Filtrar y resumir datos

Después de cargar datos desde archivos planos o bases de datos, o directamente desde la web a través de algunas APIs, a menudo tenemos que agregar, transformar o filtrar el conjunto de datos original antes de que pueda tener lugar el análisis de datos real (o trabajar con los algoritmos estudiados en un curso de minería de datos). Utilice la versión 4.1.3.

En esta parte del curso, nos centraremos en cómo:

- Filtrar filas y columnas en marcos de datos (data frames)
- Resumir y agregar datos
- Mejorar el desempeño de tales tareas con los paquetes dplyr y data.table además de los métodos base en R

Elimina datos innecesarios

Aunque no cargar los datos innecesarios es la solución óptima, a menudo tenemos que filtrar el conjunto de datos original dentro de R. Esto se puede hacer con las herramientas y funciones tradicionales de base en R, como subconjunto, usando which y el operador [o [[(ver el siguiente código), o por ejemplo con el enfoque similar a SQL del paquete sqldf:

> install.packages("sqldf")

Version de consulta SQL

```
> library(sqldf)
Loading required package: gsubfn
Loading required package: proto
Loading required package: RSQLite
> sqldf("SELECT * FROM mtcars WHERE am=1 AND vs=1")
   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
1 22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
2 32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
3 30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
4 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
5 27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
6 30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
7 21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
>
```

Estoy seguro de que todos los lectores que tienen una experiencia ‘decente’ en SQL y que recién se están poniendo en contacto con R aprecian esta forma alternativa de filtrar datos, pero personalmente prefiero la siguiente versión R bastante similar, nativa y mucho más concisa:

```
> subset(mtcars, am == 1 & vs == 1)
                mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
>
```

Version nativa de R

Ten en cuenta la ligera diferencia en los resultados. Esto se atribuye al hecho de que el argumento row.names de sqldf es FALSE por defecto, lo que, por supuesto, se puede anular para obtener exactamente los mismos resultados (el más concatena instrucciones y es automático en R():

```
> identical(
+ sqldf("SELECT * FROM mtcars WHERE am=1 AND vs=1",
+ row.names = TRUE),
+ subset(mtcars, am == 1 & vs == 1)
+ )
[1] TRUE
>
```

Comparación de igualdad (son lo mismo)

Estos ejemplos se centraron en cómo eliminar filas de data.frame, pero ¿qué pasa si también queremos eliminar algunas columnas?

El enfoque de SQL es realmente sencillo; simplemente especifica las columnas requeridas en lugar de * en la instrucción SELECT. Por otro lado, el subconjunto también admite este enfoque mediante el argumento select, que puede tomar vectores o una expresión R que describa, por ejemplo, un rango de columnas:

```
> subset(mtcars, am == 1 & vs == 1, select = hp:wt)
                hp drat    wt
Datsun 710      93 3.85 2.320
Fiat 128        66 4.08 2.200
Honda Civic     52 4.93 1.615
Toyota Corolla  65 4.22 1.835
Fiat X1-9       66 4.08 1.935
Lotus Europa   113 3.77 1.513
Volvo 142E     109 4.11 2.780
>
```

Consulta para seleccionar algunas columnas

Nota: Pasa los nombres de las columnas sin comillas como un vector a través de la función c para seleccionar una lista arbitraria de columnas en el orden dado, o excluye las columnas especificadas usando el operador -, por ejemplo, subset (mtcars, select = -c (hp, wt)).

Llevemos esto al siguiente paso y veamos cómo podemos aplicar los filtros antes mencionados en algunos conjuntos de datos más grandes, cuando enfrentamos algunos problemas de rendimiento con las funciones base.

Elimina datos innecesarios de una manera eficiente

R funciona mejor con conjuntos de datos que pueden caber en la memoria física real y algunos paquetes de R proporcionan un acceso extremadamente rápido a esta cantidad de datos.

Nota: Algunos puntos de referencia proporcionan ejemplos de la vida real de funciones de R de resumen más eficientes que las que ofrecen las bases de datos de código abierto actuales (por ejemplo, MySQL, PostgreSQL e Impala) y comerciales (como HP Vertica).
Veamos cómo funcionan los ejemplos anteriores en este conjunto de datos de un cuarto de millón de filas:

> install.packages("hflights")

```
> library(hflights)
> system.time(sqldf("SELECT * FROM hflights WHERE Dest == 'BNA'",
+ row.names = TRUE))
   user  system elapsed
   1.81    0.07    2.23
> system.time(subset(hflights, Dest == 'BNA'))
   user  system elapsed
   0.03    0.00    0.04
>
```

Prueba de eficiencia entre dos consultas

La función base::subset parece funcionar bastante bien, pero ¿podemos hacerlo más rápido? Bueno, la segunda generación del paquete plyr, llamado dplyr (los detalles relevantes se discuten en la sección de funciones auxiliares de alto rendimiento en las notas de los desarrolladores de R), proporciona implementaciones C++ extremadamente rápidas de los métodos de manipulación de bases de datos más comunes de una manera bastante intuitiva:

> str

```
> library(dplyr)

Attaching package: 'dplyr'

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

    filter, lag

The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union

> system.time(filter(hflights, Dest == 'BNA'))
   user  system elapsed
   0.03    0.00    0.03
>
```

Además, podemos extender esta solución eliminando algunas columnas del conjunto de datos como lo hicimos antes con el subconjunto, aunque ahora llamamos a la función de selección en lugar de pasar un argumento con el mismo nombre:

```
> str(select(filter(hflights, Dest == 'BNA'), DepTime:ArrTime))
'data.frame':   3481 obs. of  2 variables:
 $ DepTime: int  1419 1232 1813 900 716 1357 2000 1142 811 1341 ...
 $ ArrTime: int  1553 1402 1948 1032 845 1529 2132 1317 945 1519 ...
>
```

Consulta eficiente con select y filtrado

Por lo tanto, es como llamar a la función de filtro en lugar de subconjunto, ¡y obtenemos los resultados más rápido que un abrir y cerrar de ojos! El paquete dplyr puede funcionar con objetos data.frame u data.table tradicionales, o puede interactuar directamente con los motores de base de datos más utilizados. Ten en cuenta que los nombres de las filas no se conservan en dplyr, por lo que si los necesitas, vale la pena copiar los nombres en variables explícitas antes de pasarlas a dplyr o directamente a data.table de la siguiente manera:

```
> mtcars$rownames <- rownames(mtcars)
> select(filter(mtcars, hp > 300), c(rownames, hp))
                  rownames  hp
Maserati Bora Maserati Bora 335
>
```

Guardado de nombres dado que dplyr no carga nombres

Elimina datos innecesarios de otra manera eficiente

Veamos un ejemplo rápido de la solución data.table por sí sola, sin dplyr.

Nota. El paquete data.table proporciona una manera extremadamente eficiente de manejar conjuntos de datos más grandes en una estructura de datos en memoria autoindexada y basada en columnas, con compatibilidad con los métodos tradicionales data.frame.

Después de cargar el paquete, tenemos que transformar el data.frame tradicional de hflights en data.table. Luego, creamos una nueva columna, llamada nombres de fila, a la que asignamos los nombres de fila del conjunto de datos original con la ayuda del: = operador de asignación específico de data.table:

```
> install.packages("data.table")
> library(data.table)
data.table 1.13.0 using 1 threads (see ?getDTthreads).  Latest news: r-datatable.com

Attaching package: 'data.table'

The following objects are masked from 'package:dplyr':

    between, first, last

> hflights_dt <- data.table(hflights)
> hflights_dt[, rownames := rownames(hflights)]
> system.time(hflights_dt[Dest == 'BNA'])
   user  system elapsed
   0.02    0.00    0.02
>
```

Pasa a data.table hflights y asigna en la columna rownames los nombres de cada registro

Bueno, lleva algo de tiempo acostumbrarse a la sintaxis personalizada de data.table e incluso puede parecer un poco extraño para el usuario de R tradicional a primera vista, pero definitivamente vale la pena dominarlo a largo plazo. Obtienes un gran rendimiento y la sintaxis resulta ser natural y flexible después de la curva de aprendizaje relativamente empinada de los primeros ejemplos.

De hecho, la sintaxis de data.table es bastante similar a SQL:

DT[i, j, ... , drop = TRUE]

Esto se podría describir con comandos SQL de la siguiente manera:

DT[where, select | update, group by][having][order by][ ]...[ ]

Por lo tanto, [.data.table (que significa el operador [operador aplicado a un objeto data.table) tiene algunos argumentos diferentes en comparación con la sintaxis tradicional [.data.frame], como ya has visto en el ejemplo anterior.

Nota. Ahora, no estamos tratando con el operador de asignación en detalle, ya que este ejemplo podría ser demasiado complejo para una parte tan introductoria del curso, y probablemente estemos saliendo de nuestra zona de confort.

Parece que el primer argumento (i) del operador [.data.table representa el filtrado, o en otras palabras, la declaración WHERE en el lenguaje SQL, mientras que [.data.frame espera índices que especifiquen qué filas mantener del original conjunto de datos. La diferencia real entre los dos argumentos es que el primero puede tomar cualquier expresión R, mientras que el último método tradicional espera principalmente números enteros o valores lógicos.

