Paquetes de R para la bioinformática

PID_00293471

Marta Casals Fontanet
Alícia Vila Grifo

Tiempo mínimo de dedicación recomendado: 4 horas

Universitat Oberta de Catalunya

Marta Casals Fontanet

Estadística de formación y profesión. Licenciada en ITM. Máster en Educación y nuevas tecnologías. Profesora de Matemáticas en secundaria y universidad. Profesora colaboradora en el máster de Bioinformática y bioestadística de la UOC y la UB.

Alicia Vila Grifo

Licenciada en Matemáticas por la Universidad de Valencia y profesora consultora de Probabilidad, Estadística y Análisis de Datos en diferentes estudios de la UOC. Actualmente es profesora funcionaria de Informática en el Departamento de Educación de la Generalitat de Cataluña, en los ámbitos de programación y bases de datos. También participa en la coordinación de proyectos de aplicaciones web y de análisis de datos.

El encargo y la creación de este recurso de aprendizaje UOC han sido coordinados por los profesores: Álvaro Leitao Rodríguez y David Cantón Fabà

Cómo citar este recurso de aprendizaje con el estilo Harvard:
Casals, M. y Vila A. (2024). *Paquetes de R para la bioinformática.* [Recurso de aprendizaje textual]. 1.a ed. Fundació Universitat Oberta de Catalunya (FUOC).

Primera edición: febrero 2024

Av. Tibidabo, 39-43, 08035 Barcelona
Autoría: Marta Casals Fontanet, Alícia Vila Grifo
Producción: FUOC

Índice

Índice 3
1. Introducción al LAB6 4
2. Instrucciones del LAB6 4
3. Contenidos del LAB6 4
4. Ecosistema de R: Entornos reproducibles y visualización de datos con Shiny 4
5. Bioconductor, un proyecto de código abierto para bioinformática 14
6. Aplicación al ámbito biosanitario de algunos paquetes relevantes de R 22
6.1. El paquete caret 23
6.2. Paquete Hmisc 26
6.3. Paquete igraph 29
7. Ejercicios y casos prácticos con R 31
Solución a los ejercicios propuestos y casos prácticos con R 34
Información adicional 55

1. Introducción al LAB6

El **LAB6** es un recurso didáctico complementario de la asignatura **Software para el análisis de datos (SAD)** del máster interuniversitario de Bioinformática y Bioestadística de la Universitat Oberta de Catalunya (UOC) y la Universidad de Barcelona (UB).

Este **LAB6** forma parte de un conjunto de laboratorios prácticos que conjugan contenidos teóricos con ejemplos, ejercicios y casos prácticos reales del ámbito de conocimiento del máster.

En el **LAB6** se trabaja la herramienta Shiny, utilizada para generar aplicaciones web sencillas y amigables para el tratamiento y visualización de datos, y también una de las bibliotecas más importantes de **R** aplicadas al ámbito biosanitario, Bioconductor.

2. Instrucciones del LAB6

El **LAB6** contiene una serie de apartados con introducciones teóricas, ejemplos y casos prácticos, además de otros ejercicios que se propondrán para que sean resueltos por el estudiante. La temporización y las pautas para trabajar el **LAB6** serán indicadas en el aula de la asignatura. Del **LAB6** se presentará una propuesta de solución orientativa que servirá para que el estudiante pueda autoevaluar las soluciones realizadas por él mismo.

Todos los ejercicios se realizan en el entorno de desarrollo integrado para **R**, **RStudio** (https://cran.rstudio.com/), como lenguaje de programación para la informática, estadística y los gráficos.

3. Contenidos del LAB6

En este **LAB6** trabajaremos los siguientes contenidos:

- Ecosistema de **R**: Entornos reproducibles y visualización de datos con Shiny.
- Bioconductor, un proyecto de código abierto para bioinformática.
- Aplicación al ámbito biosanitario de algunos paquetes relevantes de **R**.
- Ejercicios y casos prácticos con **R**.
- Soluciones de los ejercicios y casos prácticos con **R**.

4. Ecosistema de R: Entornos reproducibles y visualización de datos con Shiny

Tal y como hemos ido viendo a lo largo de los diferentes **LAB** de esta asignatura, **R** es un lenguaje de programación con entornos de software asociados que se utiliza en estadística y análisis de datos con amplia aplicación a diferentes ámbitos de conocimiento.

RStudio es el entorno de desarrollo integrado (IDE) de **R** que ofrece características como la gestión de paquetes, el tratamiento y gestión de datos, la visualización de gráficos y la programación con **R** en un entorno amigable.

La herramienta **RMarkdown** facilita la creación de documentos dinámicos que combinan código **R**, resultados de la ejecución de dicho código y texto, y todo junto permite la narrativa del estudio en un solo documento. Este formato es el más utilizado para la realización de informes y presentaciones reproducibles. No desarrollaremos el uso de esta herramienta, ya que se trata, extensamente, en el **LAB1** de esta asignatura.

Por otra parte, y es lo que nos ocupa en este apartado, existen herramientas de visualización de datos como **Shiny** que permiten la creación de sencillas aplicaciones web interactivas y paneles de control directamente desde **R**. Shiny es un paquete que se puede utilizar tanto desde **R** como desde otros lenguajes de programación como Python. Una de las particularidades de Shiny es que permite convertir un análisis de datos en aplicaciones web interactivas sin que sea necesario tener conocimientos específicos de HTML, CSS o Javascript. Una de las referencias web más importantes para profundizar en los contenidos y aplicaciones de Shiny es https://shiny.posit.co/.

Para empezar a trabajar con Shiny, el primer paso es instalar el paquete desde ***Tools > Install Packages > Shiny***. Es importante, tener activada la opción de *Install dependencies*, por si la instalación de este paquete requiere otros. También podemos ejecutar el siguiente bloque de código:

```
if (!requireNamespace("shiny", quietly = TRUE))

install.packages("shiny")
```

Para entender el funcionamiento general de Shiny, es interesante conocer los siguientes aspectos:

- En la estructura de una aplicación Shiny hay dos componentes clave: el UI (interfaz de usuario) y el *server* (servidor). El UI define la interfaz de usuario, es decir, cómo se mostrará la aplicación en el navegador web, mientras que el *server* maneja la lógica y las acciones que realiza la aplicación.
- UI (interfaz de usuario): El UI se crea utilizando funciones de Shiny que definen la apariencia de la aplicación web, pudiendo incluir botones, gráficos o cuadros de texto, como en cualquier aplicación web. El UI se define en un archivo **R** separado o directamente en el *script* **R**.
- *Server*: El *server* contiene el código **R** que se ejecuta en respuesta a las acciones del usuario (cálculos, representación de gráficos, etc.).
- Una aplicación Shiny es *interactiva*, es decir, detecta automáticamente cambios en los valores de entrada y actualiza dinámicamente la interfaz de usuario y los resultados en función de esos cambios, lo cual permite a los usuarios interactuar con la aplicación.

- Una vez creada la aplicación Shiny, puede *desplegarse* de diversas maneras y ejecutarse localmente o en otras aplicaciones que sean accesibles para todos los usuarios.

En los ejemplos que veremos a continuación, iremos introduciendo las funciones y los pasos más importantes para realizar una aplicación sencilla en Shiny.

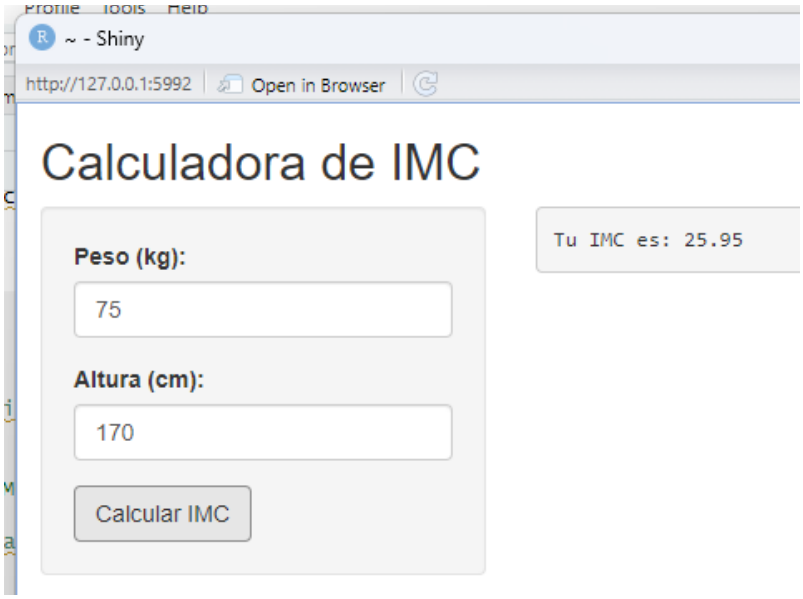
Ejemplo 1. Aplicación Shiny que calcula IMC

Supongamos que queremos calcular el valor del índice de masa corporal (IMC) en función del peso. Para ello, definimos unos parámetros básicos de peso y altura. En el siguiente ejemplo vemos cómo definir una aplicación Shiny:

```
# Cargamos el paquete Shiny
library(shiny)
# Definimos la interfaz de usuario (UI) con la función fluidPage()
ui <- fluidPage(
  # Título de la aplicación
  titlePanel("Calculadora de IMC"),

  # Entrada de diferentes datos de tipo numérico con numericInput()
  sidebarLayout(
    sidebarPanel(
      numericInput("peso", "Peso (kg):", value = 70, min = 0, max = 200),
      numericInput("altura", "Altura (cm):", value = 170, min = 0, max = 300),
      actionButton("calcular", "Calcular IMC")
    ),
    # Resultados panel principal
    mainPanel(
      verbatimTextOutput("resultado")
    )
  )
)
# Definimos los parámetros del servidor
```

```
server <- function(input, output) {
  # Botón de cálculo
  observeEvent(input$calcular, {
    # Calcular el IMC
    peso_kg <- input$peso
    altura_m <- input$altura / 100
    imc <- peso_kg / (altura_m^2)

    # Determinamos la categoría del IMC
    categoria <- ifelse(imc < 18.5, "Bajo peso",
                ifelse(imc < 24.9, "Peso normal",
                ifelse(imc < 29.9, "Sobrepeso", "Obeso")))

    # Mostramos el resultado final
    output$resultado <- renderText({
      paste("Tu IMC es:", round(imc, 2) ,categoría)
    })
  })
}

# Creamos la aplicación Shiny
shinyApp(ui = ui, server = server)
```

Veremos que, al ejecutar el anterior bloque de código, obtenemos una nueva ventana con la siguiente interfaz:



Podemos ver cómo, cambiando el valor del peso y la altura, obtenemos diferentes resultados en cuanto al valor del IMC y la categoría asociada.

