Laboratorio 6

Factores, data frames y listas

Factores

Los factores son esencialmente solo vectores, pero de valores **categóricos**. Eso significa que los elementos de un factor deben considerarse categorías o clases y no como números. Por ejemplo, las categorías como chico, medio y grande podrían codificarse como números, pero en realidad no hay números naturales que asignarles. Podríamos codificar tamaños de bebidas sin alcohol como 1, 2 y 3 para pequeño, mediano y grande. Al hacer esto, estamos diciendo implícitamente que la diferencia entre chico y medio es la mitad de la diferencia entre chico y grande, que puede no ser el caso. Los datos con los tamaños *chico, medio y grande deben codificarse como datos categóricos, no como números y en R eso significa codificarlos como factores.

Un factor generalmente se construye dando una lista de cadenas. Estas se traducen en diferentes categorías y el factor se convierte en un vector de estas categorías.

```
f <- factor(c("chico", "chico", "medio","grande", "chico", "grande"))
f
```

```
[1] chico  chico  medio  grande chico  grande
Levels: chico grande medio
```

Las categorías son llamados *niveles*

```
levels(f)
```

```
[1] "chico"  "grande" "medio"
```

Por defecto, estos están ordenados alfabéticamente, lo que en este ejemplo nos da el orden *chico*, *grande*, *medio*. Puede cambiar este orden especificando los niveles cuando se crea el factor.

```
ff <- factor(c("chico", "chico", "medio", "grande", "chico", "grande"),
             levels = c( "chico", "medio", "grande"))
ff
```

```
[1] chico  chico  medio  grande chico  grande
Levels: chico medio grande
```

Cambiar el orden de los niveles de esta manera cambia la cantidad de funciones que manejan el factor. El orden de los niveles de los factores afecta principalmente cómo se imprime la información resumida y cómo se muestran los factores.

```
summary(f)
```

```
 chico grande  medio
     3      2      1
```

```
summary(ff)
```

```
 chico  medio grande
     3      1      2
```

Sin embargo, el orden en que se asignan los niveles no debe ser un orden de las categorías. Solo se usa para mostrar resultados; no hay una semántica de orden dada a los niveles a menos que especifiques uno explícitamente.

Algunos datos categóricos tienen un orden natural. Como *chico*, *grande*, *mediano*. Otras categorías no están ordenadas de forma natural. No hay una forma natural de ordenar *rojo*, *verde* y *azul*. Cuando imprimimos datos, siempre salen ordenados ya que el texto siempre sale ordenado. Cuando graficamos datos, generalmente también se ordena.

Pero en muchos modelos matemáticos, tratamos los datos categóricos ordenados diferentes de los datos caóticos no ordenados, por lo que la distinción a veces es importante. Por defecto, los factores no tratan los niveles como ordenados, por lo que suponen que los datos categóricos son como rojo, azul, verde en lugar de los ordenados como chico, grande, medio.

Si desea especificar que los niveles están ordenados, puede hacerse utilizando el argumento `ordered` con la función `factor ()`.

```
of <- factor(c("chico", "chico", "medio",
               "grande", "chico", "grande"),
             levels = c( "chico", "medio", "grande"),
             ordered = TRUE)
of
```

```
[1] chico  chico  medio  grande chico  grande
Levels: chico < medio < grande
```

En realidad, un factor no se almacena como cadenas, aunque lo creamos a partir de un vector de cadenas. Se almacena como un vector de enteros donde los enteros son índices en los niveles. Esto puede incomodar si tratas de usar un factor para indexar.

Lee el siguiente código cuidadosamente. Tenemos el vector `v` que puede indexarse con las letras A, B, C y D.

Creamos un factor `ff`, que consiste en estas cuatro letras en ese orden. Cuando indexamos con él, obtenemos lo que esperaríamos. Como `ff` son las letras A a D, seleccionamos los valores de `v` con esas etiquetas y en ese orden.

```
v <- 1:4
names(v) <- LETTERS[1:4]
v
```

```
A B C D
1 2 3 4
```

```
(ff <- factor(LETTERS[1:4]))
```

```
[1] A B C D
Levels: A B C D
```

```
v[ff]
```

```
A B C D
1 2 3 4
```

Sin embargo, tenemos suerte de obtener el resultado esperado, por que esta expresión no se indexa con los nombres que podríamos esperar que use. Lee lo siguiente con más cuidado,

```
(ff <- factor(LETTERS[1:4], levels = rev(LETTERS[1:4])))
```

```
[1] A B C D
Levels: D C B A
```

```
v[ff]
```

```
D C B A
4 3 2 1
```

Esta vez `ff` sigue siendo un vector con las categorías A a D en ese orden, pero hemos especificado que los niveles son D, C, B y A, en ese orden. Asi los valores numéricos de las categorías se almacenan como son en realidad.

```
as.numeric(ff)
```

```
[1] 4 3 2 1
```

Lo que obtenemos cuando lo usamos para indexar en `v` son los índices numéricos, por lo que obtenemos los valores sacados de v en el orden inverso de lo que esperaríamos si no lo supiéramos (lo que ahora sabes).

