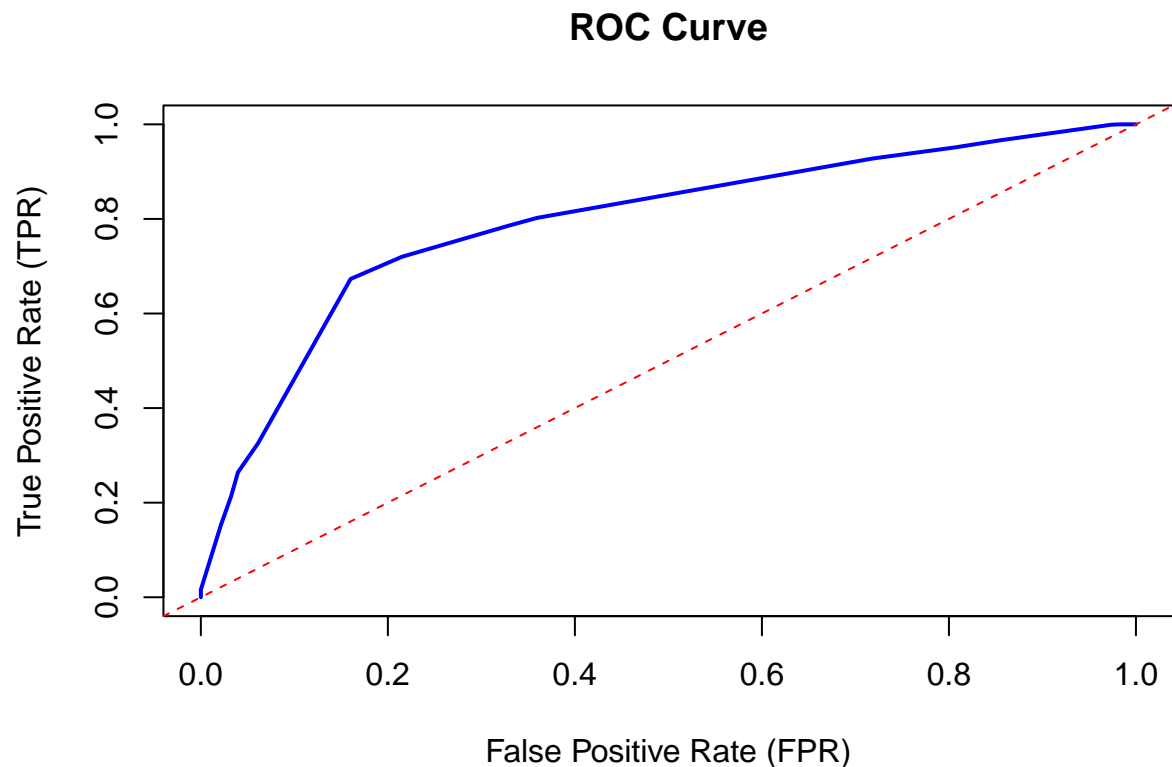
```
roc_obj <- roc.curve(scores.class0 = pred_prob[, "edible"],
                     weights.class0 = test_set$class == "edible",
                     curve = TRUE)
```

```
# Plot the ROC curve
```

```
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")
```

```
# Diagonal line representing random classifier
```

```
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.

Ejercicio 5: Optimización del modelo

```
library(pROC)
```

```
## Type 'citation("pROC")' for a citation.
```

```
##
```

```
## Attaching package: 'pROC'
```

```
## The following objects are masked from 'package:stats':
```

```
##
```

```
##      cov, smooth, var
```

```
opt_hiperparams <- function(train_data, valid_data, maxdepth_set, minsplit_set, minbucket_set) {
  # Initialize a data frame to store results
  results <- data.frame(maxdepth = integer(),
                        minsplit_val = integer(),
                        minbucket_val = integer(),
```

```

        auc = numeric())

# Loop through all combinations of hyperparameters
for (depth in maxdepth_set) {
  for (split in minsplit_set) {
    for (bucket in minbucket_set) {
      # Adjust the model
      tree_model_modified <- rpart(class ~ .,
                                data = train_data,
                                method = "class",
                                control = rpart.control(minbucket = bucket,
                                                         minsplit = split,
                                                         maxdepth = depth,
                                                         cp = 0,
                                                         xval = 0))

      # Predict probabilities on the validation set
      pred_probs <- predict(tree_model_modified, newdata = valid_data, type = "prob")[, 2]

      # Calculate 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)
}

```

VISUALIZACION!!!!

```

maxdepth_options <- c(3, 5, 7, 9, 11)
minsplit_options <- c(1000, 3000, 6000, 10000, 20000) #si ponemos 1000 como minimo elige eso (y maxdepth)
minbucket_options <- c(50, 100, 500, 1000, 5000)

```

```

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

```

```

new_model <- rpart(class ~ .,
                  data = train_set,
                  method = "class",
                  minsplit = best_minsplit,
                  minbucket = best_minbucket,
                  maxdepth = best_maxdepth,
                  cp = 0,
                  xval = 0)

```

```

# Hacer predicciones en el conjunto de testeo
test_predictions <- predict(new_model, newdata = test_set, type = "prob")[,2]

# Calcular el AUC-ROC
roc_curve <- roc(test_set$class, test_predictions)

## Setting levels: control = edible, case = poisonous
## Setting direction: controls < cases
auc_value <- auc(roc_curve)
auc_value <- roc_curve$auc

print(paste("Max depth *: ",best_maxdepth))

