EBook Gratis

APRENDIZAJE data.table

Free unaffiliated eBook created from
Stack Overflow contributors.

#data.table

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: data-table

It is an unofficial and free data.table ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official data.table.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Primeros pasos con data.table

Observaciones

Data.table es un paquete para el entorno informático estadístico R Amplía la funcionalidad de los marcos de datos desde la base R, mejorando particularmente su rendimiento y sintaxis. Una serie de tareas relacionadas, incluidas las combinaciones sucesivas y no equitativas, se manejan en una sintaxis concisa consistente como `DT[where, select|update|do, by]` .

Una serie de funciones complementarias también se incluyen en el paquete:

- I / O: `fread` / `fwrite`
- Remodelación: `melt` / `dcast` / `rbindlist` / `split`
- Corridas de valores: `rleid`

Versiones

Versión	Notas	Fecha de lanzamiento en CRAN
1.9.4		2014-10-02
1.9.6		2015-09-19
1.9.8		2016-11-24
1.10.0	"En retrospectiva, la última versión v1.9.8 debería haber sido nombrada v1.10.0"	2016-12-03
1.10.1	En desarrollo	2016-12-03

Examples

Instalación y configuración

Instale la versión estable de CRAN:

```
install.packages("data.table")
```

O la versión de desarrollo de github:

```
install.packages("data.table", type = "source",
  repos = "http://Rdatatable.github.io/data.table")
```

Para volver de devel a CRAN, primero se debe eliminar la versión actual:

```
remove.packages("data.table")
install.packages("data.table")
```

Visite el sitio web para obtener instrucciones de instalación completas y los últimos números de versión.

Usando el paquete

Normalmente querrá cargar el paquete y todas sus funciones con una línea como

```
library(data.table)
```

Si solo necesita una o dos funciones, puede referirse a ellas como `data.table::fread` .

Empezando y encontrando ayuda

El wiki oficial del paquete tiene algunos materiales esenciales:

- Como nuevo usuario, querrá revisar las viñetas, las preguntas frecuentes y la hoja de trucos .
- Antes de hacer una pregunta, aquí en StackOverflow o en cualquier otro lugar, lea la página de soporte .

Para obtener ayuda sobre funciones individuales, la sintaxis es `help("fread")` o `?fread` . Si el paquete no se ha cargado, use el nombre completo como `?data.table::fread` .

Sintaxis y características

Sintaxis basica

`DT[where, select|update|do, by]` sintaxis de `DT[where, select|update|do, by]` se utiliza para trabajar con columnas de una tabla de datos.

- La parte "donde" es el argumento `i`
- La parte "seleccionar | actualizar | hacer" es el argumento `j`

Estos dos argumentos generalmente se pasan por posición en lugar de por nombre.

Una secuencia de pasos se puede encadenar como `DT[...][...]` .

Atajos, funciones especiales y símbolos especiales dentro de `DT[...]`

Función o símbolo	Notas
`.()`	en varios argumentos, reemplaza `list()`
`J()`	en `i` , reemplaza la `list()`
`:=`	en `j` , una función utilizada para agregar o modificar columnas
`.N`	en `i` , el número total de filas en `j` , el número de filas en un grupo
`.I`	en `j` , el vector de los números de fila en la tabla (filtrado por `i`)
`.SD`	en `j` , el subconjunto actual de los datos seleccionado por el argumento `.SDcols`
`.GRP`	en `j` , el índice actual del subconjunto de los datos
`.BY`	en `j` , la lista de valores por el subconjunto actual de datos
`V1, V2, ...`	nombres predeterminados para columnas sin nombre creadas en `j`

Se une dentro de `DT[...]`

Notación	Notas
`DT1[DT2, on, j]`	unir dos mesas
`i.*`	prefijo especial en las columnas de DT2 después de la unión
`by=.EACHI`	Opción especial disponible solo con una unión
`DT1[!DT2, on, j]`	anti-unirse a dos mesas
`DT1[DT2, on, roll, j]`	unir dos tablas, rodando en la última columna en `on=`

Remodelación, apilado y fraccionamiento.

Notación	Notas
`melt(DT, id.vars, measure.vars)`	transformar a formato largo para columnas múltiples, use `measure.vars = patterns(...)`
`dcast(DT, formula)`	transformar a formato ancho
`rbind(DT1, DT2, ...)`	apilar datos enumerados.

Notación	Notas
`rbindlist(DT_list, idcol)`	apilar una lista de data.tables
`split(DT, by)`	dividir una tabla de datos en una lista

Algunas otras funciones especializadas para data.tables.

Función (es)	Notas
`foverlaps`	superposición de combinaciones
`merge`	otra forma de unir dos mesas
`set`	Otra forma de agregar o modificar columnas.
`fintersect` , `fsetdiff` , `funion` , `fsetequal` , `unique` , `duplicated` , `anyDuplicated`	Operaciones de teoría de conjuntos con filas como elementos.
`CJ`	El producto cartesiano de vectores.
`uniqueN`	el número de filas distintas
`rowidv(DT, cols)`	ID de fila (1 a .N) dentro de cada grupo determinado por cols
`rleidv(DT, cols)`	ID de grupo (1 a .GRP) dentro de cada grupo determinado por ejecuciones de cols
`shift(DT, n)`	aplicar un operador de turno a cada columna
`setorder` , `setcolorder` , `setnames` , `setkey` , `setindex` , `setattr`	Modificar atributos y ordenar por referencia.

Otras características del paquete.

Caracteristicas	Notas
`IDate` y `ITime`	fechas y horas enteras

Lea Primeros pasos con data.table en línea: https://riptutorial.com/es/data-

table/topic/3389/primeros-pasos-con-data-table

Capítulo 2: ¿Por qué mi antiguo código no funciona?

Introducción

El paquete data.table ha sufrido una serie de cambios e innovaciones a lo largo del tiempo. Aquí hay algunos escollos potenciales que pueden ayudar a los usuarios a mirar el código heredado o revisar las publicaciones antiguas del blog.

Examples

Único y duplicado ya no funciona en datos con clave.

Esto es para aquellos que se mueven a data.table> = 1.9.8

Tienes un conjunto de datos de dueños y nombres de mascotas, pero sospechas que se han capturado algunos datos repetidos.

```
library(data.table)
DT <- data.table(pet = c("dog","dog","cat","dog"),
                 owner = c("Alice","Bob","Charlie","Alice"),
                 entry.date = c("31/12/2015","31/12/2015","14/2/2016","14/2/2016"),
                 key = "owner")


> tables()
     NAME NROW NCOL MB COLS                 KEY
[1,] DT      4    3  1 pet,owner,entry.date owner
Total: 1MB
```

Recordar tecleando una tabla lo ordenará. Alice ha sido ingresada dos veces.

```
> DT
   pet   owner entry.date
1: dog   Alice 31/12/2015
2: dog   Alice  14/2/2016
3: dog     Bob 31/12/2015
4: cat Charlie  14/2/2016
```

Supongamos que usó `unique` método `unique` para deshacerse de los duplicados en sus datos en función de la clave, utilizando la fecha de captura de datos más reciente configurando de Último a VERDADERO.

1.9.8

```
clean.DT <- unique(DT, fromLast = TRUE)

> tables()
```

```
    NAME     NROW NCOL MB COLS                 KEY
[1,] clean.DT    3    3  1 pet,owner,entry.date owner
[2,] DT          4    3  1 pet,owner,entry.date owner
Total: 2MB
```

Alice duplicado ha sido eliminado.

1.9.8

```
clean.DT <- unique(DT, fromLast = TRUE)

> tables()
     NAME     NROW NCOL MB COLS                 KEY
[1,] clean.DT    4    3  1 pet,owner,entry.date owner
[2,] DT          4    3  1 pet,owner,entry.date owner
```

Esto no funciona. Todavía 4 filas!

Fijar

Utilice el parámetro `by=` **que ya no** utiliza de forma **predeterminada su clave,** sino todas las columnas.

```
clean.DT <- unique(DT, by = key(DT), fromLast = TRUE)
```

Ahora todo está bien.

```
> clean.DT
   pet   owner entry.date
1: dog   Alice  14/2/2016
2: dog     Bob 31/12/2015
3: cat Charlie  14/2/2016
```

Detalles y corrección provisional

Vea el artículo 1 en las notas de la versión de NOTICIAS para más detalles:

> Cambios en v1.9.8 (en CRAN 25 Nov 2016)
>
> POTENCIALMENTE QUE CAMBIAN
>
> 1. Por defecto, todas las columnas ahora se usan con los métodos de tabla de datos `unique()` , `duplicated()` y `uniqueN()` , # 1284 y # 1841. Para restaurar el comportamiento anterior: `options(datatable.old.unique.by.key=TRUE)` . En 1 año, esta opción para restaurar el valor predeterminado anterior quedará en desuso con una advertencia. En 2 años se eliminará la opción. Por favor pase explícitamente la `by=key(DT)` para mayor claridad. Sólo se ve afectado el código que se basa en el valor predeterminado. 266 paquetes de CRAN y

> bioconductores que usan data.table se verificaron antes de su lanzamiento. 9 necesarios para cambiar y fueron notificados. Cualquier línea de código sin cobertura de prueba se habrá perdido por estas verificaciones. Cualquier paquete que no esté en CRAN o Bioconductor no fue verificado.