La forma más fácil de tratar un factor como las actuales etiquetas es trasladarlo a un vector de cadenas. Puedes usar ese vector para indexar:

```
as.vector(ff)
```

```
[1] "A" "B" "C" "D"
```

```
v[as.vector(ff)]
```

```
A B C D
1 2 3 4
```

Si alguna vez te encuentras usando un factor para indexar algo, o de alguna otra manera tratas a un factor como si fuera un vector de cadenas, realmente deberías detenerte y asegurarte de convertirlo explícitamente en un vector de cadenas.

Tratar un factor como si fuera un vector de cadenas, cuando de hecho es un vector de números enteros, solo conduce a sufrimiento en el largo plazo.

Data frames

Los vectores que hemos visto, cualquiera que sea su tipo, son solo secuencias de datos. No hay estructura para ellos excepto el orden de secuencia, que puede o no ser relevante para la interpretación de los datos. No es así como se ven los datos que queremos analizar. Lo que generalmente tenemos es varias variables que están relacionadas como parte de las mismas observaciones.

Para cada punto de datos observado, tenemos un valor para cada una de estas variables (o indicaciones de datos faltantes si no se observaron algunas variables). Básicamente, lo que tienes es una tabla con una fila por observación y una columna por variable. El tipo de datos para tales tablas en R es el `data.frame`.

Un `data.frame` es una colección de vectores, donde todos deben ser de la misma longitud y se trata como una tabla bidimensional. Normalmente pensamos en los data frames como si cada fila correspondiera a alguna observación y cada columna a alguna propiedad de las observaciones. El tratamiento de los data frames de esa manera los hace extremadamente útiles para el modelado y el ajuste estadístico.

Puedes crear un `data frame` explícitamente utilizando la función `data.frame`, pero generalmente se leerá en el data frame de los archivos.

```
df <- data.frame(a = 1:4, b = letters[1:4])
df
```

```
  a b
1 1 a
2 2 b
3 3 c
4 4 d
```

Para llegar a los elementos individuales en un data frame, se debe indexarlos. Como se trata de una estructura de datos bidimensional, debes darle dos índices.

```
df[1,1]
```

```
[1] 1
```

Sin embargo, puede dejar uno vacío, en cuyo caso obtendrá una columna completa o una fila completa.

```
df[1,]
```

```
  a b
1 1 a
```

```
df[, 1]
```

```
[1] 1 2 3 4
```

Si las filas o columnas tienen un nombre, también se pueden usar esos nombres para indexar. Esto se utiliza principalmente para los nombres de las columna, ya que son las columnas las que corresponden a las variables observadas en un conjunto de datos. Hay dos formas de llegar a una columna, indexando explícitamente:

```
df[,"a"]
```

```
[1] 1 2 3 4
```

O usando la notación `$ column_name` que hace lo mismo pero te permite acceder a una columna sin tener que usar la operación `[]` y citar el nombre de una columna.

```
df$b
```

```
[1] a b c d
Levels: a b c d
```

De forma predeterminada, un data frame considerará un vector de caracteres como un factor, deberás indicar explícitamente si desea un vector de caracteres.

```
df <- data.frame(a = 1:4, b = letters[1:4], stringsAsFactors = FALSE)
```

Las funciones para leer datos de varios formatos de texto generalmente también convertirán vectores de cadena en factores, y se debe evitar esto explícitamente. El paquete readr es una notable excepción donde es predeterminado tratar vectores de caracteres como vectores de caracteres.

Podemos combinar dos data frames por filas o columnas mediante las funciones `rbind` y `cbind`:

```
df2 <- data.frame(a = 5:7, b = letters[5:7])
rbind(df, df2)
```

```
  a b
1 1 a
2 2 b
3 3 c
4 4 d
5 5 e
6 6 f
7 7 g
```

```
df3 <- data.frame(c = 5:8, d = letters[5:8])
cbind(df, df3)
```

```
  a b c d
1 1 a 5 e
2 2 b 6 f
3 3 c 7 g
4 4 d 8 h
```

Listas

Un objeto lista es un objeto R genérico que puede almacenar otros objetos de cualquier tipo. En un objeto de lista, podemos almacenar constantes únicas, vectores de valores numéricos, factores, data frames e incluso matrices.

```
var1 <- c(101,102,103,104,105)
var2 <- c(25,22,29,34,33)
var3 <- c("No-diabetico", "Diabetico", "No-diabetico", "No-diabetic", "Diabetico")
var4 <- factor(c("masculino","masculino","femenino","femenino","masculino"))
diab.dat <- data.frame(var1,var2,var3,var4)
mat.array<-array(dim=c(2,2,3))
set.seed(12345)
mat.array[,,1]<-rnorm(4)
mat.array[,,2]<-rnorm(4)
mat.array[,,3]<-rnorm(4)

# creando una lista
obj.lista <- list(elem1=var1,elem2=var2,elem3=var3,elem4=var4,elem5=diab.dat,elem6=mat.array)
obj.lista
```

```
$elem1
[1] 101 102 103 104 105

$elem2
[1] 25 22 29 34 33

$elem3
[1] "No-diabetico" "Diabetico"    "No-diabetico" "No-diabetic"
[5] "Diabetico"

$elem4
[1] masculino masculino femenino  femenino  masculino
Levels: femenino masculino

$elem5
  var1 var2         var3      var4
1  101   25 No-diabetico masculino
2  102   22    Diabetico masculino
3  103   29 No-diabetico  femenino
4  104   34  No-diabetic  femenino
5  105   33    Diabetico masculino

$elem6
, , 1

          [,1]       [,2]
[1,] 0.5855288 -0.1093033
[2,] 0.7094660 -0.4534972

, , 2

           [,1]       [,2]
[1,]  0.6058875  0.6300986
[2,] -1.8179560 -0.2761841
```

```
, , 3

           [,1]       [,2]
[1,] -0.2841597 -0.1162478
[2,] -0.9193220  1.8173120
```