De todos modos, filtrar es tan fácil como pasar una expresión R al argumento i del operador [específico de data.table. Además, veamos cómo podemos seleccionar las columnas en la sintaxis data.table, lo que debería hacerse en el segundo argumento (j) de la llamada sobre la base de la sintaxis general data.table mencionada anteriormente:

Consulta SQL realizada con data.table

```
> str(hflights_dt[Dest == 'BNA', list(DepTime, ArrTime)])
Classes 'data.table' and 'data.frame':  3481 obs. of  2 variables:
 $ DepTime: int  1419 1232 1813 900 716 1357 2000 1142 811 1341 ...
 $ ArrTime: int  1553 1402 1948 1032 845 1529 2132 1317 945 1519 ...
 - attr(*, ".internal.selfref")=<externalptr>
>
```

Bien, ahora tenemos las dos columnas esperadas con las 3481 observaciones. Ten en cuenta que list se usó para definir las columnas requeridas para mantener, aunque el uso de c (una función de la base R para concatenar elementos vectoriales) se usa más tradicionalmente

con [.data.frame. Esto último también es posible con [.data.table, pero luego, debes pasar los nombres de las variables como un vector de caracteres y establecer con FALSE:

```
> hflights_dt[Dest == 'BNA', c('DepTime', 'ArrTime'), with = FALSE]
      DepTime ArrTime
   1:    1419    1553
   2:    1232    1402
   3:    1813    1948
   4:     900    1032
   5:     716     845
  ---
3477:    1055    1240
3478:    1834    2006
3479:    1251    1422
3480:     656     836
3481:    1047    1219
>
```

Paso de nombre de variables como vector de R, se debe establecer con FALSE

En lugar de list, puedes utilizar un punto como nombre de función en el estilo del paquete plyr; por ejemplo: hflights_dt [,. (DepTime, ArrTime)].

Ahora que estamos más o menos familiarizados con nuestras opciones para filtrar datos dentro de una sesión de R en vivo, y conocemos la sintaxis general de los paquetes dplyr y data.table, veamos cómo se pueden usar para agregar y resumir datos en acción.

Agregación

La forma más sencilla de resumir datos es llamar a la función *aggregate* desde el paquete de estadísticas (*stats*), que hace exactamente lo que estamos buscando: dividir los datos en subconjuntos por una variable de agrupación y luego calcular las estadísticas de resumen para ellos por separado.

aggwit

La forma más básica de llamar a la función aggregate es pasar el vector numérico que se agregará y una variable de factor para definir las divisiones de la función pasada en el argumento FUN que se aplicará. Ahora, veamos la proporción promedio de vuelos desviados cada día de la semana:

```
> aggregate(hflights$Diverted, by = list(hflights$DayOfWeek),
+ FUN = mean)
  Group.1           x
1       1 0.002997672
2       2 0.002559323
3       3 0.003226211
4       4 0.003065727
5       5 0.002687865
6       6 0.002823121
7       7 0.002589057
>
```

Aqui se agruparon las varibles y se obtuvo el promedio de los grupos

La pasa por una variable factor

Bueno, tomó algún tiempo ejecutar el script anterior, pero ten en cuenta que acabamos de agregar alrededor de un cuarto de millón de filas para ver los promedios diarios del número de vuelos desviados que partieron del aeropuerto de Houston en 2011.

En otras palabras, lo que también tiene sentido para todos aquellos que no están interesados en estadísticas, el porcentaje de vuelos desviados por día de la semana. Los resultados son bastante interesantes, ya que parece que los vuelos se desvían con más frecuencia a mitad de semana (alrededor del 0.3 por ciento) que durante los fines de semana (alrededor del 0.05 por ciento menos), al menos desde Houston.

Una forma alternativa de llamar a la función anterior es proporcionar los argumentos dentro de la función *with*, que parece ser una expresión más amigable para los humanos, después de todo, porque nos salva de la mención repetida de la base de datos hflights:

```
> with(hflights, aggregate(Diverted, by = list(DayOfWeek),
+ FUN = mean))
  Group.1           x
1       1 0.002997672
2       2 0.002559323
3       3 0.003226211
4       4 0.003065727
5       5 0.002687865
6       6 0.002823121
7       7 0.002589057
>
```

Simplificacion de la funcion agregate para solo escribir una vez el nombre del dataset o dataframe

Los resultados que se muestran aquí son exactamente los mismos que los mostrados anteriormente. El manual de la función aggregate (ver? aggregate) establece que devuelve los resultados en una forma conveniente. Bueno, comprobar los nombres de las columnas de los datos devueltos antes mencionados no parece conveniente, ¿verdad? Podemos solucionar este problema utilizando la notación de fórmulas en lugar de definir las variables numéricas y factoriales por separado:

```
> aggregate(Diverted ~ DayOfWeek, data = hflights, FUN = mean)
  DayOfWeek    Diverted
1         1 0.002997672
2         2 0.002559323
3         3 0.003226211
4         4 0.003065727
5         5 0.002687865
6         6 0.002823121
7         7 0.002589057
>
```

Notacion para agregar nombres a las columnas (notacion de formulas)

La ganancia mediante el uso de la notación de fórmula es al menos doble:

- Hay relativamente pocos caracteres para escribir

- Los encabezados y los nombres de las filas son correctos en los resultados.
- Esta versión también se ejecuta un poco más rápido que las llamadas agregadas anteriores; consulta el punto de referencia total al final de esta sección

La única desventaja de usar la notación de fórmulas es que tienes que aprenderla, lo que puede parecer un poco incómodo al principio, pero como las fórmulas se usan mucho en un montón de funciones y paquetes de R, particularmente para definir modelos, definitivamente vale la pena aprender cómo para que usarlos a largo plazo.