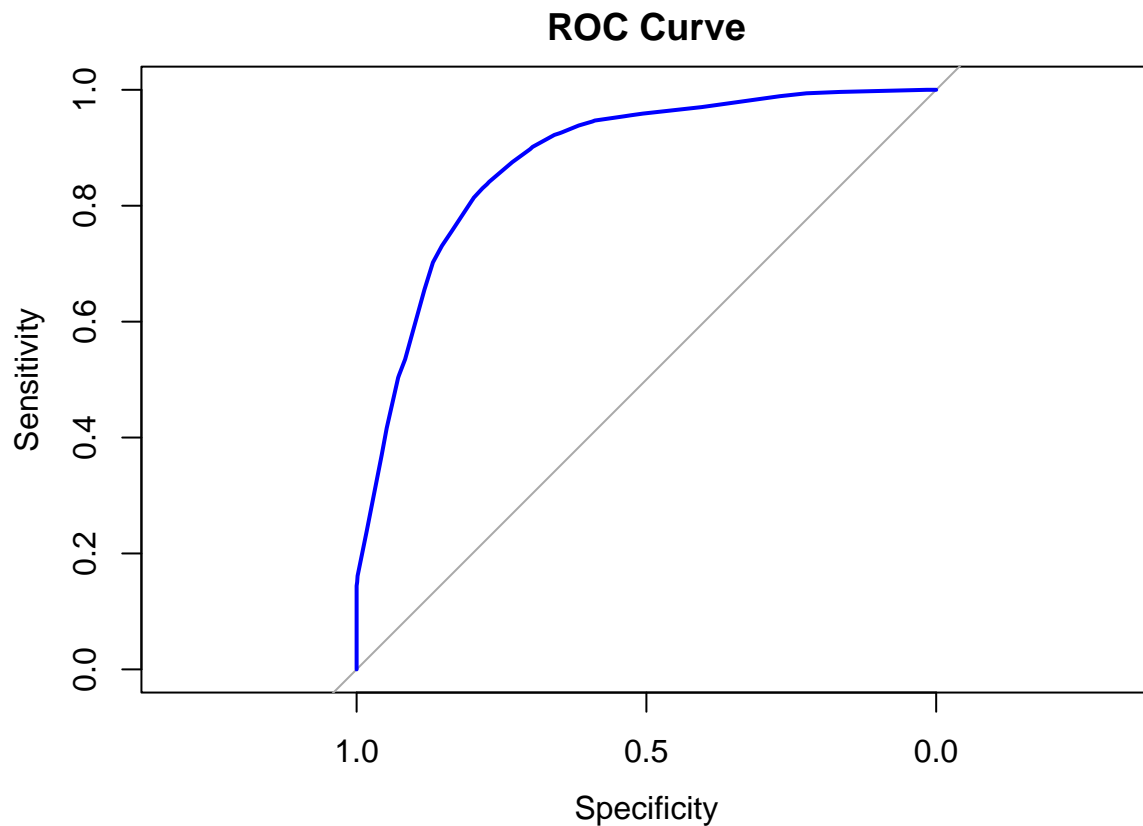
## [1] "Max depth *: 9"
print(paste("Min bucket *: ",best_minbucket))

## [1] "Min bucket *: 50"
print(paste("Min split *: ",best_minsplit))

## [1] "Min split *: 1000"
library(rpart.plot)
rpart.plot(new_model)

## Warning: labs do not fit even at cex 0.15, there may be some overplotting

```

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

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

```
## [1] "Matriz de Confusión:"
```

```
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 the metrics
```

```
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

Los primeros cortes del árbol optimizado son iguales a los del árbol original. Los nuevos cortes se generan en parte porque se repiten categorías ya usadas a lo largo del árbol ('cap.color', 'cap.shape'). Pero además se utilizan variables que antes no habían aparecido, como 'does.bruise.bleed', 'habitat' o 'season'. Estas se usan a partir del cuarto nivel del árbol e incluso más abajo, indicando que no eran variables tan influyentes en la predicción como las ya usadas, pero igualmente ayudaron en su conjunto a mejorar la predicción.

Ejercicio 7: Análisis del impacto de los valores faltantes

```
add_na_to_df <- function(df, na_fraction) {  
  # Seleccionar las columnas predictoras  
  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)  
}  
  
veinte_train <- add_na_to_df(train_set, 0.20)  
cincuenta_train <- add_na_to_df(train_set, 0.50)  
setentaicinco_train <- add_na_to_df(train_set, 0.75)  
  
veinte_valid <- add_na_to_df(valid_set, 0.20)  
cincuenta_valid <- add_na_to_df(valid_set, 0.50)  
setentaicinco_valid <- add_na_to_df(valid_set, 0.75)  
  
veinte_test <- add_na_to_df(test_set, 0.20)  
cincuenta_test <- add_na_to_df(test_set, 0.50)  
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"
```

Revisamos que efectivamente incorporamos estos datos correctamente:

```
# Verificar los resultados con algunas columnas samples
summary(veinte_train$cap.diameter)["NA's"]
```

```
## NA's
```

```
## 6450
```

```
summary(cinuenta_train$gill.attachment)["NA's"]
```

```
## NA's
```

```
## 16124
```

```
summary(setentaicinco_train$veil.type)["NA's"]
```

```
## NA's
```

```
## 24186
```

Ejercicio 8: Conclusiones y discusión