Por lo tanto, puede usar las opciones como una solución temporal hasta que su código esté arreglado.

```
options(datatable.old.unique.by.key=TRUE)
```

Lea ¿Por qué mi antiguo código no funciona? en línea: https://riptutorial.com/es/data-table/topic/8196/-por-que-mi-antiguo-codigo-no-funciona-

Capítulo 3: Añadiendo y modificando columnas.

Observaciones

La viñeta oficial, "Semántica de referencia" , es la mejor introducción a este tema.

Un recordatorio: la sintaxis de `DT[where, select|update|do, by]` se utiliza para trabajar con columnas de una tabla de datos.

- La parte "donde" es el argumento `i`
- La parte "seleccionar | actualizar | hacer" es el argumento `j`

Estos dos argumentos generalmente se pasan por posición en lugar de por nombre.

Todas las modificaciones a las columnas se pueden hacer en `j` . Además, la función de `set` está disponible para este uso.

Examples

Editando valores

```
# example data
DT = as.data.table(mtcars, keep.rownames = TRUE)
```

Editando una columna

Use el operador `:=` dentro de `j` para crear nuevas columnas o modificar las existentes:

```
DT[, mpg_sq := mpg^2]
```

Edición en un subconjunto de filas

Utilice el argumento `i` para subcontratar a las filas "donde" se deben realizar las ediciones:

```
DT[1:3, newvar := "Hello"]
```

Al igual que en un data.frame, podemos subcontratar utilizando números de fila o pruebas lógicas. También es posible utilizar [una "unión" en `i` cuando se modifica] [need_a_link].

Quitando una columna

Elimine las columnas estableciendo `NULL` :

```
DT[, mpg_sq := NULL]
```

Tenga en cuenta que no `<-` asignamos el resultado, ya que `DT` se ha modificado in situ.

Editando multiples columnas

Agregue varias columnas usando el formato multivariado del operador `:=`

```
DT[, `:=`(mpg_sq = mpg^2, wt_sqrt = sqrt(wt))]
# or
DT[, c("mpg_sq", "wt_sqrt") := .(mpg^2, sqrt(wt))]
```

La sintaxis `.()` Se usa cuando el lado derecho de `LHS := RHS` es una lista de columnas.

Edición de múltiples columnas dependientes secuencialmente

Si las columnas son dependientes y deben definirse en secuencia, algunas formas de hacerlo son:

```
DT[, c("mpg_sq", "mpg2_hp") := .(temp1 <- mpg^2, temp1/hp)]
# or
DT[, c("mpg_sq", "mpg2_hp") := {temp1 = mpg^2; .(temp1, temp1/hp)}]
```

Edición de columnas por nombres determinados dinámicamente.

Para nombres de columna determinados dinámicamente, use paréntesis:

```
vn = "mpg_sq"
DT[, (vn) := mpg^2]
```

Usando `set`

Las columnas también pueden modificarse con el `set` para una pequeña reducción en la

sobrecarga, aunque esto rara vez es necesario:

```
set(DT, j = "hp_over_wt", v = mtcars$hp/mtcars$wt)
```

Reordenar columnas

```
# example data
DT = as.data.table(mtcars, keep.rownames = TRUE)
```

Para reorganizar el orden de las columnas, use `setcolorder` . Por ejemplo, para revertirlos.

```
setcolorder(DT, rev(names(DT)))
```

Esto no cuesta casi nada en términos de rendimiento, ya que solo está permutando la lista de punteros de columna en la tabla de datos.

Renombrando columnas

```
# example data
DT = as.data.table(mtcars, keep.rownames = TRUE)
```

Para cambiar el nombre de una columna (mientras se mantienen sus datos iguales), no es necesario copiar los datos a una columna con un nombre nuevo y eliminar el anterior. En su lugar, podemos usar

```
setnames(DT, "mpg_sq", "mpq_squared")
```

Para modificar la columna original por referencia.

Modificación de niveles de factor y otros atributos de columna.

```
# example data
DT = data.table(iris)
```

Para modificar los niveles de factor por referencia, use `setattr` :

```
setattr(DT$Species, "levels", c("set", "ver", "vir")
# or
DT[, setattr(Species, "levels", c("set", "ver", "vir"))]
```

La segunda opción podría imprimir el resultado en la pantalla.

Con `setattr` , evitamos la copia en la que se incurre al realizar los `levels(x) <- lvls` , pero también omite algunas comprobaciones, por lo que es importante tener cuidado de asignar un vector de niveles válido.

Lea Añadiendo y modificando columnas. en línea: https://riptutorial.com/es/data-

table/topic/3781/anadiendo-y-modificando-columnas-

Capítulo 4: Creando una tabla de datos

Observaciones

Un data.table es una versión mejorada de la clase data.frame desde la base R. Como tal, su atributo `class()` es el vector `"data.table" "data.frame"` y las funciones que funcionan en un data.frame también Trabajar con una tabla de datos. Hay muchas formas de crear, cargar o forzar una tabla de datos, como se ve aquí.

Examples

Coaccionar un data.frame

Para copiar un data.frame como data.table, use `as.data.table` o `data.table` :

```
DF = data.frame(x = letters[1:5], y = 1:5, z = (1:5) > 3)

DT <- as.data.table(DF)
# or
DT <- data.table(DF)
```

Esto rara vez es necesario. Una excepción es cuando se usan conjuntos de datos `mtcars` como `mtcars` , que deben copiarse ya que no se pueden modificar in situ.

Construir con data.table ()

Hay un constructor del mismo nombre:

```
DT <- data.table(
  x = letters[1:5],
  y = 1:5,
  z = (1:5) > 3
)
#    x y     z
# 1: a 1 FALSE
# 2: b 2 FALSE
# 3: c 3 FALSE
# 4: d 4  TRUE
# 5: e 5  TRUE
```

A diferencia de `data.frame` , `data.table` no `data.table` cadenas a factores por defecto:

```
sapply(DT, class)
#               x           y           z
#     "character"   "integer"   "logical"
```

Leer con fread ()

Podemos leer desde un archivo de texto:

```
dt <- fread("my_file.csv")
```

A diferencia de `read.csv` , `fread` leerá cadenas como cadenas, no como factores por defecto.

Vea el [tema en `fread`] [need_a_link] para más ejemplos.

Modificar un data.frame con setDT ()

Para la eficiencia, data.table ofrece una forma de alterar un data.frame o lista para hacer una data.table in situ:

```
# example data.frame
DF = data.frame(x = letters[1:5], y = 1:5, z = (1:5) > 3)

# modification
setDT(DF)
```

Tenga en cuenta que no `<-` asignamos el resultado, ya que el objeto `DF` se ha modificado in situ.

Los atributos de clase del data.frame se mantendrán:

```
sapply(DF, class)
#         x         y         z
#  "factor" "integer" "logical"
```

Copia otra tabla de datos con copia ()

```
# example data
DT1 = data.table(x = letters[1:2], y = 1:2, z = (1:2) > 3)
```

Debido a la forma en que se manipulan las tablas de datos, `DT2 <- DT1` *no* hará una copia. Es decir, las modificaciones posteriores a las columnas u otros atributos de `DT2` también afectarán a `DT1` . Cuando quieras una copia real, usa

```
DT2 = copy(DT1)
```

Para ver la diferencia, esto es lo que sucede sin una copia:

```
DT2 <- DT1
DT2[, w := 1:2]

DT1
#    x y     z w
# 1: a 1 FALSE 1
# 2: b 2 FALSE 2
DT2
#    x y     z w
# 1: a 1 FALSE 1
# 2: b 2 FALSE 2
```

Y con una copia:

```
DT2 <- copy(DT1)
DT2[, w := 1:2]

DT1
#    x y     z
# 1: a 1 FALSE
# 2: b 2 FALSE
DT2
#    x y     z w
# 1: a 1 FALSE 1
# 2: b 2 FALSE 2
```

Así que los cambios no se propagan en este último caso.

Lea Creando una tabla de datos en línea: https://riptutorial.com/es/data-table/topic/3782/creando-una-tabla-de-datos

Capítulo 5: Datos de limpieza

Examples

Manejo de duplicados

```
# example data
DT = data.table(id = c(1,2,2,3,3,3))[, v := LETTERS[.I]][]
```

Para tratar los "duplicados", combine filas de conteo en un grupo y subconjunto de filas por grupo .

Mantener una fila por grupo

Aka "soltar duplicados" alias "deduplicate" alias "uniquify".

```
unique(DT, by="id")
# or
DT[, .SD[1L], by=id]
#    id v
# 1:  1 A
# 2:  2 B
# 3:  3 D
```

Esto mantiene la primera fila. Para seleccionar una fila diferente, se puede jugar con la parte `1L` o usar el `order` en `i` .