Nota. La notación de fórmulas se hereda del lenguaje S con la siguiente sintaxis general: response_variable ~ predictor_variable_1 +… + Predictor_variable_n. La notación también incluye algunos otros símbolos, como - para excluir variables y: o * para incluir la interacción entre las variables con o sin ellas mismas.

Agregación más rápida con comandos base R

Una solución alternativa para agregar datos podría ser llamar a la función tapply o by, que puedes aplicar una función R sobre una matriz *irregular*. Esto último significa que podemos proporcionar una o más variables INDEX, que serán coaccionadas para factorizar, y luego, ejecutar la función R proporcionada por separado en todas las celdas de cada subconjunto. El siguiente es un ejemplo rápido:

```
> tapply(hflights$Diverted, hflights$DayOfWeek, mean)
        1         2         3         4         5         6         7
0.002997672 0.002559323 0.003226211 0.003065727 0.002687865 0.002823121 0.002589057
```

Ten en cuenta que tapply devuelve un objeto arreglo en lugar de marcos de datos convenientes; por otro lado, se ejecuta mucho más rápido que las llamadas agregadas mencionadas anteriormente. Por lo tanto, podría ser razonable usar tapply para los cálculos y luego convertir los resultados a data.frame con los nombres de columna apropiados.

Funciones convenientes de ayuda

Dichas conversiones se pueden realizar fácilmente y de una manera muy fácil de usar, por ejemplo, utilizando el paquete plyr, una versión general del paquete dplyr, que significa plyr especializado para marcos de datos.

El paquete plyr proporciona una variedad de funciones para aplicar datos de data.frame, list o arreglos de objetos, y puede devolver los resultados en cualquiera de los formatos mencionados.

El esquema de nombrado de estas funciones es fácil de recordar: el primer carácter del nombre de la función representa la clase de los datos de entrada y el segundo carácter representa el formato de salida, todo seguido de ply en todos los casos. Además de las tres clases R mencionadas anteriormente, hay algunas opciones especiales codificadas por los caracteres:

- d significa data.frame
- s significa array
- l significa lista
- m es un tipo de entrada especial, lo que significa que proporcionamos múltiples argumentos en un formato tabular para la función
- r tipo de entrada que espera un número entero, que especifica el número de veces que se replicará la función
- _ es un tipo de salida especial que no devuelve nada para la función

Por lo tanto, están disponibles las siguientes combinaciones más utilizadas:

- ddply toma data.frame como entrada y devuelve data.frame
- ldply toma la lista como entrada pero devuelve data.frame
- l_ply no devuelve nada, pero es realmente útil, por ejemplo, iterar a través de varios elementos en lugar de un ciclo for; como con un argumento .progress establecido, la función puede mostrar el estado actual de las iteraciones, el tiempo restante

Encuentra más detalles, ejemplos y casos de uso de plyr en la referencia del paquete. Aquí, solo nos concentraremos en cómo resumir los datos. Con este fin, usaremos ddply (que no debe confundirse con el paquete dplyr) en todos los siguientes ejemplos: tomando data.frame como argumento de entrada y devolviendo datos con la misma clase.

Entonces, carguemos el paquete y apliquemos la función media en la columna Diverted sobre cada subconjunto por DayOfWeek:

```
> library(plyr)
> ddply(hflights, .(DayOfWeek), function(x) mean(x$Diverted))
  DayOfWeek          V1
1         1 0.002997672
2         2 0.002559323
3         3 0.003226211
4         4 0.003065727
5         5 0.002687865
6         6 0.002823121
7         7 0.002589057
>
```

Aplica funciones al dataframe, lista, entre otros y devuelve el tipo de dato especificado, en este caso tambien un dataframe.

Uso de función anónima

Nota. La función . del paquete plyr nos proporciona una forma conveniente de referirnos a una variable (name) tal cual; de lo contrario, el contenido de las columnas DayOfWeek sería interpretado por ddply, dando como resultado un error.
Una cosa importante a tener en cuenta aquí es que ddply es mucho más rápido que nuestro primer intento con la función aggregate. Por otro lado, todavía no estamos satisfechos con los resultados, V1 y nombres de columnas tan creativos siempre nos han asustado. En lugar de actualizar los nombres del procesamiento posterior de data.frame, llamemos a la función auxiliar de resumen en lugar de la anónima aplicada previamente; aquí, también podemos proporcionar el nombre deseado para nuestra columna recién calculada:

```
> ddply(hflights, .(DayOfWeek), summarise, Diverted = mean(Diverted))
  DayOfWeek    Diverted
1         1 0.002997672
2         2 0.002559323
3         3 0.003226211
4         4 0.003065727
5         5 0.002687865
6         6 0.002823121
7         7 0.002589057
>
```

Aplica la funcion a la columna, calcula dicha funcion y asigna el nombre separando en grupos de dias de la semana.