Para acceder a elementos individuales desde un objeto de lista, podemos usar el nombre de ese elemento o usar corchetes dobles con el índice de esos elementos. Por ejemplo, `obj.lista[[1]]` dará el primer elemento del objeto de lista recién creado.

Trabajando con valores faltantes

La mayoría de los conjuntos de datos tienen valores faltantes: parámetros que no se observaron o que se registraron incorrectamente y tuvieron que ocultarse. La forma en que manejes los datos faltantes en un análisis depende de los datos y el análisis, pero debe abordarse, incluso si todo lo que se hace es eliminar todas las observaciones con datos faltantes.

Los datos faltantes se representan en R por el valor especial `NA` (no disponible). Los valores de cualquier tipo pueden se faltantes y representarse como `NA` y lo que es más importante, R sabe que `NA` significa valores faltantes y trata las NA en como lo que son en consecuencia. Siempre debes representar datos faltantes como `NA` en lugar de algún número especial (como `-1` o `999` o lo que sea). R sabe cómo trabajar con `NA`, pero no tiene forma de saber que `-1` significa algo además de menos uno.

Las operaciones que involucran a `NA` son en sí `NA`. No puedes operar con datos faltantes y obtener a cambio nada más que valores faltantes. Esto también significa que si comparas dos `NA`, obtienes `NA`. Como a NA le falta información, ni siquiera es igual a ella misma.

```
NA + 5
```

```
[1] NA
```

```
NA == NA
```

```
[1] NA
```

```
NA != NA
```

```
[1] NA
```

Si desea verificar si un valor es faltante , debes usar la función `is.na`:

```
is.na(NA)
```

```
[1] TRUE
```

```
is.na(4)
```

```
[1] FALSE
```

```
v <- c(1,NA,2)
sum(v)
```

```
[1] NA
```

Si solo quieres ignorar los valores de `NA`, a menudo hay un parámetro para especificar esto:

```
sum(v, na.rm = TRUE)
```

```
[1] 3
```

Ejercicios

1 .¿Qué ocurre cuando a un factor modificas sus niveles(levels)?

```
f1 <- factor(letters)
levels(f1) <- rev(levels(f1))
```

2 .¿Cómo deberias describir los tres objetos?.

```
x1 <- array(1:5, c(1, 1, 5))
x2 <- array(1:5, c(1, 5, 1))
x3 <- array(1:5, c(5, 1, 1))
```

3 . ¿Qué atributos posee un data frame?.

4 . ¿Qué hace `as.matrix()` cuando se aplica a un data frane con columnas de diferentes tipos?.

5 . ¿Puedes tener un data frame con 0 filas? ¿Qué hay de 0 columnas?.

6 . ¿Qué devuelve `dim()` cuando se aplica a un vector?

7 . Si `is.matrix(x)` es `TRUE`, ¿qué devolverá `is.array(x)`?.

8 . `Cu\'al es el funcionamiento de`is.vector()y is.numeric()`y explica que tiene de diferente con`is.list()y is.character()‘?.

9 . ¿ Por qué es `1 = "1"` verdad (TRUE)?. Por qué `-1 < FALSE` verdad?, ¿ Por qué `"one" < 2` es falso (FALSE)?.

10 . Predice la salida del siguiente c'odigo

```
c(1, FALSE)
c("a", 1)
c(list(1), "a")
c(TRUE, 1L)
```

Manipulación de datos usando plyr y dplyr

A menudo recolectamos datos a través de diferentes lugares y analizamos los datos como un todo, pero a veces es útil realizar algunas tareas por separado entre diferentes grupos. Como ejemplo, si recopilamos en detalles los ingresos de diferentes personas de seis regiones diferentes, entonces podríamos estar interesados en ver la distribución del ingreso entre diferentes profesiones (considerando cinco profesiones diferentes), en seis regiones.

Este ingreso puede variar dependiendo de si la persona es hombre o mujer. En esta situación, podemos conceptualizar este problema dividiendo el conjunto de datos según la profesión, el género y la región. Debe haber `5 x 6 x 2 = 60` grupos diferentes, y necesitamos calcular el ingreso promedio por separado para cada grupo. Finalmente, queremos combinar el resultado para ver toda la información una al lado de la otra. Esta operación grupal suele denominarse método de análisis de datos `dividir-aplicar-combinar`.