Mantener solo filas únicas

```
DT[, if (.N == 1L) .SD, by=id]
#    id v
# 1:  1 A
```

Mantener sólo filas no únicas

```
DT[, if (.N > 1L) .SD, by=id]
#    id v
# 1:  2 B
# 2:  2 C
# 3:  3 D
# 4:  3 E
# 5:  3 F
```

Lea Datos de limpieza en línea: https://riptutorial.com/es/data-table/topic/5206/datos-de-limpieza

Capítulo 6: Informes estadísticos de resumen.

Observaciones

Un recordatorio: la sintaxis de `DT[where, select|update|do, by]` se utiliza para trabajar con columnas de una tabla de datos.

- La parte "donde" es el argumento `i`
- La parte "seleccionar | actualizar | hacer" es el argumento `j`

Estos dos argumentos generalmente se pasan por posición en lugar de por nombre.

Examples

Contando filas por grupo

```
# example data
DT = data.table(iris)
DT[, Bin := cut(Sepal.Length, c(4,6,8))]
```

Utilizando `.N`

`.N` en `j` almacena el número de filas en un subconjunto. Al explorar datos, `.N` es útil para ...

1. contar filas en un grupo,

```
DT[Species == "setosa", .N]

# 50
```

2. o contar filas en todos los grupos,

```
DT[, .N, by=.(Species, Bin)]

#       Species   Bin  N
# 1:     setosa (4,6] 50
# 2: versicolor (6,8] 20
# 3: versicolor (4,6] 30
# 4:  virginica (6,8] 41
# 5:  virginica (4,6]  9
```

3. o encuentra grupos que tengan un cierto número de filas.

```
DT[, .N, by=.(Species, Bin)][ N < 25 ]
```

```
#       Species   Bin  N
# 1: versicolor (6,8] 20
# 2:  virginica (4,6]  9
```

Manejo de grupos faltantes

Sin embargo, faltan grupos con un conteo de cero arriba. Si importan, podemos usar la `table` desde la base:

```
DT[, data.table(table(Species, Bin))][ N < 25 ]

#       Species   Bin  N
# 1:  virginica (4,6]  9
# 2:     setosa (6,8]  0
# 3: versicolor (6,8] 20
```

Alternativamente, podemos unirnos en todos los grupos:

```
DT[CJ(Species=Species, Bin=Bin, unique=TRUE), on=c("Species","Bin"), .N, by=.EACHI][N < 25]

#       Species   Bin  N
# 1:     setosa (6,8]  0
# 2: versicolor (6,8] 20
# 3:  virginica (4,6]  9
```

Una nota en `.N` :

- Este ejemplo utiliza `.N` en `j` , donde se refiere al tamaño de un subconjunto.
- En `i` , se refiere al número total de filas.

Resúmenes personalizados

```
# example data
DT = data.table(iris)
DT[, Bin := cut(Sepal.Length, c(4,6,8))]
```

Supongamos que queremos la salida de la función de `summary` para `Sepal.Length` junto con el número de observaciones:

```
DT[, c(
    as.list(summary(Sepal.Length)),
    N = .N
), by=.(Species, Bin)]

#       Species   Bin Min. 1st Qu. Median  Mean 3rd Qu. Max.  N
# 1:     setosa (4,6]  4.3     4.8    5.0 5.006     5.2  5.8 50
# 2: versicolor (6,8]  6.1     6.2    6.4 6.450     6.7  7.0 20
# 3: versicolor (4,6]  4.9     5.5    5.6 5.593     5.8  6.0 30
# 4:  virginica (6,8]  6.1     6.4    6.7 6.778     7.2  7.9 41
# 5:  virginica (4,6]  4.9     5.7    5.8 5.722     5.9  6.0  9
```

Tenemos que hacer `j` una lista de columnas. Por lo general, algunos juegan con `c` , `as.list` y `.` es suficiente para averiguar la forma correcta de proceder.

Asignación de estadísticas de resumen como nuevas columnas.

En lugar de hacer una tabla de resumen, es posible que desee almacenar una estadística de resumen en una nueva columna. Podemos usar `:=` como siempre. Por ejemplo,

```
DT[, is_big := .N >= 25, by=.(Species, Bin)]
```

Escollos

Datos desordenados

Si te encuentras con ganas de analizar los nombres de las columnas, como

> Tome la media de `x.Length/x.Width` donde `x` toma diez valores diferentes.

entonces es probable que esté viendo datos incrustados en nombres de columna, lo cual es una mala idea. Lea acerca de los datos ordenados y luego cambie a formato largo.

Resúmenes de Rowwise

Los marcos de datos y data.tables están bien diseñados para datos tabulares, donde las filas corresponden a observaciones y columnas a variables. Si te encuentras con ganas de resumir sobre filas, como

> Encuentra la desviación estándar a través de columnas para cada fila.

entonces probablemente deberías usar una matriz o algún otro formato de datos por completo.

La función de resumen

```
# example data
DT = data.table(iris)
DT[, Bin := cut(Sepal.Length, c(4,6,8))]
```

`summary` es útil para navegar por las estadísticas de resumen. Además del uso directo como `summary(DT)` , también se puede aplicar por grupo convenientemente con `split` :

```
lapply(split(DT, by=c("Species", "Bin"), drop=TRUE, keep.by=FALSE), summary)

# $`setosa.(4,6]`
```

```
#   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
#  Min.   :4.300   Min.   :2.300   Min.   :1.000   Min.   :0.100
#  1st Qu.:4.800   1st Qu.:3.200   1st Qu.:1.400   1st Qu.:0.200
#  Median :5.000   Median :3.400   Median :1.500   Median :0.200
#  Mean   :5.006   Mean   :3.428   Mean   :1.462   Mean   :0.246
#  3rd Qu.:5.200   3rd Qu.:3.675   3rd Qu.:1.575   3rd Qu.:0.300
#  Max.   :5.800   Max.   :4.400   Max.   :1.900   Max.   :0.600
#
# $`versicolor.(6,8]`
#   Sepal.Length   Sepal.Width    Petal.Length    Petal.Width
#  Min.   :6.10   Min.   :2.20   Min.   :4.000   Min.   :1.20
#  1st Qu.:6.20   1st Qu.:2.80   1st Qu.:4.400   1st Qu.:1.30
#  Median :6.40   Median :2.90   Median :4.600   Median :1.40
#  Mean   :6.45   Mean   :2.89   Mean   :4.585   Mean   :1.42
#  3rd Qu.:6.70   3rd Qu.:3.10   3rd Qu.:4.700   3rd Qu.:1.50
#  Max.   :7.00   Max.   :3.30   Max.   :5.000   Max.   :1.70
#
# [...results truncated...]
```

Para incluir grupos de conteo cero, establezca `drop=FALSE` en `split` .

Aplicando una función de resumen a múltiples variables.

```
# example data
DT = data.table(iris)
DT[, Bin := cut(Sepal.Length, c(4,6,8))]
```

Para aplicar la misma función de resumen a cada columna por grupo, podemos usar `lapply` y `.SD`

```
DT[, lapply(.SD, median), by=.(Species, Bin)]

#       Species   Bin Sepal.Length Sepal.Width Petal.Length Petal.Width
# 1:     setosa (4,6]          5.0         3.4         1.50         0.2
# 2: versicolor (6,8]          6.4         2.9         4.60         1.4
# 3: versicolor (4,6]          5.6         2.7         4.05         1.3
# 4:  virginica (6,8]          6.7         3.0         5.60         2.1
# 5:  virginica (4,6]          5.8         2.7         5.00         1.9
```

Podemos filtrar las columnas en `.SD` con el argumento `.SDcols` :

```
DT[, lapply(.SD, median), by=.(Species, Bin), .SDcols="Petal.Length"]

#       Species   Bin Petal.Length
# 1:     setosa (4,6]         1.50
# 2: versicolor (6,8]         4.60
# 3: versicolor (4,6]         4.05
# 4:  virginica (6,8]         5.60
# 5:  virginica (4,6]         5.00
```

Múltiples funciones de resumen.

Actualmente, la extensión más simple a múltiples funciones es quizás:

```
DT[, unlist(recursive=FALSE, lapply(
    .(med = median, iqr = IQR),
    function(f) lapply(.SD, f)
)), by=.(Species, Bin), .SDcols=Petal.Length:Petal.Width]

#       Species   Bin med.Petal.Length med.Petal.Width iqr.Petal.Length iqr.Petal.Width
# 1:     setosa (4,6]             1.50             0.2            0.175           0.100
# 2: versicolor (6,8]             4.60             1.4            0.300           0.200
# 3: versicolor (4,6]             4.05             1.3            0.525           0.275
# 4:  virginica (6,8]             5.60             2.1            0.700           0.500
# 5:  virginica (4,6]             5.00             1.9            0.200           0.200
```

Si desea que los nombres sean como `Petal.Length.med` lugar de `med.Petal.Length` , cambie el orden:

```
DT[, unlist(recursive=FALSE, lapply(
    .SD,
    function(x) lapply(.(med = median, iqr = IQR), function(f) f(x))
)), by=.(Species, Bin), .SDcols=Petal.Length:Petal.Width]

#       Species   Bin Petal.Length.med Petal.Length.iqr Petal.Width.med Petal.Width.iqr
# 1:     setosa (4,6]             1.50            0.175             0.2           0.100
# 2: versicolor (6,8]             4.60            0.300             1.4           0.200
# 3: versicolor (4,6]             4.05            0.525             1.3           0.275
# 4:  virginica (6,8]             5.60            0.700             2.1           0.500
# 5:  virginica (4,6]             5.00            0.200             1.9           0.200
```

Lea Informes estadísticos de resumen. en línea: https://riptutorial.com/es/data-table/topic/3785/informes-estadisticos-de-resumen-

Capítulo 7: Remodelación, apilado y fraccionamiento.