De acuerdo, mucho mejor. Pero, ¿podemos hacerlo aún mejor?

Funciones auxiliares de alto rendimiento

Hadley Wickham, autor de ggplot, reshape y varios otros paquetes de R, comenzó a trabajar en la segunda generación, o más bien en una versión especializada, de plyr en 2008. El concepto básico era que plyr se usa con mayor frecuencia para transformar un data.frame a otro data.frame; por tanto, su funcionamiento requiere una atención especial. El paquete dplyr, plyr especializado para marcos de datos, proporciona una implementación más rápida de las funciones plyr, escritas en C++ sin formato, y dplyr también puede trabajar con bases de datos remotas.

Sin embargo, las mejoras de rendimiento también van de la mano con algunos otros cambios; por ejemplo, la sintaxis de dplyr ha cambiado mucho en comparación con plyr.
Aunque la función de resumen mencionada anteriormente existe en dplyr, ya no tenemos la función ddplyr, ya que todas las funciones del paquete están dedicadas a actuar como algún componente de plyr :: ddplyr.

De todos modos, para que el trasfondo teórico sea breve, si queremos resumir los subgrupos de un conjunto de datos, tenemos que definir los grupos antes de la agregación:

```
> library(dplyr)
> hflights_DayOfWeek <- group_by(hflights, DayOfWeek)
```

El objeto resultante es el mismo data.frame que teníamos anteriormente, con una excepción: un grupo de metadatos se fusionó con el objeto por medio de atributos.
Para que el siguiente resultado sea breve, no enumeramos la estructura completa (str) del objeto, sino que solo se muestran los atributos:

```
> str(attributes(hflights_DayOfWeek))
List of 4
 $ names    : chr [1:21] "Year" "Month" "DayofMonth" "DayOfWeek" ...
 $ row.names: int [1:227496] 1 2 3 4 5 6 7 8 9 10 ...
 $ groups   : tibble [7 x 2] (S3: tbl_df/tbl/data.frame)
  ..$ DayOfWeek: int [1:7] 1 2 3 4 5 6 7
  ..$ .rows    : list<int> [1:7]
  .. ..$ : int [1:34360] 3 10 17 24 31 34 41 48 55 62 ...
  .. ..$ : int [1:31649] 4 11 18 25 35 42 49 56 65 71 ...
  .. ..$ : int [1:31926] 5 12 19 26 36 43 50 57 66 72 ...
  .. ..$ : int [1:34902] 6 13 20 27 37 44 51 58 67 73 ...
  .. ..$ : int [1:34972] 7 14 21 28 38 45 52 59 68 74 ...
  .. ..$ : int [1:27629] 1 8 15 22 29 32 39 46 53 60 ...
  .. ..$ : int [1:32058] 2 9 16 23 30 33 40 47 54 61 ...
  .. ..@ ptype: int(0)
  ..- attr(*, ".drop")= logi TRUE
 $ class    : chr [1:4] "grouped_df" "tbl_df" "tbl" "data.frame"
>
```

A partir de estos metadatos, el atributo indices es importante. Simplemente enumera los ID de cada fila para uno de los días de la semana, por lo que las operaciones posteriores pueden seleccionar fácilmente los subgrupos de todo el conjunto de datos. Entonces, veamos cómo se ve la proporción de vuelos desviados con un aumento de rendimiento debido al uso de resumen de dplyr en lugar de plyr:

```
> dplyr::summarise(hflights_DayOfWeek, mean(Diverted))
`summarise()` ungrouping output (override with `.groups` argument)
# A tibble: 7 x 2
  DayOfWeek `mean(Diverted)`
      <int>            <dbl>
1         1          0.00300
2         2          0.00256
3         3          0.00323
4         4          0.00307
5         5          0.00269
6         6          0.00282
7         7          0.00259
>
```

Resumen de medias en el conjunto, divididos por los días de la semana

ESTO ES MUCHO MAS RAPIDO

Los resultados son bastante familiares, lo cual es bueno. Sin embargo, mientras ejecutaba este ejemplo, ¿mediste el tiempo de ejecución? Esto fue casi un instante, lo que hace que dplyr sea aún mejor.