Del código anterior, podemos destacar las siguientes funciones básicas:

- Creación de la interfaz de usuario con *fluidPage()* y las distintas funciones que permiten definir el título y otros elementos como botones o cuadros de texto.
- Definición del *server*.
- Creación de la aplicación: *shinyApp(ui = ui, server = server)* que integra la interfaz del usuario con el *server*.

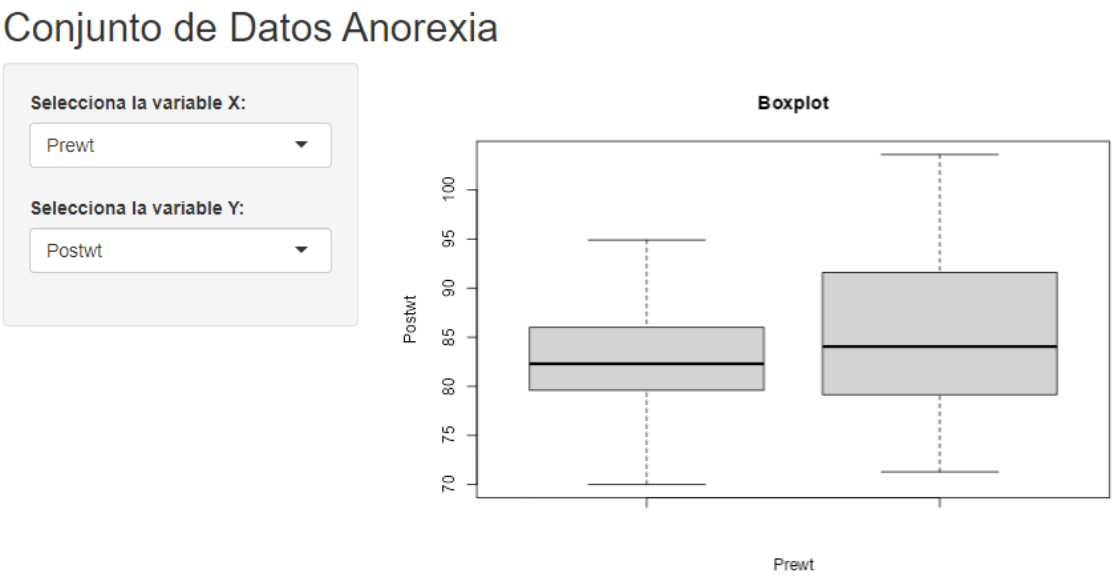
Ejemplo 2. Aplicación Shiny a partir de un conjunto de datos

En el siguiente ejemplo, utilizaremos el conjunto de datos *anorexia* del paquete *MASS* para realizar una sencilla aplicación Shiny para mostrar el diagrama de cajas de dos variables elegidas del conjunto de datos *anorexia*; contiene información acerca de un tratamiento realizado sobre un conjunto de pacientes.

```
## Cargamos los paquetes Shiny y MASS

library(shiny)

library(MASS)


# Cargamos el conjunto de datos "anorexia"

data("anorexia")
```

```
# Definimos la interfaz de la aplicación Shiny
ui <- fluidPage(
  titlePanel("Conjunto de datos anorexia"),
  sidebarLayout(
    sidebarPanel(
      # Definimos el selector de variables x e y que utilizaremos en el
gráfico
      selectInput("x", "Selecciona la variable X:", choices =
colnames(anorexia)),
      selectInput("y", "Selecciona la variable Y:", choices =
colnames(anorexia))
    ),
    mainPanel(
      # Dibujamos el boxplot
      plotOutput("boxplot")
    )
  )
)
# Definimos los parámetros del servidor
server <- function(input, output) {
  # Creamos el gráfico de dispersión
  output$boxplot <- renderPlot({
    x <- input$x
    y <- input$y
    boxplot(anorexia[, x], anorexia[, y], xlab = x, ylab = y, main =
"Boxplot")
  })
}
# Ejecutamos la aplicación Shiny
shinyApp(ui = ui, server = server)
```

Por ejemplo, podemos ver la comparativa de los diagramas de cajas antes y después del período de estudio:



En este ejemplo, es interesante destacar el uso de las siguientes funciones:

selectInput() nos permite a nivel de interfaz de usuario, seleccionar las variables del conjunto de datos.

El bloque de código siguiente (en la parte del *server*) nos permite mostrar el gráfico a partir de las variables seleccionadas:

```
output\(boxplot <- renderPlot({ x <- input\)x y <- input$y boxplot(anorexia[,
x], anorexia[, y], xlab = x, ylab = y, main = "Boxplot") })
```

Es interesante realizar ejemplos similares para diferentes conjuntos de datos y diferentes tipos de gráficos.

A continuación, veremos un ejemplo donde se muestran los resultados de un determinado análisis a partir de un conjunto de datos externo.

Ejemplo 3. Explorador de datos

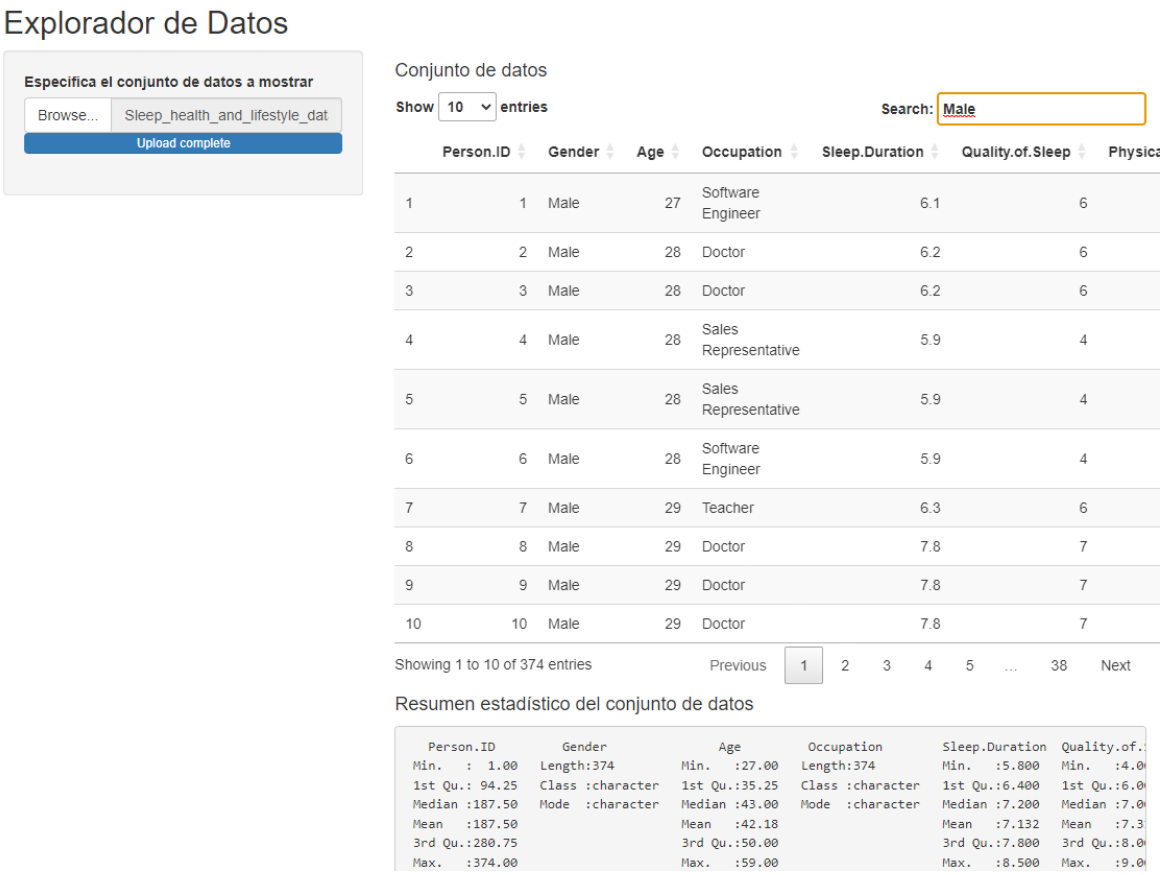
El siguiente ejemplo ilustra cómo, mediante una interfaz gráfica, podemos cargar un determinado archivo en formato .csv y mostrar el conjunto de datos, pudiendo realizar algunas búsquedas por palabras clave y un resumen estadístico.