En este enfoque, primero dividimos el conjunto de datos en algunos grupos mutuamente excluyentes. Luego aplicamos una tarea en cada grupo y combinamos todos los resultados para obtener el resultado deseado.

Esta tarea grupal podría generar nuevas variables, resumir variables existentes o incluso realizar análisis de regresión en cada grupo. Finalmente, la combinación de enfoques nos ayuda a obtener un buen resultado para comparar los resultados de diferentes grupos.

Estrategia dividir-aplicar-combinar

Para fines de demostración, utilizaremos un conjunto de datos de flores `iris`, que está disponible en R. La flor del iris tiene tres especies diferentes: `iris setosa`, `iris virgen` e `iris versicolor`.

Se recogieron `50` muestras de cada especie y para cada muestra, se midieron cuatro variables: la longitud y el ancho de los sépalos y pétalos. El nombre de cada flor se almacena debajo de la columna de especies, y la longitud y el ancho del sépalo se almacena debajo de las columnas `Sepal.Length` y `Sepal.Width`, respectivamente.

Del mismo modo, la longitud y el ancho del pétalo se almacenan en las columnas `Petal.Length` y `Petal.Width` respectivamente.

El siguiente comando muestra las primeras filas del data frame de iris:

```
head(iris)
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

Ahora utilizaremos la estrategia dividir-aplicar-combinar para encontrar el ancho y la longitud promedio de sépalos y pétalos para tres especies diferentes de iris. La estrategia será la siguiente:

1 . Primero dividiremos el conjunto de datos en tres subconjuntos de acuerdo con las especies de la flor.

2 . Luego, para cada subconjunto, calcularemos el ancho y la longitud promedio de los sépalosy pétalos.

3 . Finalmente, combinaremos todos los resultados para compararlos entre sí.

```
# Paso 1: Dividimos el conjunto de datos
iris.setosa <- subset(iris,Species=="setosa",
                      select=c(Sepal.Length,Sepal.Width,Petal.Length,Petal.Width))

iris.versicolor <- subset(iris,Species=="versicolor",
                          select=c(Sepal.Length,Sepal.Width,Petal.Length,Petal.Width))
iris.virginica <- subset(iris,Species=="virginica",
                         select=c(Sepal.Length,Sepal.Width,Petal.Length,Petal.Width))

# Paso 2: Aplicando la funcion mean para  calcular la  media
setosa <- colMeans(iris.setosa)
versicolor <- colMeans(iris.versicolor)
virginica <-colMeans(iris.virginica)

# Paso 3: Combininando resultados
rbind(setosa=setosa,versicolor=versicolor,virginica=virginica)
```

```
           Sepal.Length Sepal.Width Petal.Length Petal.Width
setosa            5.006       3.428        1.462       0.246
versicolor        5.936       2.770        4.260       1.326
virginica         6.588       2.974        5.552       2.026
```

Este es el código detallado para implementar el enfoque `dividir-aplicar-combinar`. Podríamos implementar la estrategia con menos código, de la siguiente manera:

```r
# Paso 1: Dividimos el conjunto de datos
iris.split <- split(iris,as.factor(iris$Species))