Observaciones

La viñeta oficial, "Remodelación eficiente utilizando data.tables" , es la mejor introducción a este tema.

Muchas tareas de remodelación requieren moverse entre formatos largos y anchos:

- Los datos anchos son datos con cada columna que representa una variable separada, y filas que representan observaciones separadas
- Los datos largos son datos con el ID de formulario | variable | valor, donde cada fila representa un par de observación-variable

Examples

fundir y fundir con data.table

`data.table` ofrece una amplia gama de posibilidades para remodelar sus datos de manera eficiente y fácil

Por ejemplo, mientras se remodela de largo a ancho, ambos pueden pasar varias variables a `value.var` y a los parámetros `fun.aggregate` al mismo tiempo

```
library(data.table) #v>=1.9.6
DT <- data.table(mtcars)
```

Largo a ancho

```
dcast(DT, gear ~ cyl, value.var = c("disp", "hp"), fun = list(mean, sum))
   gear disp_mean_4 disp_mean_6 disp_mean_8 hp_mean_4 hp_mean_6 hp_mean_8 disp_sum_4
disp_sum_6 disp_sum_8 hp_sum_4 hp_sum_6 hp_sum_8
1:    3     120.100       241.5    357.6167        97     107.5  194.1667      120.1
483.0     4291.4       97      215     2330
2:    4     102.625       163.8         NaN        76     116.5       NaN      821.0
655.2       0.0      608      466         0
3:    5     107.700       145.0    326.0000       102     175.0  299.5000      215.4
145.0      652.0      204      175      599
```

Esto establecerá el `gear` como la columna de índice, mientras que la `mean` y la `sum` se calcularán para `disp` y `hp` para cada combinación de `gear` y `cyl` . En caso de que no existan algunas combinaciones, puede especificar parámetros adicionales como `na.rm = TRUE` (que se pasará a `mean` funciones de `mean` y `sum`) o especificar el argumento de `fill` incorporado. También puede agregar márgenes, eliminar combinaciones que faltan y subcontratar los datos. Ver más en `?data.table::dcast`

Ancho a largo

Mientras cambia de ancho a largo, puede pasar columnas al parámetro `measure.vars` usando expresiones regulares, por ejemplo

```
print(melt(DT, c("cyl", "gear"), measure = patterns("^d", "e")), n = 10)
    cyl gear variable value1 value2
 1:   6    4        1 160.00  16.46
 2:   6    4        1 160.00  17.02
 3:   4    4        1 108.00  18.61
 4:   6    3        1 258.00  19.44
 5:   8    3        1 360.00  17.02
---
60:   4    5        2   3.77   5.00
61:   8    5        2   4.22   5.00
62:   6    5        2   3.62   5.00
63:   8    5        2   3.54   5.00
64:   4    4        2   4.11   4.00
```

Esto `melt` los datos por `cyl` y `gear` como las columnas de índice, mientras que todos los valores para las variables que comienzan con `d` (`disp` & `drat`) estarán presentes en `value1` y los valores para las variables que contienen la letra `e` en ellos (`qsec` y `gear`) estarán presentes en la columna `value2` .

También puede cambiar el nombre de todos los nombres de columna en el resultado al especificar los argumentos `variable.name` y `value.name` o decidir si desea que las columnas de `character` se conviertan automáticamente en `factor` s o no al especificar los argumentos `variable.factor` y `value.factor` . Ver más en `?data.table::melt`

Remodelar usando `data.table`

`data.table` extiende las `reshape2` de `melt` y `dcast`

(Referencia: remodelación eficiente utilizando data.tables)

```
library(data.table)

## generate some data
dt <- data.table(
  name = rep(c("firstName", "secondName"), each=4),
  numbers = rep(1:4, 2),
  value = rnorm(8)
)
dt
#          name numbers      value
# 1:  firstName       1 -0.8551881
# 2:  firstName       2 -1.0561946
# 3:  firstName       3  0.2671833
# 4:  firstName       4  1.0662379
# 5: secondName       1 -0.4771341
# 6: secondName       2  1.2830651
# 7: secondName       3 -0.6989682
# 8: secondName       4 -0.6592184
```

Largo a ancho

```
dcast(data = dt,
      formula = name ~ numbers,
      value.var = "value")

#          name          1          2         3         4
# 1:  firstName  0.1836433 -0.8356286 1.5952808 0.3295078
# 2: secondName -0.8204684  0.4874291 0.7383247 0.5757814
```

En columnas múltiples (a partir de `data.table` 1.9.6)

```
## add an extra column
dt[, value2 := value * 2]

## cast multiple value columns
dcast(data = dt,
      formula = name ~ numbers,
      value.var = c("value", "value2"))

#          name    value_1    value_2   value_3   value_4   value2_1   value2_2 value2_3
value2_4
# 1:  firstName  0.1836433 -0.8356286 1.5952808 0.3295078  0.3672866 -1.6712572 3.190562
0.6590155
# 2: secondName -0.8204684  0.4874291 0.7383247 0.5757814 -1.6409368  0.9748581 1.476649
1.1515627
```

Ancho a largo

```
## use a wide data.table
dt <- fread("name          1          2         3         4
firstName  0.1836433 -0.8356286 1.5952808 0.3295078
secondName -0.8204684  0.4874291 0.7383247 0.5757814", header = T)
dt
#          name          1          2         3         4
# 1:  firstName  0.1836433 -0.8356286 1.5952808 0.3295078
# 2: secondName -0.8204684  0.4874291 0.7383247 0.5757814

## melt to long, specifying the id column, and the name of the columns
## in the resulting long data.table
melt(dt,
     id.vars = "name",
     variable.name = "numbers",
     value.name = "myValue")
#          name  numbers    myValue
# 1:  firstName        1  0.1836433
# 2: secondName        1 -0.8204684
# 3:  firstName        2 -0.8356286
# 4: secondName        2  0.4874291
# 5:  firstName        3  1.5952808
# 6: secondName        3  0.7383247
# 7:  firstName        4  0.3295078
# 8: secondName        4  0.5757814
```

Pasando de formato ancho a largo usando melt

Derritiendo: Los fundamentos

La fusión se utiliza para transformar datos de formato ancho a largo.

Comenzando con un amplio conjunto de datos:

```
DT = data.table(ID = letters[1:3], Age = 20:22, OB_A = 1:3, OB_B = 4:6, OB_C = 7:9)
```

Podemos fundir nuestros datos utilizando la función de `melt` en data.table. Esto devuelve otra tabla de datos en formato largo:

```
melt(DT, id.vars = c("ID","Age"))
1:  a  20     OB_A     1
2:  b  21     OB_A     2
3:  c  22     OB_A     3
4:  a  20     OB_B     4
5:  b  21     OB_B     5
6:  c  22     OB_B     6
7:  a  20     OB_C     7
8:  b  21     OB_C     8
9:  c  22     OB_C     9

class(melt(DT, id.vars = c("ID","Age")))
# "data.table" "data.frame"
```

Se asume que las columnas que no `id.vars` establecidas en el parámetro `id.vars` son variables. Alternativamente, podemos establecer estos explícitamente usando el argumento `measure.vars` :

```
melt(DT, measure.vars = c("OB_A","OB_B","OB_C"))
   ID Age variable value
1:  a  20     OB_A     1
2:  b  21     OB_A     2
3:  c  22     OB_A     3
4:  a  20     OB_B     4
5:  b  21     OB_B     5
6:  c  22     OB_B     6
7:  a  20     OB_C     7
8:  b  21     OB_C     8
9:  c  22     OB_C     9
```

En este caso, se asume que todas las columnas no establecidas en `measure.vars` son ID.

Si configuramos ambos explícitamente, solo devolverá las columnas seleccionadas:

```
melt(DT, id.vars = "ID", measure.vars = c("OB_C"))
   ID variable value
1:  a     OB_C     7
2:  b     OB_C     8
3:  c     OB_C     9
```

Nombrando variables y valores en el

resultado.