Veamos el código correspondiente:

```
library(shiny)


#Comprobamos la instalación del paquete DT que nos permite definir tablas
dinámicas
if (!requireNamespace("DT", quietly = TRUE))
  install.packages("DT")


library(DT)
# Definimos la interfaz de la aplicación
ui <- fluidPage(
  titlePanel("Explorador de Datos"),
  sidebarLayout(
    sidebarPanel(
      fileInput("file", "Especifica el conjunto de datos a mostrar")
#lectura del csv
    ),
    mainPanel(
      DTOutput("table"), #muestra formato de tabla
      h4("Resumen estadístico del conjunto de datos"),
      verbatimTextOutput("resumen")
    )
  )
)


# Definimos el servidor de la aplicación
server <- function(input, output) {
  # Cargamos el conjunto de datos
  dataset <- reactive({
```

```
    req(input$file)
    read.csv(input$file$datapath)
  })

  # Mostramos la tabla dinámica
  output$table <- renderDT({
    datatable(dataset())
  })
output$resumen <- renderPrint({
  tryCatch({
    summary(dataset())
  }, error = function(e) {
    "Para mostrar el resultado, es necesario cargar previamente el archivo
de datos"
  })
})
}
# Ejecutar la aplicación Shiny
shinyApp(ui, server)
```

Para ejecutar el ejemplo anterior, es necesario que tengáis localizado un conjunto de datos en formato archivo .csv para poder importarlo. La siguiente captura ilustra el ejemplo ejecutado a partir de un conjunto de datos obtenido de https://www.kaggle.com/ que contiene los datos correspondientes a los hábitos de sueño y cómo afectan a la salud. Se ha descargado un .csv (*Sleep_health_and_lifestyle_dataset.csv*) en una carpeta local, a partir de: https://www.kaggle.com/datasets/uom190346a/sleep-health-and-lifestyle-dataset/:



En este ejemplo, cabe destacar el uso de la función *fileInput()*, que permite la búsqueda de un determinado archivo en cuanto a interfaz junto a:

```
dataset <- reactive({ req(input\(file) read.csv(input\)file$datapath) })
```

en la parte del *server* que permite mostrar el conjunto de datos.

Para ejecutar esta sencilla aplicación, podéis guardar el archivo como *app.R* en un directorio de trabajo de **R** y ejecutar *shiny::runApp()*.

Es interesante que realicéis este ejemplo a partir de diferentes archivos, así como ampliar el uso de diferentes parámetros en las funciones utilizadas para cambiar el formato de salida, el color de texto, etc. Igualmente, a partir de los ejemplos anteriormente trabajados, que dan una visión básica de cómo trabajar con Shiny, es importante profundizar en ello con la herramienta y trabajar distintas propiedades y aplicaciones.

En http://datascience.recursos.uoc.edu/es/shiny/ podéis encontrar más recursos para trabajar con Shiny.

5. Bioconductor, un proyecto de código abierto para bioinformática

En este apartado nos centraremos en la herramienta **Bioconductor** para el análisis y la visualización de datos aplicada a la genómica y biología computacional. Está diseñada específicamente para el lenguaje de programación **R** y es un entorno de software estadístico y de gráficos muy utilizado en la comunidad científica. Bioconductor facilita el análisis de datos biológicos y proporciona paquetes de software que incluyen bases de datos biológicos, métodos estadísticos, herramientas de visualización y algoritmos de aprendizaje automático.

Es importante destacar que es una herramienta de código abierto con una comunidad formada por desarrolladores que contribuyen a la mejora del programa.

En el siguiente enlace, https://www.bioconductor.org/, podéis profundizar en el uso de la herramienta. En este apartado solo trataremos los conceptos más importantes del uso de Bioconductor y algunas aplicaciones.

Para empezar a trabajar con Bioconductor, el primer paso es la instalación de dicha herramienta:

```
if (!require("BiocManager", quietly = TRUE))

    install.packages("BiocManager")

## Warning: package 'BiocManager' was built under R version 4.3.2

BiocManager::install()

## Bioconductor version 3.18 (BiocManager 1.30.22), R 4.3.1 (2023-06-16 ucrt)

## Installation paths not writeable, unable to update packages

##   path: C:/Program Files/R/R-4.3.1/library

##   packages:

##       foreign, KernSmooth, lattice, Matrix, mgcv, nlme, rpart, spatial,
survival

## Old packages: 'dplyr', 'htmltools', 'rlang', 'RSQLite', 'shiny', 'stringi',

##   'vctrs', 'xfun'
```

Para conocer más detalles de la instalación, podéis consultar en https://www.bioconductor.org/install/. Si queremos conocer cuáles son los paquetes que forman parte de Bioconductor, ejecutamos el siguiente código:

```
if (!requireNamespace("BiocManager", quietly = TRUE))

    install.packages("BiocManager")
```

```
# Listamos todos los paquetes de Bioconductor
bioc_pkgs <- BiocManager::available()

# Imprimimos la lista de paquetes
print(head(bioc_pkgs))
## [1] "A3"        "a4"        "a4Base"    "a4Classif" "a4Core"
"a4Preproc"
```

Ejemplo 4. Bioconductor utilizando el paquete GenomicFeatures

El paquete *GenomicFeatures* está integrado en Bioconductor y contiene herramientas para el análisis de datos genómicos. En el siguiente ejemplo vamos a ver cómo, utilizando el paquete *GenomicFeatures*, creamos un conjunto de genes ficticio y mostramos las características correspondientes.

```
if (!requireNamespace("GenomicFeatures", quietly = TRUE)) {
  if (!requireNamespace("BiocManager", quietly = TRUE))
    install.packages("BiocManager")
  BiocManager::install("GenomicFeatures")
}

# Cargar el paquete GenomicFeatures
library(GenomicFeatures)
# Crear un ejemplo de conjunto de genes
genes <- data.frame(
  seqnames = c("chr1", "chr2", "chr1"),
  start = c(100, 200, 300),
  end = c(200, 300, 400),
  gene_id = c("Gene1", "Gene2", "Gene3"),
  stringsAsFactors = FALSE
)
```

```
# Crear un objeto `GRanges` que representa la información de los genes
gr <- GRanges(seqnames = genes$seqnames, ranges = IRanges(start =
genes$start, end = genes$end), id = genes$gene_id)


# Mostrar la información de los genes
print(gr)
## GRanges object with 3 ranges and 1 metadata column:
##       seqnames    ranges strand |          id
##          <Rle> <IRanges>  <Rle> | <character>
##   [1]     chr1   100-200      * |       Gene1
##   [2]     chr2   200-300      * |       Gene2
##   [3]     chr1   300-400      * |       Gene3
##   -------
##   seqinfo: 2 sequences from an unspecified genome; no seqlengths
```

GRanges() es un objeto utilizado para representar y manipular rangos genómicos.

Ejemplo 5. Bioconductor utilizando el paquete DESeq2

El paquete *DESeq2* se utiliza para realizar análisis diferencial en las expresiones génicas. El análisis diferencial de expresión génica trata de identificar genes cuya expresión varía entre diferentes condiciones experimentales, con el objetivo de detectar cambios en los genes en respuesta a diferentes estímulos o condiciones.

En primer lugar, instalamos el paquete:

```
if (!requireNamespace("DESeq2", quietly = TRUE)) {
  if (!requireNamespace("BiocManager", quietly = TRUE))
    install.packages("BiocManager")
  BiocManager::install("DESeq2")
}
library(DESeq2)
```

Una vez instalado el paquete, generamos datos aleatorios a partir de un determinado tipo de distribución, por ejemplo, una distribución de Poisson:

```
# Generamos datos simulados

set.seed(123)

matriz_expresion <- matrix(rpois(1200, lambda = 5), ncol = 6)

colnames(matriz_expresion) <- c("Muestra1", "Muestra2", "Muestra3",
"Control1", "Control2", "Control3")



# Creamos un objeto DESeqDataSet

dds <- DESeqDataSetFromMatrix(countData = matriz_expresion,

     colData = data.frame(cond = rep(c("Muestra", "Control"), each = 3)),
design = ~ cond)



# Realizamos el análisis de expresión diferencial

dds <- DESeq(dds)

## estimating size factors

## estimating dispersions

## gene-wise dispersion estimates

## mean-dispersion relationship

## final dispersion estimates

## fitting model and testing

# Obtenemos resultados

resultados <- results(dds)

# Mostramos los resultados

print(resultados)

## log2 fold change (MLE): cond Muestra vs Control

## Wald test p-value: cond Muestra vs Control

## DataFrame with 200 rows and 6 columns

##      baseMean log2FoldChange     lfcSE      stat    pvalue      padj
```

```
##     <numeric>      <numeric> <numeric> <numeric> <numeric> <numeric>
## 1     4.63562       0.529732  0.802126  0.660411  0.508990  0.993232
## 2     5.42524       0.362026  0.732450  0.494267  0.621118  0.993232
## 3     4.01141       1.111367  0.862183  1.289015  0.197393  0.993232
## 4     4.96575       0.478917  0.769487  0.622385  0.533688  0.993232
## 5     4.79385       0.394723  0.786308  0.501996  0.615671  0.993232
## ...       ...            ...       ...       ...       ...       ...
## 196   4.76571       0.223330  0.778359  0.286924  0.774170  0.993232
## 197   4.64073      -0.282078  0.792819 -0.355791  0.721997  0.993232
## 198   3.48490       0.286955  0.895140  0.320570  0.748537  0.993232
## 199   5.28881      -0.604062  0.771778 -0.782689  0.433810  0.993232
## 200   4.17105       0.936548  0.894291  1.047252  0.294983  0.993232
```

El anterior resultado muestra la expresión génica diferencial entre las condiciones «Muestra» y «Control». La columna baseMean es la expresión media del gen en todas las muestras. Es una medida de la cantidad en promedio del gen. *log2FoldChange* explica el cambio en la expresión génica entre las condiciones comparadas. Un valor positivo representa 'sobreexpresión' en «Muestra» en comparación con «Control», mientras que un valor negativo sugiere 'subexpresión'. En este caso, los valores son pequeños, lo que representaría cambios sutiles. lfcSE (*log2FoldChange Standard Error*) proporciona la estimación del error estándar asociado con el log2FoldChange. stat es el estadístico de prueba Wald. Puede utilizarse para calcular el valor p. p-value es el valor p asociado con el estadístico de la prueba de Wald. Indica la probabilidad de observar el log2FoldChange, si la verdadera diferencia es cero. padj es el valor p ajustado para controlar el error tipo I debido a las pruebas múltiples. Se calcula utilizando otros métodos de ajuste.

Por lo tanto, lo que se busca es un valor para log2FoldChange significativo (con un p-value pequeño y un *padj* pequeño después de la corrección para múltiples pruebas), para identificar genes que se expresan de manera diferencial entre las condiciones. En este ejemplo, todos los genes tienen un valor de padj de 0.993232, lo que indica que no hay genes que de manera significativa estén diferencialmente expresados en las condiciones simuladas. Esto podría deberse a que los datos fueron generados de manera aleatoria y no hay una verdadera expresión diferencial simulada en este caso.