# Paso 2: Aplicando la funcion mean para  calcular la  media
iris.apply <- lapply(iris.split,function(x)colMeans(x[-5]))

# Paso 3: Combinando resultados
iris.combinado <- do.call(rbind,iris.apply)
iris.combinado
```

```
           Sepal.Length Sepal.Width Petal.Length Petal.Width
setosa            5.006       3.428        1.462       0.246
versicolor        5.936       2.770        4.260       1.326
virginica         6.588       2.974        5.552       2.026
```

Las librerias plyr y dplyr

El paquete `plyr` nos ayuda a implementar el enfoque en una línea. Como R tiene múltiples estructuras de datos, necesitamos múltiples funciones para trabajar en diferentes estructuras de datos. R tiene tres estructuras de datos principales: lista, matriz y data frames . Entonces, podría haber tres tipos diferentes de entrada, y la salida podría producir tres tipos diferentes de estructuras de datos.

Podría haber `3 x 3 = 9` combinaciones posibles y por esta razón, `plyr` tiene `9` funciones para incorporar todas las combinaciones de entrada y salida. Además, tenemos tres funciones adicionales que toman seis tipos diferentes de entrada pero muestran solo un tipo de salida.

El paquete `plyr` funciona en todo tipo de estructura de datos, mientras que el paquete `dplyr` está diseñado para funcionar solo en data frames. El paquete `dplyr` ofrece un conjunto completo de funciones para realizar todo tipo de manipulación de datos que necesitamos en el proceso de análisis.

Estas funciones toman un data frame como entrada y también producen un data frame como salida; de ahí el nombre: `dplyr`. Hay dos tipos diferentes de funciones en el paquete dplyr: una `función de tabla única` y una `función de agregado`. La función de tabla única toma un data frame como entrada y toma una acción, como subdividir el data frame, generando nuevas columnas en el data frame o reorganizar el data frame.

La función de agregado toma una columna como entrada y produce un solo valor como salida, que se usa principalmente para resumir columnas. Estas funciones no nos permiten realizar ninguna operación en grupo, pero combinemos estas funciones con la función `group_by ()`. Esto nos permite implementar el enfoque `dividir-aplicar-combinar`.

Para realizar cualquier tipo de procesamiento de datos, necesitamos saber el tipo de entrada que debemos proporcionar y el formato de salida esperado. En la mayoría de las funciones de R, es difícil entender a partir de los nombres de funciones qué tipos de entrada aceptan y cuáles son los tipos de salida esperados. Los nombres de funciones en el paquete `plyr` son mucho más intuitivos e instructivos sobre sus tipos de entrada y salida, en comparación con cualquier otro paquete disponible.

Cada función se nombra de acuerdo con el tipo de entrada que toma y el tipo de salida que produce. La primera letra del nombre de la función especifica la entrada, y la segunda letra especifica el tipo de salida; `a` representa una matriz, `d` representa un data frame, `l` representa una lista, etc.

Por ejemplo, el nombre de función `adply ()` toma entrada como una matriz y produce salida como un data frame.

Hay algunos casos especiales que implican operar con matrices que corresponden a la función `mapply()` de R. En el paquete base R, `mapply()` puede tomar múltiples entradas como argumentos separados, mientras que `a* ply( )` toma solo un solo argumento de matriz. Sin embargo, el argumento separado en `mapply()` debe ser de la misma longitud.

Las funciones `mapply()` que son equivalentes en `plyr` son `maply()`, `mdply()`, `mlply()` y `m_ply()`. Ten en cuenta que, siempre que un nombre de función se escribe utilizando un símbolo de estrella, como `*ply ()`, indica que la entrada es una matriz. La salida puede estar en cualquier formato: matriz, data frame o lista. Opcionalmente, la salida puede descartarse.

Para explicar la naturaleza intuitiva de la entrada y la salida, proporcionaremos un ejemplo utilizando el conjunto de datos `iris`. Esta vez, usaremos el conjunto de datos `iris3`; este es el mismo conjunt de datos, pero almacenado en un formato de matriz tridimensional. Calcularemos la media de cada variable para cada especie, como se muestra en el siguiente código:

```
library(plyr)
# clase del conjunto de datos  iris3
class(iris3)
```

```
[1] "array"
```

```
# dimension del conjunto de datos iris3
dim(iris3)
```

```
[1] 50  4  3
```

```
# Calculamos la media para la columna para cada especie y
# la salida es un data frame
iris_media <- adply(iris3,3,colMeans)
class(iris_media)
```

```
[1] "data.frame"
```

```
iris_media
```

```
          X1 Sepal L. Sepal W. Petal L. Petal W.
1     Setosa    5.006    3.428    1.462    0.246
2 Versicolor    5.936    2.770    4.260    1.326
3  Virginica    6.588    2.974    5.552    2.026
```

Como `iris3` es una matriz, necesitamos especificar de acuerdo a qué dimensión dividiremos la matriz. Especificamos esto usando el parámetro `.margins`, en la función `adply`.

Ponemos `.margins = 3` en la función `adply` como: `adply (iris3, .margins = 3, colMeans)` para decirle a la función `adply` que queremos la división de acuerdo con la tercera dimensión de un objeto de matriz tridimensional.

Si quisiéramos dividir los datos según la fila o columna, pondríamos `1` o `2`, respectivamente. También es legítimo usar una combinación de dimensiones. En ese caso, `c(1,2)` podría ser una opción.

El siguiente fragmento de código calculamos la media de la columna para cada especie, con la entrada como una matriz y la salida como una matriz:

```
# Calculamos la media, pero la salida  es una matriz
iris_media<- aaply(iris3,3,colMeans)
class(iris_media)
```

```
[1] "matrix"
```

```
iris_media
```

```
X1           Sepal L. Sepal W. Petal L. Petal W.
  Setosa        5.006    3.428    1.462    0.246
  Versicolor    5.936    2.770    4.260    1.326
  Virginica     6.588    2.974    5.552    2.026
```

```r
# Calculamos la media, pero la salida  es una lista
iris_media <- alply(iris3,3,colMeans)
class(iris_media)
```