Podemos manipular los nombres de columna de la tabla devuelta usando `variable.name` y `value.name`

```
melt(DT,
     id.vars = c("ID"),
     measure.vars = c("OB_C"),
     variable.name = "Test",
     value.name = "Result"
     )
   ID Test Result
1:  a OB_C      7
2:  b OB_C      8
3:  c OB_C      9
```

Configuración de tipos de variables de medida en el resultado.

Por defecto, la fusión de una data.table convierte todos los `measure.vars` de factores:

```
M_DT <- melt(DT,id.vars = c("ID"), measure.vars = c("OB_C"))
class(M_DT[, variable])
# "factor"
```

Para establecer como carácter en su lugar, use el argumento `variable.factor` :

```
M_DT <- melt(DT,id.vars = c("ID"), measure.vars = c("OB_C"), variable.factor = FALSE)
class(M_DT[, variable])
# "character"
```

Los valores generalmente se heredan del tipo de datos de la columna de origen:

```
class(DT[, value])
# "integer"
class(M_DT[, value])
# "integer"
```

Si hay un conflicto, los tipos de datos serán coaccionados. Por ejemplo:

```
M_DT <- melt(DT,id.vars = c("Age"), measure.vars = c("ID","OB_C"))
class(M_DT[, value])
# "character"
```

Al fundirse, cualquier variable factor será coaccionada al tipo de carácter:

```
DT[, OB_C := factor(OB_C)]
M_DT <- melt(DT,id.vars = c("ID"), measure.vars = c("OB_C"))
```

```
class(M_DT)
# "character"
```

Para evitar esto y conservar la escritura inicial, use el argumento `value.factor` :

```
M_DT <- melt(DT,id.vars = c("ID"), measure.vars = c("OB_C"), value.factor = TRUE)
class(M_DT)
# "factor"
```

Manejo de valores perdidos

Por defecto, cualquier valor de `NA` se conserva en los datos fundidos.

```
DT = data.table(ID = letters[1:3], Age = 20:22, OB_A = 1:3, OB_B = 4:6, OB_C = c(7:8,NA))
melt(DT,id.vars = c("ID"), measure.vars = c("OB_C"))
   ID variable value
1:  a     OB_C     7
2:  b     OB_C     8
3:  c     OB_C    NA
```

Si se deben eliminar de sus datos, establezca `na.rm = TRUE`

```
melt(DT,id.vars = c("ID"), measure.vars = c("OB_C"), na.rm = TRUE)
   ID variable value
1:  a     OB_C     7
2:  b     OB_C     8
```

Pasando de formato largo a ancho usando dcast

Casting: Los fundamentos

La conversión se utiliza para transformar datos de formato largo a ancho.

Comenzando con un conjunto de datos largo:

```
DT = data.table(ID = rep(letters[1:3],3), Age = rep(20:22,3), Test =
rep(c("OB_A","OB_B","OB_C"), each = 3), Result = 1:9)
```

Podemos echar nuestras datos utilizando el `dcast` función en data.table. Esto devuelve otra tabla de datos en formato ancho:

```
dcast(DT, formula = ID ~ Test, value.var = "Result")
   ID OB_A OB_B OB_C
1:  a    1    4    7
2:  b    2    5    8
3:  c    3    6    9

class(dcast(DT, formula = ID ~ Test, value.var = "Result"))
```

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

Lanzar un valor

Un argumento `value.var` es necesario para una `value.var` adecuada; si no se proporciona, el dcast se basará en sus datos.

```
dcast(DT, formula = ID ~ Test, value.var = "Result")
   ID OB_A OB_B OB_C
1:  a    1    4    7
2:  b    2    5    8
3:  c    3    6    9

   ID OB_A OB_B OB_C
1:  a   20   20   20
2:  b   21   21   21
3:  c   22   22   22
```

Se pueden proporcionar múltiples `value.var` s en una lista

```
dcast(DT, formula = ID ~ Test, value.var = list("Result","Age"))
   ID Result_OB_A Result_OB_B Result_OB_C Age_OB_A Age_OB_B Age_OB_C
1:  a           1           4           7       20       20       20
2:  b           2           5           8       21       21       21
3:  c           3           6           9       22       22       22
```

Fórmula

El casting se controla mediante el argumento de fórmula en `dcast` . Esto es de la forma ROWS ~ COLUMNAS

```
dcast(DT, formula = ID ~ Test, value.var = "Result")
   ID OB_A OB_B OB_C
1:  a    1    4    7
2:  b    2    5    8
3:  c    3    6    9

dcast(DT, formula = Test ~ ID, value.var = "Result")
   Test a b c
1: OB_A 1 2 3
2: OB_B 4 5 6
3: OB_C 7 8 9
```

Tanto las filas como las columnas se pueden expandir con otras variables usando `+`

```
dcast(DT, formula = ID + Age ~ Test, value.var = "Result")
   ID Age OB_A OB_B OB_C
1:  a  20    1    4    7
2:  b  21    2    5    8
3:  c  22    3    6    9
```

```
dcast(DT, formula = ID ~ Age + Test, value.var = "Result")
   ID 20_OB_A 20_OB_B 20_OB_C 21_OB_A 21_OB_B 21_OB_C 22_OB_A 22_OB_B 22_OB_C
1:  a       1       4       7      NA      NA      NA      NA      NA      NA
2:  b      NA      NA      NA       2       5       8      NA      NA      NA
3:  c      NA      NA      NA      NA      NA      NA       3       6       9

#order is important

dcast(DT, formula = ID ~ Test + Age, value.var = "Result")
   ID OB_A_20 OB_A_21 OB_A_22 OB_B_20 OB_B_21 OB_B_22 OB_C_20 OB_C_21 OB_C_22
1:  a       1      NA      NA       4      NA      NA       7      NA      NA
2:  b      NA       2      NA      NA       5      NA      NA       8      NA
3:  c      NA      NA       3      NA      NA       6      NA      NA       9
```

El lanzamiento a menudo puede crear celdas donde no existe observación en los datos. Por defecto, esto se denota por `NA` , como se indica arriba. Podemos anular esto con el argumento `fill=` .

```
dcast(DT, formula = ID ~ Test + Age, value.var = "Result", fill = 0)
   ID OB_A_20 OB_A_21 OB_A_22 OB_B_20 OB_B_21 OB_B_22 OB_C_20 OB_C_21 OB_C_22
1:  a       1       0       0       4       0       0       7       0       0
2:  b       0       2       0       0       5       0       0       8       0
3:  c       0       0       3       0       0       6       0       0       9
```

También puede usar dos variables especiales en el objeto de fórmula

- . no representa otras variables
- ... representa todas las demás variables

```
dcast(DT, formula = Age ~ ., value.var = "Result")
   Age .
1:  20 3
2:  21 3
3:  22 3

dcast(DT, formula = ID + Age ~ ..., value.var = "Result")
   ID Age OB_A OB_B OB_C
1:  a  20    1    4    7
2:  b  21    2    5    8
3:  c  22    3    6    9
```

Agregando nuestro valor.

También podemos emitir y agregar valores en un solo paso. En este caso, tenemos tres observaciones en cada una de las intersecciones de Edad e ID. Para establecer qué agregación queremos, usamos el argumento `fun.aggregate` :

```
#length
dcast(DT, formula = ID ~ Age, value.var = "Result", fun.aggregate = length)
   ID 20 21 22
1:  a  3  0  0
2:  b  0  3  0
```

```
3:  c  0  0  3

#sum
dcast(DT, formula = ID ~ Age, value.var = "Result", fun.aggregate = sum)
   ID 20 21 22
1:  a 12  0  0
2:  b  0 15  0
3:  c  0  0 18

#concatenate
dcast(DT, formula = ID ~ Age, value.var = "Result", fun.aggregate =
function(x){paste(x,collapse = "_")})
ID    20    21    22
1:  a 1_4_7
2:  b       2_5_8
3:  c             3_6_9
```

También podemos pasar una lista a `fun.aggregate` para usar múltiples funciones

```
dcast(DT, formula = ID ~ Age, value.var = "Result", fun.aggregate = list(sum,length))
   ID Result_sum_20 Result_sum_21 Result_sum_22 Result_length_20 Result_length_21
Result_length_22
1:  a            12             0             0                3                0
0
2:  b             0            15             0                0                3
0
3:  c             0             0            18                0                0
3
```

Si pasamos más de una función y más de un valor, podemos calcular todas las combinaciones pasando un vector de value.vars

```
dcast(DT, formula = ID ~ Age, value.var = c("Result","Test"), fun.aggregate =
list(function(x){paste0(x,collapse = "_")},length))
   ID Result_function_20 Result_function_21 Result_function_22 Test_function_20
Test_function_21 Test_function_22 Result_length_20 Result_length_21
1:  a              1_4_7                                        OB_A_OB_B_OB_C
3                0
2:  b                                  2_5_8
OB_A_OB_B_OB_C                                      0                3
3:  c                                                         3_6_9
OB_A_OB_B_OB_C                0                0
   Result_length_22 Test_length_20 Test_length_21 Test_length_22
1:                0              3              0              0
2:                0              0              3              0
3:                3              0              0              3
```

donde cada par se calcula en el orden `value1_formula1, value1_formula2, ... , valueN_formula(N-1), valueN_formulaN`.