Nota. Para entender todos los conceptos, tanto a estadística como interpretativamente de los resultados en el ámbito biosanitario, es necesario profundizar más en contenidos que no son propios de esta asignatura. Solo se pretende dar una pincelada del uso de Bioconductor aplicando algunos de sus paquetes más relevantes.

Ejemplo 6. Uso de Bioconductor utilizando el paquete edgeR

El paquete edgeR está diseñado para el análisis de datos de expresión génica y es especialmente útil para la identificación de genes diferencialmente expresados en estudios de ARN-Seq (secuenciación de ARN) y otros experimentos de perfiles de expresión génica.

```
# Instalamos y cargamos los paquetes necesarios

if (!requireNamespace("edgeR", quietly = TRUE)) {

  if (!requireNamespace("BiocManager", quietly = TRUE))

    install.packages("BiocManager")

  BiocManager::install("edgeR")

}

library(edgeR)
```

Crearemos un conjunto de datos (muestras) de expresión simulados a partir de una distribución de Poisson y definiremos los dos grupos a los que pertenecen las muestras:

```
# Creamos datos ficticios

set.seed(123)

counts_matrix <- matrix(rpois(120, lambda = 5), ncol = 6)


# Definir grupos

group <- factor(rep(c("Grupo1", "Grupo2"), each = 3))
```

Creamos ahora un objeto DGEList (*Differential Gene Expression List*), que es una estructura de datos específica de edgeR para almacenar datos de expresión génica y metadatos asociados:

```
# Creamos un objeto DGEList

dge <- DGEList(counts = counts_matrix, group = group)


Finalmente, realizamos un análisis de expresión génica diferencial:

# Realizamos análisis de expresión génica diferencial

dge <- estimateDisp(dge)

## Using classic mode.
```

```
fit <- glmFit(dge, design = model.matrix(~group))
lrt <- glmLRT(fit)


# Mostramos los resultados
top_genes <- topTags(lrt)$table
print(top_genes)
##         logFC   logCPM        LR     PValue       FDR
## 7   0.9284868 16.26028 3.8920720 0.04851454 0.7132553
## 15  0.8809341 15.74572 2.1865200 0.13922356 0.7132553
## 16 -0.8000274 15.82586 1.9627954 0.16121467 0.7132553
## 13 -0.6421789 16.04270 1.5708155 0.21008863 0.7132553
## 6   0.6157170 15.93834 1.3147552 0.25153550 0.7132553
## 9   0.5392671 16.07586 1.1508055 0.28338059 0.7132553
## 10 -0.5629392 15.82589 0.9902237 0.31968770 0.7132553
## 18  0.5301519 15.90183 0.9470778 0.33046424 0.7132553
## 20 -0.5210542 15.90182 0.9146520 0.33888310 0.7132553
## 2  -0.4849941 15.97397 0.8497339 0.35662764 0.7132553
```

El análisis de expresión génica diferencial se ha realizado utilizando lo que se denomina modelo lineal general (GLM) y el test de razón de verosimilitud (LRT). Este análisis ayuda a identificar genes cuya expresión difiere significativamente de los dos grupos.

De los resultados obtenidos, la variable top_genes contiene la tabla con estadísticas para cada gen, incluyendo el logaritmo del *fold change* (*logFC*), el valor p (p-value) y el valor ajustado de p (FDR).

La interpretación de los resultados realizada para comparar la expresión génica diferencial de los dos grupos se centra en que los genes con valores más ajustados de p bajos (FDR) sugieren una mayor confianza en la significancia estadística de la diferencia de expresión.

Algunas de las funciones características son:

- estimateDisp se utiliza para realizar estimaciones de las dispersiones de los datos. Dicha variabilidad es clave en el análisis de la expresión génica diferencial.
- glmFit se utiliza para ajustar el modelo a los datos y obtener información sobre los coeficientes del modelo para cada gen.

- glmLRT realiza un test de razón de verosimilitudes (Likelihood Ratio Test) para comparar modelos y determinar si hay diferencias significativas en la expresión génica entre las condiciones especificadas en la matriz.
- topTags extrae los genes más relevantes o diferencialmente expresados según los resultados del test.

Ejemplo 7. Bioconductor con el paquete biomaRt

biomaRt es un paquete de Bioconductor que permite acceder a una base de datos que contiene datos biológicos (genes, proteínas, ontologías, etc.). A continuación, veamos un ejemplo que visualiza el contenido de la base de datos ensembl.

```
# Instalamos y cargamos el paquete biomaRt

if (!requireNamespace("biomaRt", quietly = TRUE)) {

  if (!requireNamespace("BiocManager", quietly = TRUE))

    install.packages("BiocManager")

  BiocManager::install("biomaRt")

}

library(biomaRt)

# Conectamos con la base de datos ensembl mediante la interfaz de BioMart

ensembl <- useMart("ensembl", dataset = "hsapiens_gene_ensembl")

# Obtenemos información sobre genes específicos (por ejemplo, los primeros 5
genes)

genes <- c("ENSG00000139618", "ENSG00000141510", "ENSG00000171862",
"ENSG00000137075", "ENSG00000139970")

# Especificamos los atributos a mostrar

attributes <- c("external_gene_name", "chromosome_name", "start_position",
"end_position")

# Realizamos la consulta

gene_info <- getBM(attributes, filters = "ensembl_gene_id", values = genes,
mart = ensembl)

# Mostramos los resultados

print(gene_info)

##   external_gene_name chromosome_name start_position end_position
```

```
## 1          RNF38          9       36336396      36487548
## 2          BRCA2         13       32315086      32400268
## 3           RTN1         14       59595976      59870776
## 4           TP53         17        7661779       7687538
## 5           PTEN         10       87862638      87971930
```

Algunas de las funciones destacadas del código anterior son:

useMart se utiliza para acceder a una base de datos y conjunto de datos específico. getBM es la función que utilizaremos para realizar una consulta sobre la anterior base de datos especificada.

Los ejemplos trabajados en este apartado pretenden dar una visión inicial de cómo trabajar con Bioconductor y cómo acceder a algunos paquetes interesantes. No obstante, el proyecto Bioconductor es mucho más extenso y se recomienda al estudiante profundizar en el uso de otros paquetes y aplicaciones.

6. Aplicación al ámbito biosanitario de algunos paquetes relevantes de R

A lo largo de los diferentes **LAB** hemos utilizado diversidad de paquetes de **R** para diferentes aplicaciones y como fuentes de datos para el desarrollo de diversos ejercicios. En este apartado, veremos algunos de los paquetes que se utilizan de forma más específica al ámbito biosanitario.

Como sabemos, **R** es ampliamente utilizado en la bioinformática debido a su flexibilidad, capacidad de análisis de datos y la existencia de numerosos paquetes especializados. Algunos de los tipos de análisis y paquetes asociados que podríamos mencionar son:

Análisis de datos de expresión génica

Como hemos visto en ejemplos en el anterior apartado, con **R** podemos realizar análisis de expresión génica diferencial utilizando paquetes como *DESeq2*, *edgeR* o *limma* y visualizar diferentes tipos de gráficos utilizando ggplot2.

Análisis de secuencias biológicas

Principalmente, Bioconductor proporciona paquetes como Biostrings para manipular y analizar secuencias biológicas (ADN, ARN, proteínas). La visualización de datos biológicos podría realizarse con los diferentes tipos de gráficos generados utilizando ggplot2. También ComplexHeatmap permitirá visualizar datos de estructuras más complejas.

Análisis de variantes genéticas

Algunos paquetes como VariantAnnotation y, nuevamente, paquetes de Bioconductor son útiles para el análisis de variantes genéticas a partir de datos de secuenciación.

Análisis de redes biológicas

Algunos paquetes como igraph permiten realizar análisis de redes biológicas y visualizaciones.

Análisis de datos de microarreglos

Paquetes como affy y limma son utilizados para analizar datos de microarreglos.

Para realizar aprendizaje automático (*machine learning*) en bioinformática, pueden utilizarse paquetes como caret y, para acceder a bases de datos biológicas, podemos utilizar paquetes como biomaRt, ya visto en ejemplos anteriores para obtener datos, como el *dataset* Ensembl.

Para desarrollar aplicaciones web interactivas, el paquete Shiny, ya trabajado en este **LAB**, permite compartir y visualizar resultados de análisis de datos biológicos.

Este breve resumen pretende dar una visión general de algunos ámbitos de aplicación de **R**. A continuación, no obstante, mostraremos el uso de algunos paquetes en el ámbito biosanitario.

6.1. El paquete caret

Este paquete ofrece herramientas para el entrenamiento y la evaluación de modelos predictivos en el ámbito biosanitario.

Ejemplo 8. Aplicación del paquete caret

Este ejemplo utiliza un modelo de k-NN para clasificar las especies del paquete de datos Iris. El paquete caret proporciona una interfaz unificada para entrenar modelos y realizar evaluaciones. En este caso, se utiliza la validación cruzada para evaluar el rendimiento del modelo, y se muestra la matriz de confusión como una métrica de evaluación.