Agregado con data.table

Recuerdas el segundo argumento de [.data.table? Se llama j, que significa SELECT o UPDATE SQL, y la característica más importante es que puede ser cualquier expresión R. Por lo tanto, podemos simplemente pasar una función allí y establecer grupos con la ayuda del argumento by:

```
> hflights_dt[, mean(Diverted), by = DayOfWeek]
   DayOfWeek          V1
1:         6 0.002823121
2:         7 0.002589057
3:         1 0.002997672
4:         2 0.002559323
5:         3 0.003226211
6:         4 0.003065727
7:         5 0.002687865
```

Usando data frame se obtiene el promedio

Es aun mas rápido, pero da desordenado

Estamos bastante seguro de que no te sorprende lo más mínimo la rapidez con la que data.table devolvió los resultados, ya que la gente puede acostumbrarse a las grandes herramientas muy rápidamente.
Además, fue muy conciso en comparación con la anterior llamada dplyr de dos líneas, verdad?
El único inconveniente de esta solución es que los días de la semana están ordenados por un rango apenas inteligible. Existe un tema importante que trata sobre la reestructuración de datos, para obtener más detalles al respecto; por ahora, solucionemos el problema rápidamente configurando una llave, lo que significa que ordenamos data.table primero por DayOfWeek:

```
> setkey(hflights_dt, 'DayOfWeek')
> hflights_dt[, mean(Diverted), by = DayOfWeek]
   DayOfWeek          V1
1:         1 0.002997672
2:         2 0.002559323
3:         3 0.003226211
4:         4 0.003065727
5:         5 0.002687865
6:         6 0.002823121
7:         7 0.002589057
>
```

Aqui primero se configura una llave para ordenar y apicar promedio, ya da el resultado como debe

Nota. Para especificar un nombre para la segunda columna en el objeto tabular resultante en lugar de V1, puedes especificar el objeto de resumen (*summary*) como una lista con nombre, por ejemplo, como hflights_dt [, list ('mean (Diverted)' = mean (Diverted)), by = DayOfWeek], donde puede usar. (dot) en lugar de list, como en plyr.

Además de obtener los resultados en el orden esperado, el resumen de datos mediante una llave ya existente también se ejecuta relativamente rápido. ¡Verifiquemos esto con alguna evidencia empírica en tu máquina!

Ejecución de benchmarks

Como ya se discutió anteriormente, con la ayuda del paquete microbenchmark, podemos ejecutar cualquier número de funciones diferentes durante un número específico de veces en la misma máquina para obtener resultados reproducibles en el rendimiento.

Para ello, primero tenemos que definir las funciones que queremos comparar. Estos fueron compilados a partir de los ejemplos anteriores:

```
> AGGR1 <- function() aggregate(hflights$Diverted,
+ by = list(hflights$DayOfWeek), FUN = mean)
> AGGR2 <- function() with(hflights, aggregate(Diverted,
+ by = list(DayOfWeek), FUN = mean))
> AGGR3 <- function() aggregate(Diverted ~ DayOfWeek,
+ data = hflights, FUN = mean)
> TAPPLY <- function() tapply(X = hflights$Diverted,
+ INDEX = hflights$DayOfWeek, FUN = mean)
> PLYR1 <- function() ddply(hflights, .(DayOfWeek),
+ function(x) mean(x$Diverted))
> PLYR2 <- function() ddply(hflights, .(DayOfWeek), summarise,
+ Diverted = mean(Diverted))
> DPLYR <- function() dplyr::summarise(hflights_DayOfWeek,
+ mean(Diverted))
>
```

Sin embargo, como se mencionó anteriormente, la función de resumen en dplyr necesita una reestructuración de datos previa, lo que también lleva tiempo. Para ello, definamos otra función que también incluye la creación de la nueva estructura de datos junto con la agregación real:

```
> DPLYR_ALL <- function() {
+ hflights_DayOfWeek <- group_by(hflights, DayOfWeek)
+ dplyr::summarise(hflights_DayOfWeek, mean(Diverted))
+ }
```

Funcion de agregacion completa

De manera similar, la evaluación comparativa data.table también requiere algunas variables adicionales para el entorno de prueba; como hlfights_dt ya está ordenado por DayOfWeek, creemos un nuevo objeto data.table para la evaluación comparativa:

```
> hflights_dt_nokey <- data.table(hflights)
```

Además, probablemente tenga sentido verificar que no tenga claves:

```
> key(hflights_dt_nokey)
NULL
```

Bien, ahora podemos definir los casos de prueba de data.table junto con una función que también incluye la transformación a data.table y agregar un índice solo para ser justos con dplyr:

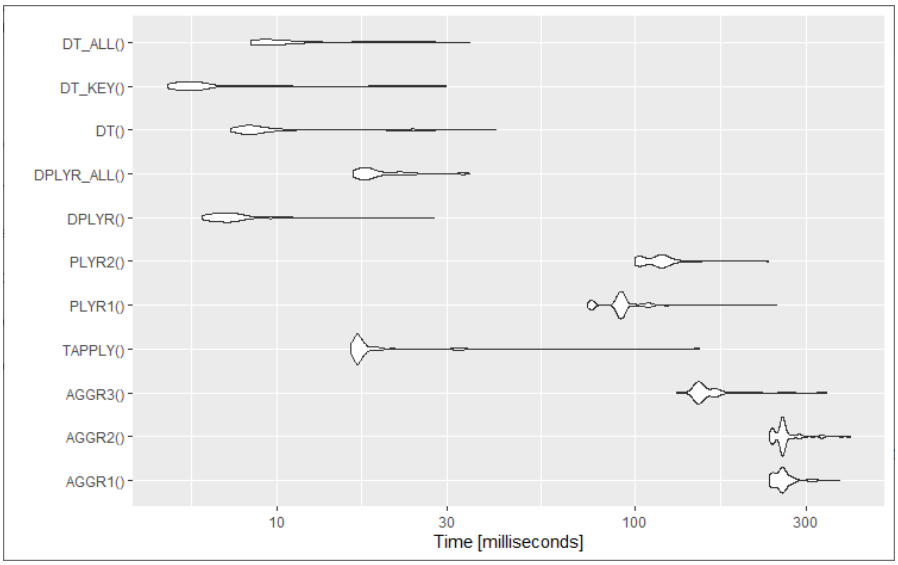
Ahora que tenemos todas las implementaciones descritas listas para probar, carguemos el paquete microbenchmark para que haga su trabajo:

```
> install.packages("microbenchmark")
> library(microbenchmark)
> res <- microbenchmark(AGGR1(), AGGR2(), AGGR3(), TAPPLY(), PLYR1(),
+ PLYR2(), DPLYR(), DPLYR_ALL(), DT(), DT_KEY(), DT_ALL())
```

```
> print(res, digits=3)
Unit: milliseconds
        expr    min     lq   mean median     uq   max neval
     AGGR1() 236.76 246.84 261.15 257.57 263.97 371.7   100
     AGGR2() 236.77 252.67 265.39 258.34 263.66 400.6   100
     AGGR3() 131.29 149.16 159.43 152.00 165.98 342.4   100
    TAPPLY()  16.11  16.71  20.92  17.00  17.66 151.7   100
     PLYR1()  73.56  89.07  94.78  91.65  93.94 248.4   100
     PLYR2() 100.05 103.59 115.87 117.20 120.45 236.1   100
     DPLYR()   6.21   6.80   7.97   7.37   8.00  27.6   100
 DPLYR_ALL()  16.30  17.37  19.84  17.92  21.29  34.6   100
        DT()   7.45   8.22  11.99   8.69  10.46  41.1   100
    DT_KEY()   4.96   5.44   7.82   5.90   6.38  29.8   100
    DT_ALL()   8.48   9.10  12.79   9.55  14.70  34.6   100
>
```

Los resultados son bastante espectaculares: a partir de más de 2000 milisegundos, podríamos mejorar nuestras herramientas para proporcionar los mismos resultados en solo un poco más de 1 milisegundo. La extensión se puede demostrar fácilmente en un diagrama de violín con una escala logarítmica:

```
> install.packages("ggplot2")
> library(ggplot2)
> autoplot(res)
```



dplyr es mas eficiente aunque si ya hay un data.table cargado y se puede guardar data.frame en data.table este resulta ser mas eficiente

Por lo tanto, dplyr parece ser la solución más eficiente, aunque si también tomamos en cuenta el paso adicional (agrupar data.frame), hace que la clara ventaja sea poco convincente. De hecho, si ya tenemos un objeto data.table, y podemos guardar la transformación de un objeto data.frame tradicional en data.table, entonces data.table funciona mejor que dplyr. Sin embargo, estamos bastante seguro de que realmente no notarás la diferencia de tiempo entre las dos soluciones de alto rendimiento; ambos hacen un muy buen trabajo con conjuntos de datos aún más grandes.

Vale la pena mencionar que dplyr también puede funcionar con objetos data.table; por lo tanto, para asegurarse de que no está bloqueado en ninguno de los paquetes, definitivamente vale la pena usar ambos si es necesario. El siguiente es un ejemplo de POC:

```
> dplyr::summarise(group_by(hflights_dt, DayOfWeek), mean(Diverted))
`summarise()` ungrouping output (override with `.groups` argument)
# A tibble: 7 x 2
  DayOfWeek `mean(Diverted)`
      <int>            <dbl>
1         1          0.00300
2         2          0.00256
3         3          0.00323
4         4          0.00307
5         5          0.00269
6         6          0.00282
7         7          0.00259
>
```

Bien, ahora estamos bastante seguros de que usaremos data.table o dplyr para calcular promedios de grupo en el futuro. Sin embargo, ¿qué pasa con las operaciones más complejas?