```
[1] "list"
```

```r
iris_media
```

```
$`1`
Sepal L. Sepal W. Petal L. Petal W.
   5.006    3.428    1.462    0.246

$`2`
Sepal L. Sepal W. Petal L. Petal W.
   5.936    2.770    4.260    1.326

$`3`
Sepal L. Sepal W. Petal L. Petal W.
   6.588    2.974    5.552    2.026

attr(,"split_type")
[1] "array"
attr(,"split_labels")
          X1
1     Setosa
2 Versicolor
3  Virginica
```

Entradas y argumentos

Las funciones en el paquete plyr aceptan varios objetos de entrada: data frames, matrices y listas. Cada objeto de entrada tiene su propia regla para dividir el proceso.

Las matrices se dividen por dimensión en submatrices de menor dimensión. La función correspondiente es `a*ply()`, donde una matriz es la entrada y la salida puede ser una matriz, un data frame o una lista.

Los data frames son divididos y puesto en subconjuntos por una combinación de variables del conjunto de datos de entrada. La función correspondiente es `d*ply()`, donde un data frame es la entrada y la salida puede ser una matriz, un data frame o una lista.

Los elementos de una lista se procesan por separado y la función común es `l*ply()`, donde la entrada común es una lista, y la salida puede ser una matriz, un data frame o otra lista .

Según el tipo de entrada, existen dos o tres argumentos principales para estas funciones: `a*ply ()`, `d*ply()` y `l*ply()`. Los siguientes son los principales argumentos para estas funciones comunes:

- `a*ply(.data, .margins, .fun, ..., .progress = "none")`
- `d*ply(.data, .variables, .fun, ..., .progress = "none")`
- `l*ply(.data, .fun, ..., .progress = "none")`

El primer argumento, `.data`, es el conjunto de datos de entrada que se debe procesar dividiéndolo y la salida se combinará de cada división. El argumento `.margins` o `.variables` especifica cómo se deben dividir los datos en partes más pequeñas. El argumento `.fun` especifica la tarea de procesamiento; esta puede ser cualquier función aplicable a cada división de la entrada.

Si omitimos el argumento `.fun`, los datos de entrada se convierten a la estructura de salida especificada por la función. Si queremos monitorear el progreso de la tarea de procesamiento, se debe especificar el argumento

`progress`. No se mostrará el estado de progreso por defecto.

En el siguiente ejemplo, veremos qué sucederá si no especificamos el argumento `.fun` en cualquier función del paquete `plyr`. Si damos la entrada como una matriz y queremos la salida como un data frame, pero no hemos dado el argumento `.fun`, la función `adply()` simplemente convertirá el objeto matriz en un data frame.

```
# convertiendo una matriz 3 dimensional a un data frame 2 dimensional
iris_dat <- adply(iris3, .margins=3)
class(iris_dat)
```

```
[1] "data.frame"
```

```
str(iris_dat)
```

```
'data.frame':   150 obs. of  5 variables:
 $ X1      : Factor w/ 3 levels "Setosa","Versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ Sepal L.: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal W.: num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal L.: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal W.: num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
```

El argumento `.margins` funciona de manera similar a la función `apply()` del paquete base de `R`. Hace lo siguiente:

- Divide una fila especificando `.margins = 1`
- Divide una columna especificando `.margins = 2`
- Divide partes individuales especificando `.margins = c (1,2)`.

El argumento `.margins` funciona para dimensiones más altas, utilizando combinatoria en el número de formas posibles de dividir la matriz.

Funciones multiargumento

A veces, tenemos que tratar con funciones que toman múltiples argumentos y los valores de cada argumento pueden provenir de un data frame , una lista o una matriz. El paquete `plyr` tiene funciones fáciles de usar para trabajar con funciones multiargumento. Veamos un ejemplo de generación de números aleatorios a partir de una distribución normal, con varias combinaciones de media y desviación estándar. Los valores de la media y la desviación estándar se almacenan en un data frame datos.

Ahora, generaremos números aleatorios usando las funciones R predeterminadas, como el ciclo `for` y también utilizando la función `mlply()` del paquete `plyr`. Los paramétros de combinación son dados en la siguiente tabla:

Tamaño muestral	Media	Desviación estándar
25	0	1
50	2	1.5
100	3.5	2
200	2.5	5
500	0.1	2

Con estas combinaciones de parámetros, generaremos números aleatorios normales usando `R` y `plyr` , como se muestra en el siguiente código:

```
# Definimos el conjunto de parametros

parametros.datos <- data.frame(n=c(25,50,100,200,400),
```

```
                            mean=c(0,2,3.5,2.5,0.1),
                            sd=c(1,1.5,2,5,2))
# Mostrando el conjunto de parametros
parametros.datos
```

```
    n mean  sd
1  25  0.0 1.0
2  50  2.0 1.5
3 100  3.5 2.0
4 200  2.5 5.0
5 400  0.1 2.0
```

```
# Variable aleatoria bivariada usando R
# set seed hace un ejemplo reproducible

set.seed(12345)

# Inicializamos un lista vacia para almacenr la variable generada
dato <- list()
for(i in 1:nrow(parametros.datos))
  {
  dato[[i]] <- rnorm(n=parametros.datos[i,1],
                     mean=parametros.datos[i,2],sd=parametros.datos[i,3])
}
# estimando la media desde los datos generados

estmedia <- lapply(dato,mean)
estmedia
```

```
[[1]]
[1] -0.001177287

[[2]]
[1] 2.417842

[[3]]
[1] 3.667193

[[4]]
[1] 2.999662

[[5]]
[1] 0.1765926
```

Realizamos la misma tarea pero esta vez usando el paquete `plyr`:

```
datos_plyr <- mlply(parametros.datos,rnorm)
estmedia_plyr <- llply(datos_plyr,mean)
estmedia_plyr
```