Nota. - k-NN (k-Nearest Neighbors) es un algoritmo de aprendizaje automático supervisado.

```
#Instalamos el paquete caret

if (!requireNamespace("caret", quietly = TRUE))

    install.packages("caret")

library(caret)

## Loading required package: ggplot2
```

```
## Warning: package 'ggplot2' was built under R version 4.3.2

## Loading required package: lattice

# Cargamos el conjunto de datos iris

data(iris)


# Define el controlador de entrenamiento utilizando el método de
clasificación k-Nearest Neighbors (k-NN)

controlador <- trainControl(method = "cv", number = 5)  # Validación cruzada
con 5 repeticiones


# Entrena el modelo k-NN

model <- train(Species ~ ., data = iris, method = "knn", trControl =
controlador)


# Muestra el modelo entrenado

print(model)

## k-Nearest Neighbors

##

## 150 samples

##   4 predictor

##   3 classes: 'setosa', 'versicolor', 'virginica'

##

## No pre-processing

## Resampling: Cross-Validated (5 fold)

## Summary of sample sizes: 120, 120, 120, 120, 120

## Resampling results across tuning parameters:

##

##   k  Accuracy   Kappa

##   5  0.9666667  0.95
```

```
##   7  0.9733333  0.96
##   9  0.9733333  0.96
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was k = 9.
# Realiza predicciones en el conjunto de datos de prueba
predictions <- predict(model, newdata = iris)

# Muestra la matriz de confusión
confusion_matrix <- confusionMatrix(predictions, iris$Species)
print(confusion_matrix)
## Confusion Matrix and Statistics
##
##             Reference
## Prediction   setosa versicolor virginica
##   setosa         50          0         0
##   versicolor      0         48         1
##   virginica       0          2        49
##
## Overall Statistics
##
##                Accuracy : 0.98
##                  95% CI : (0.9427, 0.9959)
##     No Information Rate : 0.3333
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.97
##
```

```
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: setosa Class: versicolor Class: virginica
## Sensitivity                 1.0000            0.9600           0.9800
## Specificity                 1.0000            0.9900           0.9800
## Pos Pred Value              1.0000            0.9796           0.9608
## Neg Pred Value              1.0000            0.9802           0.9899
## Prevalence                  0.3333            0.3333           0.3333
## Detection Rate              0.3333            0.3200           0.3267
## Detection Prevalence        0.3333            0.3267           0.3400
## Balanced Accuracy           1.0000            0.9750           0.9800
```

El modelo obtenido ha ido considerando diferentes valores de k (número de vecinos más cercanos) y se observa que la precisión (*Accuracy*) aumenta a medida que k aumenta, con valores de 0.9733 para k = 5, 0.98 para k = 7 y 0.9733 para k = 9. La elección final del modelo se basa en el valor más alto de precisión, seleccionando k = 9. La matriz de confusión muestra cómo el modelo clasifica las muestras en cada clase. En la diagonal principal, se encuentran los casos correctamente clasificados. Por ejemplo, para la clase Setosa, se clasificaron correctamente los 50 casos. Algunos errores de clasificación se observan en la matriz fuera de la diagonal principal. La precisión global del modelo es del 97 %, lo que indica un alto grado de aciertos en la clasificación. El valor kappa, que tiene en cuenta la posibilidad de aciertos al azar, es alto (0.96), lo que indica un buen rendimiento del modelo más allá de lo que se esperaría.

Finalmente, los resultados sugieren que el modelo k-NN con k = 9 tiene un rendimiento sólido en la clasificación de las especies de iris en este conjunto de datos.

6.2. Paquete Hmisc

El paquete Hmisc en **R** proporciona diversas herramientas para el análisis y la visualización de datos estadísticos. Aunque no está específicamente diseñado para el análisis de datos biomédicos o de salud, se utiliza comúnmente en ese contexto debido a su versatilidad.

Ejemplo 9. Aplicación del paquete Hmisc

```
#Instalamos y cargamos el paquete Hmisc
if (!requireNamespace("Hmisc", quietly = TRUE))
    install.packages("Hmisc")
library(Hmisc)
## Warning: package 'Hmisc' was built under R version 4.3.2
##
## Attaching package: 'Hmisc'
## The following object is masked from 'package:AnnotationDbi':
##
##     contents
## The following object is masked from 'package:Biobase':
##
##     contents
## The following objects are masked from 'package:base':
##
##     format.pval, units
# Creamos un conjunto de datos de ejemplo
data <- data.frame(
  Age = c(25, 30, 35, 40, 45),
  Sex = c("Male", "Female", "Male", "Female", "Male"),
  Height = c(170, 160, 180, 155, 175),
  Weight = c(70, 55, 85, 50, 80)
)

# Realizamos el resumen descriptivo
summary(data)
##       Age         Sex                Height        Weight
```

```
##  Min.   :25   Length:5           Min.   :155   Min.   :50
##  1st Qu.:30   Class :character   1st Qu.:160   1st Qu.:55
##  Median :35   Mode  :character   Median :170   Median :70
##  Mean   :35                      Mean   :168   Mean   :68
##  3rd Qu.:40                      3rd Qu.:175   3rd Qu.:80
##  Max.   :45                      Max.   :180   Max.   :85
# Creamos una tabla de resumen para variables cuantitativas
quant_summary <- describe(data$Age)
print(quant_summary)
## data$Age
##        n  missing distinct     Info     Mean      Gmd
##        5        0        5        1       35       10
##
## Value       25  30  35  40  45
## Frequency    1   1   1   1   1
## Proportion 0.2 0.2 0.2 0.2 0.2
##
## For the frequency table, variable is rounded to the nearest 0
# Creamos una tabla de resumen para variables categóricas
cat_summary <- describe(data$Sex)
print(cat_summary)
## data$Sex
##        n  missing distinct
##        5        0        2
##
## Value      Female   Male
## Frequency       2      3
## Proportion    0.4    0.6
```

La función *describe* de Hmisc se utiliza para obtener resúmenes tanto para variables cuantitativas (*Age*) como para variables categóricas (*Sex*).

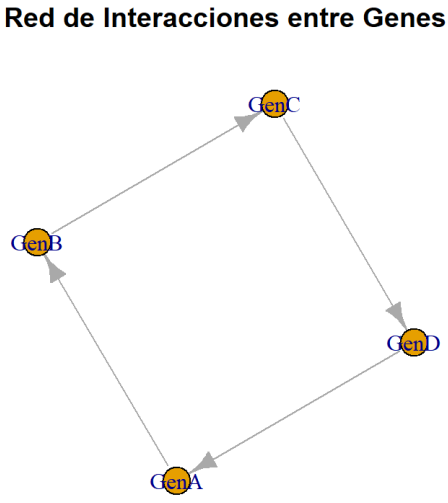
6.3. Paquete igraph

El paquete *igraph* se utiliza para el análisis y la visualización de redes y grafos de amplia aplicación en diferentes ámbitos.

Ejemplo 10. Aplicación del paquete igraph

En este ejemplo sencillo, podemos ver cómo definir una serie de genes y representar una conexión entre ellos.

```
# Instalamos y cargamos el paquete igraph

if (!requireNamespace("igraph", quietly = TRUE))

    install.packages("igraph")

library(igraph)

# Crear un grafo de interacciones entre genes relacionados con una enfermedad

grafo_genes <- graph(edges=c("GenA", "GenB", "GenB", "GenC", "GenC", "GenD",
"GenD", "GenA"),directed=TRUE)


# Visualizar el grafo

plot(grafo_genes, main="Red de Interacciones entre Genes",
vertex.label=c("GenA", "GenB", "GenC", "GenD"))
```



Los paquetes trabajados en este apartado son solo algunos de los que podemos utilizar en **R**. Desde **RStudio**, podemos acceder a la pestaña *Packages* para visualizar los paquetes que

tenemos instalados. Por otra parte, ejecutando las siguientes instrucciones, podemos conocer todos los paquetes que **R** tiene disponible.

```
#package_list <- available.packages()

#head(rownames(package_list))
```

Es necesario tener en cuenta que algunos de estos paquetes pueden instalarse dependiendo las versiones de **R**.

7. Ejercicios y casos prácticos con R

Ejercicio 1. Aplicación Shiny a partir de un conjunto de datos

A partir del conjunto de datos *heart* del paquete *survival*, realizad una aplicación Shiny que a partir de la selección de dos variables del conjunto de datos represente un gráfico de dispersión y uno de líneas, además de un histograma solo para la variable horizontal seleccionada. Para hacer este ejercicio podéis basaros en el *Ejemplo 2* realizado en el apartado *1.1. Ecosistema de R: Entornos reproducibles y visualización de datos*. Se aconseja ampliar conocimientos en el uso de diferentes parámetros en las funciones definidas, así como explorar otro tipo de funcionalidades para profundizar en el estudio del conjunto de datos, a criterio del estudiante.

Ejercicio 2. Explorador de datos

Mostrad un conjunto de datos a partir de un archivo .csv a vuestra elección que muestre también el resumen estadístico del conjunto de datos elegido. Por otra parte, añadid, la selección de una variable del conjunto de datos a partir de la cual se represente un gráfico del tipo histograma.

Ejercicio 3. Aplicación Shiny para conjunto de datos Iris

A partir del conjunto de datos Iris, realizad una aplicación Shiny para mostrar, a partir de una de las variables para seleccionar del conjunto de datos, los gráficos *boxplot* y *barplot*.

Ejercicio 4. Bioconductor con el paquete GenomicFeatures

Utilizando el paquete *GenomicFeatures*, definid un vector con la información de cada gen, la posición inicial y final, la dirección de la hebra (*strand*), el identificador del gen (id) y el tipo de gen (*type*).

Una orientación para definir algunos de estos datos los podéis encontrar en el *Ejemplo 4. Bioconductor utilizando el paquete GenomicFeatures*, y podemos definir la dirección de la hebra y el tipo de gen de la siguiente manera:

```
strand = c("+", "-", "+"), gene_type = c("protein_coding", "non_coding",
"protein_coding")
```

Mostrad los detalles utilizando un objeto de tipo GRanges.

Ejercicio 5. Bioconductor con el paquete limma

El paquete limma es un paquete de **R** que se utiliza para analizar datos de expresión génica y realizar análisis estadísticos en experimentos de *microarrays* y secuenciación de ARN (ARN-Seq). La sigla limma significa *linear models for microarray data*. A partir de los siguientes datos:

```
Gene Sample1 Sample2 Sample3 Sample4 Sample5 Sample6
Gene1   5              6          7        8         9          10
Gene2   4              5          6        7         8           9
Gene3   3              4          5        6         7           8
```

Realizad un análisis de expresión génica diferencial con el paquete limma.

Ejercicio 6. Bioconductor con ComplexHeatMap

El paquete ComplexHeatMap es útil para visualizar datos de matriz en forma de mapas de calor complejos con anotaciones adicionales. A partir de una matriz ficticia, cread un objeto HeatMap, mostradlo e interpretad los resultados.