Alternativamente, podemos evaluar nuestros valores y funciones uno a uno al pasar 'value.var' como una lista:

```
dcast(DT, formula = ID ~ Age, value.var = list("Result","Test"), fun.aggregate =
list(function(x){paste0(x,collapse = "_")},length))
   ID Result_function_20 Result_function_21 Result_function_22 Test_length_20 Test_length_21
```

```
Test_length_22
1:  a              1_4_7                                    3             0
0
2:  b                                 2_5_8                 0             3
0
3:  c                                              3_6_9    0             0
3
```

Nombrando columnas en el resultado

De forma predeterminada, los componentes de nombre de columna están separados por un guión bajo _ . Esto puede ser anulado manualmente usando el argumento `sep=` :

```
dcast(DT, formula = Test ~ ID + Age, value.var = "Result")
Test a_20 b_21 c_22
1: OB_A    1    2    3
2: OB_B    4    5    6
3: OB_C    7    8    9

dcast(DT, formula = Test ~ ID + Age, value.var = "Result", sep = ",")
   Test a,20 b,21 c,22
1: OB_A    1    2    3
2: OB_B    4    5    6
3: OB_C    7    8    9
```

Esto `fun.aggregate` cualquier `fun.aggregate` o `value.var` que usemos:

```
dcast(DT, formula = Test ~ ID + Age, value.var = "Result", fun.aggregate = c(sum,length), sep
= ",")
   Test Result,sum,a,20 Result,sum,b,21 Result,sum,c,22 Result,length,a,20 Result,length,b,21
Result,length,c,22
1: OB_A               1               2               3                  1                  1
1
2: OB_B               4               5               6                  1                  1
1
3: OB_C               7               8               9                  1                  1
1
```

Apilando múltiples tablas usando rbindlist

Un refrán común en R va en esta línea:

> No debe tener un montón de tablas relacionadas con nombres como `DT1` , `DT2` , ..., `DT11` . Literalmente, leer y asignar objetos a los objetos es complicado. La solución es una lista de tablas de datos!

Una lista así parece

```
set.seed(1)
DT_list = lapply(setNames(1:3, paste0("D", 1:3)), function(i)
  data.table(id = 1:2, v = sample(letters, 2)))
```

```
$D1
   id v
1:  1 g
2:  2 j

$D2
   id v
1:  1 o
2:  2 w

$D3
   id v
1:  1 f
2:  2 w
```

Otra perspectiva es que debe almacenar estas tablas juntas *como una tabla* , apilándolas. Esto es sencillo de hacer usando `rbindlist` :

```
DT = rbindlist(DT_list, id="src")

   src id v
1:  D1  1 g
2:  D1  2 j
3:  D2  1 o
4:  D2  2 w
5:  D3  1 f
6:  D3  2 w
```

Este formato tiene mucho más sentido con la sintaxis data.table, donde las operaciones "por grupo" son comunes y directas.

Para una mirada más profunda, la respuesta de Gregor podría ser un buen lugar para comenzar. También echa un vistazo a `?rbindlist` , por supuesto. Hay un ejemplo separado que cubre la lectura en un montón de tablas de CSV y luego las apila .

Lea Remodelación, apilado y fraccionamiento. en línea: https://riptutorial.com/es/data-table/topic/4117/remodelacion--apilado-y-fraccionamiento-

Capítulo 8: Se une y se fusiona

Introducción

Una combinación combina dos tablas que contienen columnas relacionadas. El término cubre una amplia gama de operaciones, esencialmente todo excepto agregar las dos tablas . "Fusionar" es un sinónimo. Escriba `?`[.data.table`` para los documentos oficiales.

Sintaxis

- x [i, on, j]
  # join: data.table x & data.table o lista i
- x [! i, on, j]
  # anti-join

Observaciones

Trabajar con tablas con llave

Si `x` & `i` tiene una clave o `x` se teclea para coincidir con las primeras columnas de `i` , entonces el `on` puede omitirse como `x[i]` .

Desambiguación de nombres de columnas en común.

En `j` de `x[i, on, j]` , las columnas de `i` pueden referirse con `i.*` Prefijos.

Agrupación en subconjuntos

En `j` de `x[i, on, j, by=.EACHI]` , `j` se calcula para cada fila de `i` .

Este es el único valor de `by` valor de usar. Para cualquier otro valor, las columnas de `i` no están disponibles.

Examples

Actualizar valores en una unión

Cuando los datos están "ordenados" , a menudo se organizan en varias tablas. Para combinar los

datos para el análisis, necesitamos "actualizar" una tabla con valores de otra.

Por ejemplo, podríamos tener datos de ventas para actuaciones, donde los atributos del ejecutante (su presupuesto) y de la ubicación (su población) se almacenan en tablas separadas:

```
set.seed(1)
mainDT = data.table(
  p_id = rep(LETTERS[1:2], c(2,4)),
  geo_id = sample(rep(state.abb[c(1,25,50)], 3:1)),
  sales = sample(100, 6)
)
pDT   = data.table(id = LETTERS[1:2], budget = c(60, 75))
geoDT = data.table(id = state.abb[c(1,50)], pop = c(100, 200))

mainDT # sales data
#    p_id geo_id sales
# 1:    A     AL    95
# 2:    A     WY    66
# 3:    B     AL    62
# 4:    B     MO     6
# 5:    B     AL    20
# 6:    B     MO    17


pDT # performer attributes
#    id budget
# 1:  A     60
# 2:  B     75

geoDT # location attributes
#    id pop
# 1: AL 100
# 2: WY 200
```

Cuando estemos listos para hacer un análisis, necesitamos tomar variables de estas otras tablas:

```
DT = copy(mainDT)

DT[pDT, on=.(p_id = id), budget := i.budget]
DT[geoDT, on=.(geo_id = id), pop := i.pop]

#    p_id geo_id sales budget pop
# 1:    A     AL    95     60 100
# 2:    A     WY    66     60 200
# 3:    B     AL    62     75 100
# 4:    B     MO     6     75  NA
# 5:    B     AL    20     75 100
# 6:    B     MO    17     75  NA
```

Se toma una `copy` para evitar contaminar los datos sin procesar, pero podríamos trabajar directamente en `mainDT` .

Ventajas de usar tablas separadas.

Las ventajas de esta estructura se tratan en el documento sobre datos ordenados, pero en este

contexto:

1. *Rastreo de datos faltantes.* Sólo las filas que coinciden en la combinación reciben una asignación. No tenemos datos para `geo_id == "MO"` arriba, por lo que sus variables son `NA` en nuestra tabla final. Si vemos datos faltantes como este de forma inesperada, podemos rastrearlos hasta la observación faltante en la tabla `geoDT` e investigar desde allí si tenemos un problema de datos que se pueda solucionar.

2. *Comprensibilidad* Al construir nuestro modelo estadístico, podría ser importante tener en cuenta que el `budget` es constante para cada actor. En general, la comprensión de la estructura de los datos paga dividendos.

3. *Tamaño de la memoria.* Puede haber un gran número de atributos de ubicación y de ejecutante que no terminan en el modelo estadístico. De esta manera, no necesitamos incluirlos en la tabla (posiblemente masiva) utilizada para el análisis.

Columnas determinantes programáticamente

Si hay muchas columnas en `pDT` , pero solo queremos seleccionar algunas, podemos usar

```
p_cols = "budget"
DT[pDT, on=.(p_id = id), (p_cols) := mget(sprintf("i.%s", p_cols))]
```

Los paréntesis alrededor de `(p_cols) :=` son esenciales, como se indica en el documento al crear columnas .

Equi-unirse

```
# example data
a = data.table(id = c(1L, 1L, 2L, 3L, NA_integer_), x = 11:15)
#    id  x
# 1:  1 11
# 2:  1 12
# 3:  2 13
# 4:  3 14
# 5: NA 15

b = data.table(id = 1:2, y = -(1:2))
#    id  y
# 1:  1 -1
# 2:  2 -2
```

Intuición

Piense en `x[i]` como seleccionar un subconjunto de `x` para cada fila de `i` . Esta sintaxis refleja el subconjunto de matriz en la base R y es consistente con el primer argumento que significa "dónde", en `DT[where, select|update|do, by]` .

Uno podría preguntarse por qué vale la pena aprender esta nueva sintaxis, ya que `merge(x,i)` aún funciona con data.tables. La respuesta corta es que normalmente queremos fusionarnos y luego hacer algo más. La sintaxis `x[i]` captura de manera concisa este patrón de uso y también permite un cálculo más eficiente. Para una explicación más detallada, lea las preguntas frecuentes 1.12 y 2.14 .