```
$`1`
[1] 0.4252469

$`2`
[1] 2.037528
```

```
$`3`
[1] 3.070231

$`4`
[1] 2.144276

$`5`
[1] 0.05399488
```

Comparando la base de R y plyr

En esta parte compararemos el código para resolver el mismo problema utilizando tanto `R` como `plyr` predeterminados. Reutilizando los datos de `iris3`, estamos interesados en producir estadísticas de de cinco números para cada grupo de variables por especies. Los cinco números serán el mínimo, la media, la mediana,el máximo y la desviación estándar.

El resultado será una lista de data frames. Para calcular las estadísticas de los cinco números, seguimos estos pasos:

1. Definimos una función que calculará las estadísticas de los cinco números para un vector dado.
2. Producimos la salida de esta función en un objeto data frame.
3. Aplicamos esta función en el conjunto de datos `iris3` usando un ciclo `for`.
4. Aplicamos la misma función usando la función `apply()` del paquete `plyr`.

Un ejemplo que explica el cálculo de las estadísticas de los cinco números es el siguiente:

```
# Funcion para calcula estadisticas  de 5 numeros
cinconum.resumen <- function(x)
  {
  resultados <-data.frame(min=apply(x,2,min),
                          media=apply(x,2,mean),
                          mediana=apply(x,2,median),
                          max=apply(x,2,max),
                          sd=apply(x,2,sd))
return(resultados)
}
```

En el siguiente código puede verse cómo calculamos los resúmenes de los cinco números usando un ciclo `for` de R :

```
# Inicializamos con un objeto lista de salida

todo_stats <- list()

# el bucle for se ejecutara para cada especie

for(i in 1:dim(iris3)[3])
  {
  sub_datos <- iris3[,,i]
  todo_stat_species <- cinconum.resumen(sub_datos)
  todo_stats[[i]] <- todo_stat_species
}
```

```r
# clase del objeto de salida
class(todo_stats)
```

```
[1] "list"
```

```r
todo_stats
```

```
[[1]]
         min media mediana max        sd
Sepal L. 4.3 5.006     5.0 5.8 0.3524897
Sepal W. 2.3 3.428     3.4 4.4 0.3790644
Petal L. 1.0 1.462     1.5 1.9 0.1736640
Petal W. 0.1 0.246     0.2 0.6 0.1053856

[[2]]
         min media mediana max        sd
Sepal L. 4.9 5.936    5.90 7.0 0.5161711
Sepal W. 2.0 2.770    2.80 3.4 0.3137983
Petal L. 3.0 4.260    4.35 5.1 0.4699110
Petal W. 1.0 1.326    1.30 1.8 0.1977527

[[3]]
         min media mediana max        sd
Sepal L. 4.9 6.588    6.50 7.9 0.6358796
Sepal W. 2.2 2.974    3.00 3.8 0.3224966
Petal L. 4.5 5.552    5.55 6.9 0.5518947
Petal W. 1.4 2.026    2.00 2.5 0.2746501
```

Calculemos las mismas estadísticas, pero esta vez usando la función `alply()` del paquete `plyr`:

```r
todo_stats <- alply(iris3,3,cinconum.resumen)
class(todo_stats)
```

```
[1] "list"
```

```r
todo_stats
```

```
$`1`
         min media mediana max        sd
Sepal L. 4.3 5.006     5.0 5.8 0.3524897
Sepal W. 2.3 3.428     3.4 4.4 0.3790644
Petal L. 1.0 1.462     1.5 1.9 0.1736640
Petal W. 0.1 0.246     0.2 0.6 0.1053856

$`2`
         min media mediana max        sd
Sepal L. 4.9 5.936    5.90 7.0 0.5161711
Sepal W. 2.0 2.770    2.80 3.4 0.3137983
Petal L. 3.0 4.260    4.35 5.1 0.4699110
Petal W. 1.0 1.326    1.30 1.8 0.1977527

$`3`
         min media mediana max        sd
Sepal L. 4.9 6.588    6.50 7.9 0.6358796
Sepal W. 2.2 2.974    3.00 3.8 0.3224966
Petal L. 4.5 5.552    5.55 6.9 0.5518947
Petal W. 1.4 2.026    2.00 2.5 0.2746501
```

```
attr(,"split_type")
[1] "array"
attr(,"split_labels")
          X1
1     Setosa
2 Versicolor
3  Virginica
```

Funciones eficientes

Para ilustrar cómo codificar funciones en R, considere el siguiente problema:

Queremos crear una función para generar variables aleatorias, $S = X_1 + \cdots + X_N$, con la convención $S = 0$ cuando $N = 0$, donde N es una variable de conteo, y los X_i son variables aleatorias IID. Supongamos que N se puede generar usando alguna función genérica `rN`, y las X_i usando `rX`. Por ejemplo, considere una variable compuesta de Poisson, con tamaños exponenciales.