Ejercicio 7. Bioconductor con Biostrings

El paquete Biostrings de Bioconductor en **R** está diseñado para la manipulación y el análisis de secuencias biológicas. Supongamos que definimos las siguientes secuencias:

```
secuencia1: "ATCGAATCGA" secuencia2: "TAGCTAGCTA"
```

Combinad las secuencias y obtened el complemento inverso de cada secuencia.

Ejercicio 8. Aplicaciones del paquete caret

A partir del paquete caret y del conjunto de datos Iris, cread un particionamiento del conjunto de datos sobre la variable *species* y cread y entrenad un modelo de k-NN para realizar predicciones. Finalmente, mostrad la matriz de confusión y comentad el resultado.

Ejercicio 9. Aplicaciones del paquete caret. Modelo de clasificación *random forest*

Supongamos que tenemos los siguientes datos de 5 pacientes a los que se les han practicado algunas pruebas médicas y donde, entre otros datos, se indica si poseen o no la enfermedad estudiada:

```
Edad = 30, 45, 50, 22, 65 Sexo = "M", "F", "M", "F", "M" ResultadoPrueba1 =
0.1, 0.5, 0.8, 1.2, 0.9 ResultadoPrueba2 = 12, 9, 15, 8, 10 Enfermedad =
"No", "No", "Sí", "Sí", "Sí"
```

Se pide entrenar un modelo de clasificación supervisada, por ejemplo, el *random forest*.

Ejercicio 10. Aplicación del paquete igraph al ámbito de las proteínas

Supongamos que tenemos la siguiente matriz de adyacencia que representa las interacciones entre proteínas:

```
matrix_adyacencia ( 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0 )
```

Se pide representar gráficamente la red de interacciones entre dichas proteínas.

Caso práctico 1

A partir de Bioconductor, y utilizando el paquete Biostrings, se pide realizar una aplicación sencilla con Shiny que sea un contador de ocurrencias de patrones de ADN.

Se requiere que la interfaz que se obtenga sea similar a la siguiente:

Para realizar pruebas, se pueden usar los siguientes valores: secuencia:

ATCGATCGATCGATCGATCGATCGATCGATCG patrón: ATCG.

Solución a los ejercicios propuestos y casos prácticos con R

Solución del ejercicio 1:

Una posible solución sería la siguiente:

```
# Cargamos los paquetes necesarios

library(shiny)

library(survival)

##

## Attaching package: 'survival'

## The following object is masked from 'package:caret':

##

##     cluster

library(ggplot2)


# Cargamos el conjunto de datos "heart"

data("heart")


# Definimos la interfaz de la aplicación Shiny

ui <- fluidPage(

  titlePanel("Conjunto de datos Heart"),

  sidebarLayout(

    sidebarPanel(

      # Definimos los selectores de variables x e y

      selectInput("x", "Selecciona la variable X:", choices =
colnames(heart)),

      selectInput("y", "Selecciona la variable Y:", choices =
colnames(heart)),

      selectInput("plot_type", "Selecciona el tipo de gráfico:",

                  choices = c("Gráfico de dispersión" = "scatter", "Líneas" =
"lines", "Histograma" = "histogram")),
```

```
      hr()
    ),
    mainPanel(
      # Dibujamos el gráfico seleccionado
      plotOutput("plot")
    )
  )
)

# Definimos la lógica de la aplicación Shiny
server <- function(input, output) {
  # Crear el gráfico
  output$plot <- renderPlot({
    x <- input$x
    y <- input$y
    plot_type <- input$plot_type

    if (plot_type == "scatter") {
      ggplot(heart, aes_string(x, y)) +
        geom_point() +
        xlab(x) +
        ylab(y) +
        ggtitle("Gráfico de Dispersión")
    } else if (plot_type == "lines") {
      ggplot(heart, aes_string(x, y, group = 1)) +
        geom_line() +
        xlab(x) +
        ylab(y) +
```

```
        ggtitle("Gráfico de Líneas")
    } else if (plot_type == "histogram") {
      ggplot(heart, aes_string(x)) +
        geom_histogram(binwidth = 10, fill = "blue", color = "black") +
        xlab(x) +
        ylab("Frecuencia") +
        ggtitle("Histograma")
    }
  })

}

# Ejecutar la aplicación Shiny
shinyApp(ui = ui, server = server)
```

Solución del ejercicio 2:

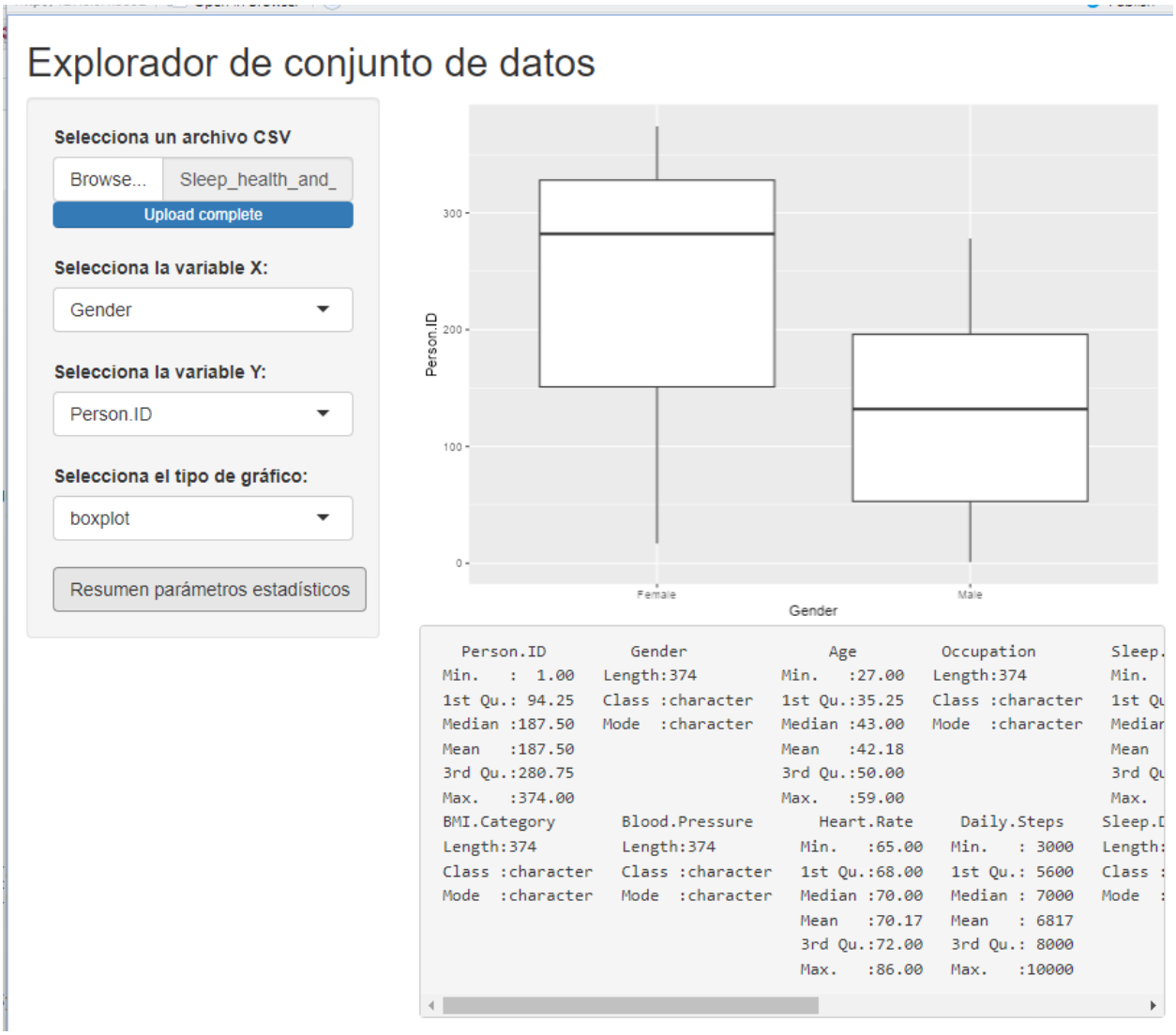
Una posible solución sería:

```
library(shiny)
library(ggplot2)
library(dplyr)
# Define la interfaz de la aplicación shiny
ui <- fluidPage(
  titlePanel("Explorador de conjunto de datos"),
  sidebarLayout(
    sidebarPanel(
      fileInput("file", "Selecciona un archivo CSV"),
      selectInput("x", "Selecciona la variable X:", ""),
      selectInput("y", "Selecciona la variable Y:", ""),
```

```
      selectInput("plot_type", "Selecciona el tipo de gráfico:",
c("scatterplot", "barplot", "boxplot", "histogram")),
      actionButton("show_summary", "Resumen parámetros estadísticos")
    ),
    mainPanel(
      plotOutput("plot"),
      verbatimTextOutput("summary_text")
    )
  )
)

# Definimos la lógica de la aplicación shiny
server <- function(input, output, session) {
  data <- reactive({
    req(input$file)
    if (!endsWith(input$file$name, ".csv")) {
      return(NULL)
    }
    read.csv(input$file$datapath)
  })

  observe({
    req(data())
    updateSelectInput(session, "x", choices = colnames(data()))
    updateSelectInput(session, "y", choices = colnames(data()))
  })

  output$plot <- renderPlot({
```

```
    req(data())
    if (input$plot_type == "scatterplot") {
      ggplot(data(), aes_string(x = input$x, y = input$y)) + geom_point()
    } else if (input$plot_type == "barplot") {
      ggplot(data(), aes_string(x = input$x)) + geom_bar()
    } else if (input$plot_type == "boxplot") {
      ggplot(data(), aes_string(x = input$x, y = input$y)) + geom_boxplot()
    } else if (input$plot_type == "histogram") {
      ggplot(data(), aes_string(x = input$x)) + geom_histogram()
    }
  })

  output$summary_text <- renderPrint({
    req(data())
    if (input$show_summary > 0) {
      summary(data())
    }
  })
}

# Ejecutamos la aplicación shiny
shinyApp(ui = ui, server = server)
```