Manejo de filas de múltiples partidos

De forma predeterminada, se devuelve cada fila de `a` coincidencia con cada fila de `b` :

```
a[b, on="id"]
#    id  x  y
# 1:  1 11 -1
# 2:  1 12 -1
# 3:  2 13 -2
```

Esto puede ser ajustado con `mult` :

```
a[b, on="id", mult="first"]
#    id  x  y
# 1:  1 11 -1
# 2:  2 13 -2
```

Manejando filas incomparables

De forma predeterminada, las filas no coincidentes de `a` muestran en el resultado:

```
b[a, on="id"]
#    id  y  x
# 1:  1 -1 11
# 2:  1 -1 12
# 3:  2 -2 13
# 4:  3 NA 14
# 5: NA NA 15
```

Para ocultar estos, use `nomatch` :

```
b[a, on="id", nomatch=0]
#    id  y  x
# 1:  1 -1 11
# 2:  1 -1 12
# 3:  2 -2 13
```

Tenga en cuenta que `x[i]` intentará hacer coincidir las NA en `i` .

Contando partidos devueltos

Para contar el número de coincidencias para cada fila de `i` , use `.N` y `by=.EACHI` .

```
b[a, on="id", .N, by=.EACHI]
#    id N
# 1:  1 1
# 2:  1 1
# 3:  2 1
# 4:  3 0
# 5: NA 0
```

Lea Se une y se fusiona en línea: https://riptutorial.com/es/data-table/topic/4976/se-une-y-se-fusiona

Capítulo 9: Subconjunto de filas por grupo

Observaciones

Un recordatorio: la sintaxis de `DT[where, select|update|do, by]` se utiliza para trabajar con columnas de una tabla de datos.

- La parte "donde" es el argumento `i`
- La parte "seleccionar | actualizar | hacer" es el argumento `j`

Estos dos argumentos generalmente se pasan por posición en lugar de por nombre.

Examples

Seleccionando filas dentro de cada grupo

```
# example data
DT <- data.table(Titanic)
```

Supongamos que, para cada sexo, queremos las filas con los números de supervivencia más altos:

```
DT[Survived == "Yes", .SD[ N == max(N) ], by=Sex]

#    Class    Sex   Age Survived   N
# 1:  Crew   Male Adult      Yes 192
# 2:   1st Female Adult      Yes 140
```

`.SD` es el subconjunto de datos asociados con cada `Sex` ; y lo estamos subdividiendo aún más, a las filas que cumplen con nuestra condición. Si la velocidad es importante, en su lugar, utilice un enfoque sugerido por eddi en SO :

```
DT[ DT[Survived == "Yes", .I[ N == max(N) ], by=Sex]$V1 ]

#    Class    Sex   Age Survived   N
# 1:  Crew   Male Adult      Yes 192
# 2:   1st Female Adult      Yes 140
```

Escollos

En la última línea de código, `.I` refiere a los números de fila de la tabla de datos completa. Sin embargo, esto no es cierto cuando no hay `by` :

```
DT[ Survived == "Yes", .I]

# 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
```

```
DT[ Survived == "Yes", .I, by=Sex]$I

# 17 18 19 20 25 26 27 28 21 22 23 24 29 30 31 32
```

Seleccionando grupos

```
# example data
DT = data.table(Titanic)
```

Supongamos que solo queremos ver la segunda clase:

```
DT[ Class == "2nd" ]

#    Class    Sex   Age Survived   N
# 1:   2nd   Male Child       No   0
# 2:   2nd Female Child       No   0
# 3:   2nd   Male Adult       No 154
# 4:   2nd Female Adult       No  13
# 5:   2nd   Male Child      Yes  11
# 6:   2nd Female Child      Yes  13
# 7:   2nd   Male Adult      Yes  14
# 8:   2nd Female Adult      Yes  80
```

Aquí, simplemente subcontratamos los datos usando `i` , la cláusula "dónde".

Selección de grupos por condición

```
# example data
DT = data.table(Titanic)
```

Supongamos que queremos ver cada clase solo si una mayoría sobrevivió:

```
DT[, if (sum(N[Survived=="Yes"]) > sum(N[Survived=="No"]) ) .SD, by=Class]

#    Class    Sex   Age Survived   N
# 1:   1st   Male Child       No   0
# 2:   1st Female Child       No   0
# 3:   1st   Male Adult       No 118
# 4:   1st Female Adult       No   4
# 5:   1st   Male Child      Yes   5
# 6:   1st Female Child      Yes   1
# 7:   1st   Male Adult      Yes  57
# 8:   1st Female Adult      Yes 140
```

Aquí, devolvemos el subconjunto de datos `.SD` solo si nuestra condición se cumple. Una alternativa es

```
DT[, .SD[ sum(N[Survived=="Yes"]) > sum(N[Survived=="No"]) ) ], by=Class]
```

pero esto a veces ha demostrado ser más lento.

Lea Subconjunto de filas por grupo en línea: https://riptutorial.com/es/data-table/topic/3784/subconjunto-de-filas-por-grupo

Capítulo 10: Usando .SD y .SDcols para el subconjunto de datos

Introducción

El símbolo especial `.SD` está disponible en `j` de `DT[i,j,by]` , la captura de la **S** ubset de ata **D** para cada `by` grupo sobrevivir el filtro, `i` . `.SDcols` es un ayudante. Escriba `?`special-symbols`` para los documentos oficiales.

Observaciones

Un recordatorio: la sintaxis de `DT[where, select|update|do, by]` se utiliza para trabajar con columnas de una tabla de datos.

- La parte "donde" es el argumento `i`
- La parte "seleccionar | actualizar | hacer" es el argumento `j`

Estos dos argumentos generalmente se pasan por posición en lugar de por nombre.

Examples

Usando .SD y .SDcols

.DAKOTA DEL SUR

`.SD` refiere al subconjunto de la `data.table` de `data.table` para cada grupo, excluyendo todas las columnas utilizadas `by` .

`.SD` junto con `lapply` se puede usar para aplicar cualquier función a varias columnas por grupo en una `data.table`

Continuaremos usando el mismo conjunto de datos `mtcars` , `mtcars` :

```
mtcars = data.table(mtcars) # Let's not include rownames to keep things simpler
```

Media de todas las columnas en el conjunto de datos por *número de cilindros* , `cyl` :

```
mtcars[ , lapply(.SD, mean), by = cyl]

#   cyl      mpg     disp        hp     drat       wt     qsec        vs        am     gear
carb
#1:   6 19.74286 183.3143 122.28571 3.585714 3.117143 17.97714 0.5714286 0.4285714 3.857143
3.428571
#2:   4 26.66364 105.1364  82.63636 4.070909 2.285727 19.13727 0.9090909 0.7272727 4.090909
```

```
1.545455
#3:   8 15.10000 353.1000 209.21429 3.229286 3.999214 16.77214 0.0000000 0.1428571 3.285714
3.500000
```

Aparte de `cyl` , hay otras columnas categóricas en el conjunto de datos como `vs` , `am` , `gear` y `carb` . Realmente no tiene sentido tomar la `mean` de estas columnas. Así que vamos a excluir estas columnas. Aquí es donde `.SDcols` entra en escena.

.SDcols

`.SDcols` especifica las columnas de la `data.table` que se incluyen en `.SD` .

La media de todas las columnas (columnas continuas) en el conjunto de datos por *número de engranajes* `gear` , y el *número de cilindros,* `cyl` , dispuestos por `gear` y `cyl` :

```
# All the continuous variables in the dataset
cols_chosen <- c("mpg", "disp", "hp", "drat", "wt", "qsec")

mtcars[order(gear, cyl), lapply(.SD, mean), by = .(gear, cyl), .SDcols = cols_chosen]

#   gear cyl    mpg     disp       hp     drat       wt    qsec
#1:    3   4 21.500 120.1000  97.0000 3.700000 2.465000 20.0100
#2:    3   6 19.750 241.5000 107.5000 2.920000 3.337500 19.8300
#3:    3   8 15.050 357.6167 194.1667 3.120833 4.104083 17.1425
#4:    4   4 26.925 102.6250  76.0000 4.110000 2.378125 19.6125
#5:    4   6 19.750 163.8000 116.5000 3.910000 3.093750 17.6700
#6:    5   4 28.200 107.7000 102.0000 4.100000 1.826500 16.8000
#7:    5   6 19.700 145.0000 175.0000 3.620000 2.770000 15.5000
#8:    5   8 15.400 326.0000 299.5000 3.880000 3.370000 14.5500
```

Tal vez no queremos calcular la `mean` por grupos. Para calcular la media de todos los autos en el conjunto de datos, no especificamos la variable `by` .

```
mtcars[ , lapply(.SD, mean), .SDcols = cols_chosen]

#        mpg     disp       hp     drat      wt     qsec
#1: 20.09062 230.7219 146.6875 3.596563 3.21725 17.84875
```

Nota: No es necesario definir `cols_chosen` antemano. `.SDcols` puede tomar directamente nombres de columna

Lea Usando .SD y .SDcols para el subconjunto de datos en línea: https://riptutorial.com/es/data-table/topic/3787/usando--sd-y--sdcols-para-el-subconjunto-de-datos

Capítulo 11: Usando columnas de lista para almacenar datos

Introducción

Data.table admite vectores de columna que pertenecen a la clase de `list` de R.