```
rN.Poisson <- function(n) rpois(n,5)
rX.Exponencial <- function(n) rexp(n,2)
```

La primera idea es usar bucles,

```
rcpd1 <- function(n,rN=rN.Poisson,rX=rX.Exponencial){
  V <- rep(0,n)
  for(i in 1:n){
    N <- rN(1)
    if(N>0){V[i] <- sum(rX(N))}
  }
  return(V)}
```

En realidad, la suma de un vector vacío es nulo

```
sum(NULL)
```

```
[1] 0
```

Así la condición `if` no es necesaria aquí

```
 rcpd1 <- function(n,rN=rN.Poisson,rX=rX.Exponencial){
   V <- rep(0,n)
   for (i in 1:n) V[i] <- sum(rX(rN(1)))
   return(V)}
```

Las funciones basadas en las funciones `apply` (incluyendo `tapply`, `sapply` o `lapply`) se pueden usar para obtener sumas por línea, por columna o por tablas cruzadas. Tenga en cuenta que `lapply` y `sapply` toman una lista o vector como primer argumento y una función para aplicar a cada elemento como segundo argumento. La diferencia entre las dos funciones es que `lapply` devolverá su resultado en una lista, mientras que `sapply` simplificará su salida a un vector (o matriz). Para ver esto consideremos el máximo de 10 variables exponenciales de paramétro 1, por ejemplo:

```
set.seed(1)
M <- rep(NA,5)
for(i in 1:5) M[i] <- max(rexp(10))
M
```

```
[1] 2.894969 4.423934 3.958933 1.435285 2.007832
```

La misma salida se puede hacer con la función `replicate()`

```
replicate(5, max(rexp(10)))
```

```
[1] 3.217789 4.832813 2.909887 2.283853 1.043609
```

O usando `apply`

```
apply(matrix(rexp(10*5),10,5),2,max)
```

```
[1] 2.693412 1.835641 3.259337 2.073700 2.251153
```

Podemos usar esas funciones variantes de `apply` para generar variables aleatorias compuestas,

```
rcpd2 <- function(n,rN=rN.Poisson,rX=rX.Exponencial){
  N <- rN(n)
  X <- rX(sum(N))
  I <- factor(rep(1:n,N),levels=1:n)
  return(as.numeric(xtabs(X ~ I)))}
```

```
rcpd3 <- function(n,rN=rN.Poisson,rX=rX.Exponencial){
  N <- rN(n)
  X <- rX(sum(N))
  I <- factor(rep(1:n,N),levels=1:n)
  V <- tapply(X,I,sum)
  V[is.na(V)] <- 0
  return(as.numeric(V))}
```

```
rcpd4 <- function(n,rN=rN.Poisson,rX=rX.Exponencial){
  return(sapply(rN(n), function(x) sum(rX(x))))}
```

```
rcpd5 <- function(n,rN=rN.Poisson,rX=rX.Exponencial){
  return(sapply(Vectorize(rX)(rN(n)),sum))}
```

```
rcpd6 <- function(n,rN=rN.Poisson,rX=rX.Exponencial){
  return(unlist(lapply(lapply(t(rN(n)),rX),sum)))}
```

La función `unlist` imprime el resultado como un vector numérico, con componentes nombradas.

Para obtener más detalles sobre el cálculo del tiempo, podemos ejecutar 1000 cálculos

```
n <- 100
library(microbenchmark)
options(digits=1)
microbenchmark(rcpd1(n),rcpd2(n),rcpd3(n),rcpd4(n),rcpd5(n),rcpd6(n))
```

```
Unit: microseconds
     expr  min   lq mean median   uq  max neval
 rcpd1(n)  614  640  883    652  735 5855   100
 rcpd2(n) 1331 1479 1819   1554 1741 7771   100
 rcpd3(n)  582  607  780    619  691 7953   100
 rcpd4(n)  468  488  599    497  515 4395   100
 rcpd5(n)  566  621  855    639  700 5211   100
 rcpd6(n)  439  467  654    480  530 4120   100
```

Estas funciones se pueden usar también en data frames. Pero si es posible una representación matricial (en lugar de un dataframe), podría acelerar las cosas.

La función `Vectorize()` podría ser útil para optimizar el código y evitar bucles. Pero tratar de evitar bucles a toda costa probablemente no sea óptimo. En los párrafos anteriores, al escribir una función para generar una variable aleatoria compuesta de Poisson, hemos visto que el código basado en bucles en realidad era

bastante rápido. Pero más importante, a veces los bucles no se pueden evitar. Y eso probablemente no es un gran problema. Sea el siguiente ejemplo. Considere la siguiente lista `matriceslista` que contiene 10000 matrices de orden `n`, denota M_k

```
matriceslista <- vector(mode = "list", length = 100000)
for (i in seq_along(matriceslista)) matriceslista[[i]] <- matrix(rnorm(n^2),n,n)
```

El objetivo es calcular la matriz de orden `n` tal que $M = \sum M_k$, parecería natural almacenar matrices en un objeto más grande que una matriz (una matriz), y luego usar la función `apply`:

```
M <- apply(array(unlist(matriceslist),dim=c(n,n,10000)),1:2,sum)
```

Incluso si el código se ejecuta, tomará un tiempo cuando `n` es grande. Más precisamente, crearemos una matriz muy grande. ¿Por qué no usar un simple bucle?

```
M <- NULL; for(i in 1:100000) M=M+matriceslist[[i]]
```

Simplemente creamos un objeto que es solo una matriz de orden `n`.