Tras seleccionar uno de los archivos utilizado a lo largo de este documento, Sleep_health_and_lifestyle_dataset.csv, el resultado es:



Solución del ejercicio 3:

```
# Cargar las bibliotecas necesarias
library(shiny)
library(ggplot2)

# Cargar un conjunto de datos de ejemplo (en este caso, Iris)
data("iris")

# Definir la interfaz de la aplicación Shiny
```

```
ui <- fluidPage(
  titlePanel("Explorador de Datos con Gráficos"),
  sidebarLayout(
    sidebarPanel(
      # Selector de variables x
      selectInput("x", "Seleccione la variable X:", choices =
colnames(iris)),
      # Selector de tipo de gráfico
      selectInput("plot_type", "Seleccione el tipo de gráfico:",
                  choices = c("Bar Plot", "Box Plot"))
    ),
    mainPanel(
      # Gráfico resultante
      plotOutput("output_plot")
    )
  )
)

# Definir la lógica de la aplicación Shiny
server <- function(input, output) {
  # Crear el gráfico seleccionado
  output$output_plot <- renderPlot({
    x <- input$x
    data <- iris

    if (input$plot_type == "Bar Plot") {
      ggplot(data, aes_string(x = x)) +
        geom_bar() +
```

```
        xlab(x) +
        ylab("Frequency")
    } else if (input$plot_type == "Box Plot") {
      ggplot(data, aes_string(x = x, y = "Sepal.Length")) +
        geom_boxplot() +
        xlab(x) +
        ylab("Sepal Length")
    }
  })
}

# Ejecutar la aplicación Shiny
shinyApp(ui = ui, server = server)
```

Solución del ejercicio 4:

```
Mostrad los detalles utilizando un objeto de tipo GRanges.
# Cargar el paquete GenomicFeatures
library(GenomicFeatures)

# Crear un conjunto de genes más detallado
genes_detalles <- data.frame(
  seqnames = c("chr1", "chr2", "chr1"),
  start = c(100, 200, 300),
  end = c(200, 300, 400),
  gene_id = c("Gene1", "Gene2", "Gene3"),
  strand = c("+", "-", "+"),
  gene_type = c("protein_coding", "non_coding", "protein_coding"),
  stringsAsFactors = FALSE
)
```

```
# Crear un objeto `GRanges` que representa la información de los genes
gr_detalles <- GRanges(
  seqnames = genes_detalles$seqnames,
  ranges = IRanges(start = genes_detalles$start, end = genes_detalles$end),
  strand = genes_detalles$strand,
  id = genes_detalles$gene_id,
  type = genes_detalles$gene_type
)

# Mostrar la información detallada de los genes
print(gr_detalles)
## GRanges object with 3 ranges and 2 metadata columns:
##       seqnames    ranges strand |          id           type
##          <Rle> <IRanges>  <Rle> | <character>    <character>
##   [1]     chr1   100-200      + |       Gene1 protein_coding
##   [2]     chr2   200-300      - |       Gene2     non_coding
##   [3]     chr1   300-400      + |       Gene3 protein_coding
##   -------
##   seqinfo: 2 sequences from an unspecified genome; no seqlengths
```

Este resultado representa un objeto *GRanges* que contiene información genómica detallada sobre los genes, su posición en el cromosoma, la dirección de la hebra y metadatos adicionales.

Solución del ejercicio 5:

```
# Instalar y cargar el paquete necesario
if (!requireNamespace("limma", quietly = TRUE)) {
  if (!requireNamespace("BiocManager", quietly = TRUE))
    install.packages("BiocManager")
```

```
  BiocManager::install("limma")
}
library(limma)

# Cargamos datos de muestra
data <- read.table(header = TRUE, text = "
Gene Sample1 Sample2 Sample3 Sample4 Sample5 Sample6
Gene1 5 6 7 8 9 10
Gene2 4 5 6 7 8 9
Gene3 3 4 5 6 7 8
")

# Definimos grupos
group <- factor(c("Group1", "Group1", "Group1", "Group2", "Group2", "Group2"))

# Creamos un objeto DGEList
exprs_data <- data[, -1]
rownames(exprs_data) <- data$Gene
group <- factor(group)
design <- model.matrix(~0 + group)
colnames(design) <- levels(group)
fit <- lmFit(exprs_data, design)

# Realizamos análisis de expresión génica diferencial con limma
contrast <- makeContrasts(Group2 - Group1, levels = design)
fit2 <- contrasts.fit(fit, contrast)
fit2 <- eBayes(fit2)
results <- topTable(fit2, coef = 1)
```

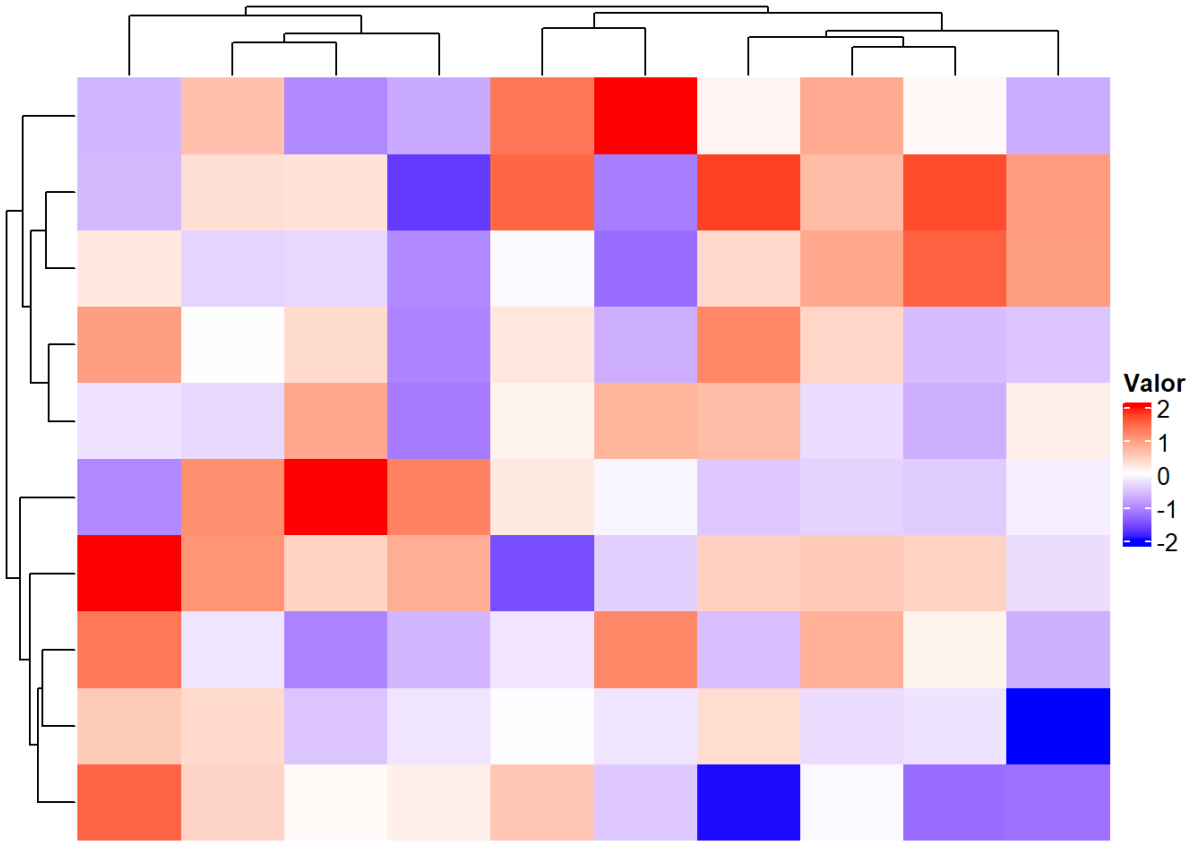
```
# Mostramos los resultados
print(results)
##       logFC AveExpr        t     P.Value   adj.P.Val         B
## Gene1     3     7.5 3.674235 0.003182214 0.003182214 0.2754422
## Gene3     3     5.5 3.674235 0.003182214 0.003182214 0.2754422
## Gene2      3     6.5 3.674235 0.003182214 0.003182214 0.2754422
```

Para entender un poco los resultados, es necesario conocer que *logFC* (*Logarithm of the Fold Change*) indica la magnitud del cambio en la expresión génica. Si *logFC* es positivo, significa que la expresión génica es más alta en el Grupo 2, en comparación con el Grupo 1, y si es negativo, significa que es más baja. Un p-value bajo sugiere que el cambio es estadísticamente significativo. FDR (*False Discovery Rate*) es una corrección del valor p; así, valores bajos de FDR (generalmente < 0.05) indican que los resultados son más confiables. Por tanto, genes con un *logFC* grande y un FDR bajo serían los más propensos a ser relevantes en un determinado estudio.

Solución del ejercicio 6:

```
# Instalamos y cargamos los paquetes ComplexHeatMap y circlize
if (!requireNamespace("ComplexHeatmap", quietly = TRUE)) {
  if (!requireNamespace("BiocManager", quietly = TRUE))
    install.packages("BiocManager")
  BiocManager::install(c("ComplexHeatmap", "circlize"))
}
library(ComplexHeatmap)
library(circlize)
# Simulamos los datos de una matriz a partir de una normal
set.seed(123)
matriz_ejemplo <- matrix(rnorm(100), ncol = 10)

# Creamos un objeto Heatmap
heatmap_objeto <- Heatmap(matriz_ejemplo,
```

```
                            name = "Valor",
                            show_row_names = FALSE,
                            show_column_names = FALSE,
                            col = colorRamp2(c(-2, 0, 2), c("blue", "white",
"red")))

# Dibujamos el mapa de calor
draw(heatmap_objeto, heatmap_legend_side = "right")
```



El mapa de calor tiene el objetivo de visualizar patrones y tendencias en los datos.