Observaciones

En caso de que parezca extraño que estemos hablando de listas sin usar esa palabra en el código, tenga en cuenta que `.()` Es un alias para la `list()` cuando se usa dentro de una llamada `DT[...]`.

Examples

Leyendo en muchos archivos relacionados

Supongamos que queremos leer y apilar un montón de archivos con formato similar. La solución rápida es:

```
rbindlist(lapply(list.files(patt="csv$"), fread), id=TRUE)
```

Puede que no estemos satisfechos con esto por un par de razones:

- Es posible que se `fread` errores al leer con `fread` o al apilar con `rbindlist` debido a un formato de datos inconsistente o `rbindlist`.
- Es posible que queramos realizar un seguimiento de los metadatos de cada archivo, tomados del nombre del archivo o quizás de algunas filas de encabezado dentro de los archivos (no del todo tabulares).

Una forma de manejar esto es hacer una "tabla de archivos" y almacenar el contenido de cada archivo como una entrada de columna de lista en la fila asociada a él.

Ejemplo de datos

Antes de crear los datos de ejemplo a continuación, asegúrese de estar en una carpeta vacía en la que pueda escribir. Ejecute `getwd()` y lea `?setwd` si necesita cambiar las carpetas.

```
# example data
set.seed(1)
for (i in 1:3)
  fwrite(data.table(id = 1:2, v = sample(letters, 2)), file = sprintf("file201%s.csv", i))
```

Identificar archivos y metadatos de archivos.

Esta parte es bastante sencilla:

```
# First, identify the files you want:
fileDT = data.table(fn = list.files(pattern="csv$"))

# Next, optionally parse the names for metadata using regex:
fileDT[, year := type.convert(sub(".*([0-9]{4}).*", "\\1", fn))]

# Finally construct a string file-ID column:
fileDT[, id := as.character(.I)]

#              fn year id
# 1: file2011.csv 2011  1
# 2: file2012.csv 2012  2
# 3: file2013.csv 2013  3
```

Leer en archivos

Lea en los archivos como una columna de la lista:

```
fileDT[, contents := .(lapply(fn, fread))]

#              fn year id     contents
# 1: file2011.csv 2011  1 <data.table>
# 2: file2012.csv 2012  2 <data.table>
# 3: file2013.csv 2013  3 <data.table>
```

Si hay un obstáculo en la lectura de uno de los archivos o si necesita cambiar los argumentos a `fread` según los atributos del archivo, este paso puede extenderse fácilmente, con el siguiente aspecto:

```
fileDT[, contents := {
  cat(fn, "\n")

  dat = if (year %in% 2011:2012){
    fread(fn, some_args)
  } else {
    fread(fn)
  }

  .(.(dat))
}, by=fn]
```

Para obtener detalles sobre las opciones de lectura en CSV y archivos similares, consulte `?fread` .

Apilar datos

A partir de aquí, queremos apilar los datos:

```
fileDT[, rbindlist(setNames(contents, id), idcol="file_id")]

#    file_id id v
# 1:       1  1 g
# 2:       1  2 j
# 3:       2  1 o
# 4:       2  2 w
# 5:       3  1 f
# 6:       3  2 w
```

Si ocurre algún problema en el apilamiento (como nombres de columnas o clases que no coinciden), podemos regresar a las tablas individuales en `fileDT` para inspeccionar dónde se originó el problema. Por ejemplo,

```
fileDT[id == "2", contents[[1]]]
#    id v
# 1:  1 o
# 2:  2 w
```

Extensiones

Si los archivos no están en su directorio de trabajo actual, use

```
my_dir = "whatever"
fileDT = data.table(fn = list.files(my_dir, pattern="*.csv"))

# and when reading
fileDT[, contents := .(lapply(fn, function(n) fread(file.path(my_dir, n))))]
```

Lea Usando columnas de lista para almacenar datos en línea: https://riptutorial.com/es/data-table/topic/4456/usando-columnas-de-lista-para-almacenar-datos

Capítulo 12: Uso de claves e índices.

Introducción

La clave y los índices de una tabla de datos permiten que ciertos cálculos se ejecuten más rápido, principalmente relacionados con uniones y subconjuntos. La clave describe el orden de clasificación actual de la tabla; mientras que cada índice almacena información sobre el orden de la tabla con respecto a una secuencia de columnas. Consulte la sección "Comentarios" a continuación para obtener enlaces a las viñetas oficiales sobre el tema.

Observaciones

Las viñetas oficiales son la mejor introducción a este tema:

- "Claves y subconjunto basado en búsqueda binaria rápida"
- "Índices secundarios e indexación automática"

Teclas vs índices

Una tabla de datos puede ser "codificada" por una secuencia de columnas, indicando a las funciones interesadas que los datos están ordenados por esas columnas. Para obtener o configurar la clave, use las funciones documentadas en `?key` .

De manera similar, las funciones pueden aprovechar los "índices" de una tabla de datos. Cada índice, y una tabla puede tener más de uno, almacena información sobre el orden de los datos con respecto a una secuencia de columnas. Al igual que una clave, un índice puede acelerar ciertas tareas. Para obtener o establecer índices, utilice las funciones documentadas en los `?indices`

Los índices también se pueden configurar automáticamente (actualmente solo para una columna a la vez). Consulte `?datatable.optimize` para obtener detalles sobre cómo funciona esto y cómo desactivarlo si es necesario.

Verificación y actualización.

Los valores que faltan se permiten en una columna de clave.

Las claves y los índices se almacenan como atributos y pueden, por accidente, no corresponder al orden real de los datos en la tabla. Muchas funciones verifican la validez de la clave o el índice antes de usarla, pero vale la pena tenerlas en cuenta.

Las claves y los índices se eliminan después de las actualizaciones donde no es obvio que se conserva el orden de clasificación. Por ejemplo, a partir de `DT = data.table(a=c(1,2,4), key="a")` , si actualizamos como `DT[2, a := 3]` , la clave está rota.

Examples

Mejora del rendimiento para seleccionar subconjuntos

```
# example data
set.seed(1)
n  = 1e7
ng = 1e4
DT = data.table(
    g1  = sample(ng, n, replace=TRUE),
    g2  = sample(ng, n, replace=TRUE),
    v   = rnorm(n)
)
```

Coincidencia en una columna

Después de la primera ejecución de una operación de subconjunto con `==` o `%in%` ...

```
system.time(
    DT[ g1 %in% 1:100]
)
#    user  system elapsed
#    0.12    0.03    0.16
```

Se ha creado automáticamente un índice para `g1` . Las siguientes operaciones de subconjunto se ejecutan casi instantáneamente:

```
system.time(
    DT[ g1 %in% 1:100]
)
#    user  system elapsed
#       0       0       0
```

Para controlar cuándo se crea o utiliza un índice, agregue la opción `verbose=TRUE` o cambie las `options(datatable.verbose=TRUE)` configuración global `options(datatable.verbose=TRUE)` .

Coincidencia en múltiples columnas

Actualmente, la coincidencia en dos columnas no crea automáticamente un índice:

```
system.time(
    DT[ g1 %in% 1:100 & g2 %in% 1:100]
)
#    user  system elapsed
#    0.57    0.00    0.57
```

Vuelva a ejecutar esto y seguirá siendo lento. Incluso si agregamos manualmente el índice con `setindex(DT, g1, g2)` , seguirá siendo lento porque esta consulta aún no está optimizada por el

paquete.

Afortunadamente, si podemos enumerar las combinaciones de valores que queremos buscar y hay un índice disponible, podemos unirnos rápidamente:

```
system.time(
    DT[ CJ(g1 = 1:100, g2 = 1:100, unique=TRUE), on=.(g1, g2), nomatch=0]
)
#    user  system elapsed
#    0.53    0.00    0.54
setindex(DT, g1, g2)
system.time(
    DT[ CJ(g1 = 1:100, g2 = 1:100, unique=TRUE), on=.(g1, g2), nomatch=0]
)
#    user  system elapsed
#       0       0       0
```

Con `CJ` , es importante tener cuidado con la cantidad de combinaciones que se vuelven demasiado grandes.

Lea Uso de claves e índices. en línea: https://riptutorial.com/es/data-table/topic/4977/uso-de-claves-e-indices-

Creditos

S. No	Capítulos	Contributors
1	Primeros pasos con data.table	Community, Frank, micstr
2	¿Por qué mi antiguo código no funciona?	Frank, micstr
3	Añadiendo y modificando columnas.	eddi, Frank, jangorecki, micstr
4	Creando una tabla de datos	Chris, Frank
5	Datos de limpieza	Frank
6	Informes estadísticos de resumen.	Frank
7	Remodelación, apilado y fraccionamiento.	Chris, David Arenburg, Frank, SymbolixAU
8	Se une y se fusiona	Chris, Frank
9	Subconjunto de filas por grupo	Frank, micstr
10	Usando .SD y .SDcols para el subconjunto de datos	Frank
11	Usando columnas de lista para almacenar datos	Frank
12	Uso de claves e índices.	Frank