En este mapa de calor, los colores tienen la siguiente interpretación:

Azul: Representa valores bajos.

Blanco: Representa valores intermedios.

Rojo: Representa valores altos.

La intensidad del color indica el nivel de expresión o la magnitud del valor asociado. Colores más intensos representan valores más altos o más bajos, según la paleta de colores utilizada. Las columnas representan diferentes muestras o condiciones y las filas representan diferentes elementos o genes. Los patrones horizontales (en filas) pueden indicar similitudes o diferencias en la expresión de genes entre diferentes elementos. Los patrones verticales (en columnas) pueden indicar similitudes o diferencias en la expresión de un gen a lo largo de diferentes condiciones. La agrupación (clusterización) de filas o columnas puede sugerir similitudes en los perfiles de expresión génica.

En este caso, los datos son simulados; no obstante, en el contexto de un estudio de genes, un cambio de color podría significar una respuesta a determinadas condiciones.

Solución del ejercicio 7:

```
# Instalamos y cargamos el paquete Biostrings
if (!requireNamespace("Biostrings", quietly = TRUE)) {
  if (!requireNamespace("BiocManager", quietly = TRUE))
    install.packages("BiocManager")
  BiocManager::install("Biostrings")
}
library(Biostrings)
## Loading required package: XVector
##
## Attaching package: 'Biostrings'
## The following object is masked from 'package:grid':
##
##     pattern
## The following objects are masked from 'package:Hmisc':
##
##     mask, translate
## The following object is masked from 'package:base':
##
```

```
##     strsplit
# Crear secuencias de ADN
secuencia1 <- DNAString("ATCGAATCGA")
secuencia2 <- DNAString("TAGCTAGCTA")

# Combinar las secuencias en un DNAStringSet
secuencias <- DNAStringSet(c(secuencia1, secuencia2))

# Obtener el complemento inverso de cada secuencia
comp_inverso <- reverseComplement(secuencias)

# Imprimir las secuencias originales y sus complementos inversos
cat("Secuencias Originales:\n")
## Secuencias Originales:
print(secuencias)
## DNAStringSet object of length 1:
##     width seq
## [1]    20 ATCGAATCGATAGCTAGCTA
cat("\nComplementos Inversos de las Secuencias:\n")
##
## Complementos Inversos de las Secuencias:
print(comp_inverso)
## DNAStringSet object of length 1:
##     width seq
## [1]    20 TAGCTAGCTATCGATTCGAT
```

Solución del ejercicio 8:

```
# Instalamos y cargamos el paquete caret
if (!requireNamespace("caret", quietly = TRUE))
```

```
    install.packages("caret")
library(caret)


# Cargamos el conjunto de datos iris
data(iris)


# Creamos un particionamiento del conjunto de datos
set.seed(123)
training_indices <- createDataPartition(iris$Species, p = 0.7, list = FALSE)
training_data <- iris[training_indices, ]
testing_data <- iris[-training_indices, ]


# Creamos y entrenamos un modelo de k-NN con caret
model <- train(Species ~ ., data = training_data, method = "knn", trControl =
trainControl(method = "cv"))


# Realizamos predicciones en el conjunto de prueba
predictions <- predict(model, testing_data)


# Mostramos la matriz de confusión y otras métricas
conf_matrix <- confusionMatrix(predictions, testing_data$Species)
print(conf_matrix)
## Confusion Matrix and Statistics
##
##             Reference
## Prediction   setosa versicolor virginica
##   setosa         15          0         0
##   versicolor      0         15         1
```

```
##   virginica       0          0        14
##
## Overall Statistics
##
##                Accuracy : 0.9778
##                  95% CI : (0.8823, 0.9994)
##     No Information Rate : 0.3333
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.9667
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: setosa Class: versicolor Class: virginica
## Sensitivity                 1.0000            1.0000           0.9333
## Specificity                 1.0000            0.9667           1.0000
## Pos Pred Value              1.0000            0.9375           1.0000
## Neg Pred Value              1.0000            1.0000           0.9677
## Prevalence                  0.3333            0.3333           0.3333
## Detection Rate              0.3333            0.3333           0.3111
## Detection Prevalence        0.3333            0.3556           0.3111
## Balanced Accuracy           1.0000            0.9833           0.9667
```

Según el valor de *Accuracy*, la precisión del modelo es del 97.78 %, lo que indica que el 97.78 % de las predicciones fueron correctas en el conjunto de prueba. Como el valor de kappa es de 0.9667, es posible que exista concordancia significativa entre las predicciones del modelo y las clases de especies reales.

Setosa: El modelo tiene un rendimiento perfecto para la clase Setosa, con sensibilidad, especificidad y valor predictivo positivo del 100 %.

Versicolor: También tiene un rendimiento sólido para Versicolor, con sensibilidad y valor predictivo positivo del 100 %, aunque la especificidad es más baja.

Virginica: El modelo tiene una sensibilidad del 93.33 % para la clase Virginica.

El *Balanced Accuracy* indica que el modelo muestra un buen equilibrio en la precisión de las tres clases, con una precisión equilibrada del 100 % para Setosa, 98.33 % para Versicolor y 96.67 % para Virginica.

Así pues, el modelo de k-NN parece ser bastante preciso en la clasificación de las flores en el conjunto de datos Iris, con un rendimiento más fuerte para las clases Setosa y Versicolor.

Solución del ejercicio 9:

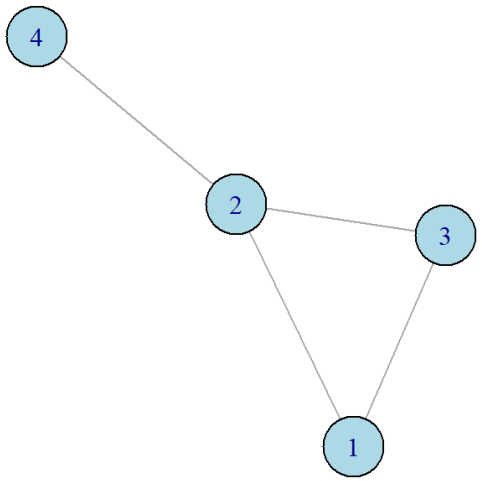
```
# Cargamos el paquete caret
library(caret)


# Crear un dataframe de ejemplo
datos <- data.frame(
  Edad = c(30, 45, 50, 22, 65),
  Sexo = c("M", "F", "M", "F", "M"),
  ResultadoPrueba1 = c(0.1, 0.5, 0.8, 1.2, 0.9),
  ResultadoPrueba2 = c(12, 9, 15, 8, 10),
  Enfermedad = c("No", "No", "Sí", "Sí", "Sí")
)


# Convertimos la variable Sexo en factor
datos$Sexo <- as.factor(datos$Sexo)


# Definir el control del entrenamiento
control <- trainControl(method = "cv", number = 5)  # Validación cruzada con
5 folds
```

```
# Entrenar un modelo de clasificación (por ejemplo, un modelo de bosque
aleatorio)

modelo <- train(Enfermedad ~ ., data = datos, method = "rf", trControl =
control)

## Warning in nominalTrainWorkflow(x = x, y = y, wts = weights, info =
trainInfo,

## : There were missing values in resampled performance measures.

# Ver los resultados del modelo

print(modelo)

## Random Forest

##

## 5 samples

## 4 predictors

## 2 classes: 'No', 'Sí'

##

## No pre-processing

## Resampling: Cross-Validated (5 fold)

## Summary of sample sizes: 4, 4, 4, 4, 4

## Resampling results across tuning parameters:

##

##   mtry  Accuracy  Kappa

##   2     0.4       0

##   3     0.6       0

##   4     0.6       0

##

## Accuracy was used to select the optimal model using the largest value.

## The final value used for the model was mtry = 3.
```

La muestra es demasiado pequeña para extraer conclusiones, no obstante, de los datos obtenidos se puede sugerir que el modelo alcanza una precisión máxima del 60 %.

Solución del ejercicio 10:

```
# Creamos la matriz de adyacencia que representa interacciones entre
proteínas

matriz_adyacencia <- matrix(c(

  0, 1, 1, 0,

  1, 0, 1, 1,

  1, 1, 0, 0,

  0, 1, 0, 0

), nrow = 4, byrow = TRUE)


# Convertimos la matriz en un objeto graph de igraph

red_proteinas <- graph.adjacency(matriz_adyacencia, mode = "undirected")


# Visualizamos la red de interacciones entre proteínas

plot(red_proteinas,

     layout = layout.fruchterman.reingold,  # Algoritmo de disposición para
organizar los nodos

     vertex.label = 1:4,                    # Etiquetas de los nodos

     vertex.color = "lightblue",            # Color de los nodos

     vertex.size = 30)                      # Tamaño de los nodos
```



Solución caso práctico:

```
library(shiny)
library(Biostrings)

# Define la interfaz de la aplicación shiny
ui <- fluidPage(
  titlePanel("Contador de Ocurrencias de Patrones de ADN"),
  sidebarLayout(
    sidebarPanel(
      textInput("secuencia", "Introduce una secuencia de ADN:"),
      textInput("patron", "Introduce el patrón a buscar:"),
      actionButton("contador", "Contar ocurrencias"),
      br(),
      verbatimTextOutput("resultado")
    ),
    mainPanel()
  )
)

# Definimos la lógica de la aplicación Shiny
server <- function(input, output) {
  output$resultado <- renderPrint({
    req(input$contador)
    isolate({
      # Obtener la secuencia de ADN introducida
      adn_secuencia <- DNAString(input$secuencia)

      # Obtener el patrón introducido
```

```
      patron <- DNAString(input$patron)

      # Contar las ocurrencias del patrón en la secuencia
      contador <- countPattern(patron, adn_secuencia)

      # Devolvemos el resultado
      paste("Número de ocurrencias:", contador)
    })
  })
}

# Ejecutar la aplicación Shiny
shinyApp(ui = ui, server = server)
```

Información adicional

Algunos de los recursos materiales utilizados y recomendados para trabajar este **LAB6** son los siguientes:

Shiny

Bioconductor