Funciones de resumen

Como hemos discutido anteriormente, todas las funciones de agregación pueden tomar cualquier función R válida para aplicar en los subconjuntos de los datos. Algunos de los paquetes R lo hacen extremadamente fácil para los usuarios, mientras que algunas funciones requieren que comprenda completamente el concepto del paquete, la sintaxis personalizada y las opciones para aprovechar al máximo las oportunidades de alto rendimiento.

Para temas más avanzados, consulta literatura sobre reestructuración de datos.

Ahora, nos concentraremos en una función de resumen muy simple, que es extremadamente común en cualquier proyecto de análisis de datos general: contar el número de casos por grupo. Este ejemplo rápido también destacará algunas de las diferencias entre las alternativas mencionadas en este documento.

Sumando el número de casos en subgrupos

Centrémonos ahora en plyr, dplyr y data.table, ya que estamos bastante seguros de que puedes construir las versiones agregadas y tapply sin problemas graves. Sobre la base de los ejemplos anteriores, la tarea actual parece bastante fácil: en lugar de la función media, podemos simplemente llamar a la función de longitud (length) para devolver el número de elementos en la columna Diverted:

```
> ddply(hflights, .(DayOfWeek), summarise, n = length(Diverted))
  DayOfWeek     n
1         1 34360
2         2 31649
3         3 31926
4         4 34902
5         5 34972
6         6 27629
7         7 32058
>
```

Ahora, también sabemos que un número relativamente bajo de vuelos salen de Houston el sábado. Sin embargo, ¿realmente tenemos que escribir tanto para responder una pregunta tan simple? Además, ¿realmente tenemos que nombrar una variable en la que podamos contar el número de casos? Tu ya sabes la respuesta:

```
> ddply(hflights, .(DayOfWeek), nrow)
  DayOfWeek    V1
1         1 34360
2         2 31649
3         3 31926
4         4 34902
5         5 34972
6         6 27629
7         7 32058
>
```

En resumen, no es necesario elegir una variable de data.frame para determinar su longitud, ya que es mucho más fácil (y más rápido) simplemente verificar el número de filas en los (sub) conjuntos de datos.

Sin embargo, también podemos devolver los mismos resultados de una manera mucho más fácil y rápida.
Probablemente, ya has pensado en usar la buena función de tabla antigua para una tarea tan sencilla:

```
> table(hflights$DayOfWeek)

   1    2    3    4    5    6    7
34360 31649 31926 34902 34972 27629 32058
```

El único problema con el objeto resultante es que tenemos que transformarlo más, por ejemplo, a data.frame en la mayoría de los casos. Bueno, plyr ya tiene una función auxiliar para hacer esto en un solo paso, con un nombre muy intuitivo:

```
> count(hflights, 'DayOfWeek')
  "DayOfWeek"      n
1   DayOfWeek 227496
```

Por lo tanto, terminamos con algunos ejemplos bastante simples para contar datos, pero veamos también cómo implementar tablas de resumen con dplyr. Si simplemente intentas modificar nuestros comandos dplyr anteriores, pronto te darás cuenta de que pasar la función length o nrow, como hicimos en plyr, simplemente no funciona. Sin embargo, la lectura de los manuales o algunas preguntas relacionadas en StackOverflow pronto nos llamará la atención sobre una práctica función auxiliar llamada n:

```
> dplyr::summarise(hflights_DayOfWeek, n())
`summarise()` ungrouping output (override with `.groups` argument)
# A tibble: 7 x 2
  DayOfWeek `n()`
      <int> <int>
1         1 34360
2         2 31649
3         3 31926
4         4 34902
5         5 34972
6         6 27629
7         7 32058
>
```

Sin embargo, para ser honesto, ¿realmente necesitamos este enfoque relativamente complejo? Si recuerdas la estructura de hflights_DayOfWeek, pronto te darás cuenta de que hay una forma mucho más fácil y rápida de averiguar el número total de vuelos en cada día de la semana:

```
> attr(hflights_DayOfWeek, 'group_sizes')
NULL
```

Además, solo para asegurarnos de no olvidar la sintaxis personalizada (aunque bonita) de data.table, calculemos los resultados con otra función auxiliar:

```
> hflights_dt[, .N, by = list(DayOfWeek)]
   DayOfWeek     N
1:         1 34360
2:         2 31649
3:         3 31926
4:         4 34902
5:         5 34972
6:         6 27629
7:         7 32058
```

Resumen

En estas notas, presentamos algunas formas efectivas y convenientes de filtrar y resumir datos. Discutimos algunos casos de uso sobre el filtrado de filas y columnas de conjuntos de datos. También aprendimos cómo resumir datos para un análisis más detallado. Después de familiarizarnos con las implementaciones más populares de tales tareas, las comparamos con ejemplos reproducibles y un paquete de evaluación comparativa.

Es sólo una pequeña muestra del trabajo que se puede realizar en R, como vaya siendo necesario, continuaremos más adelante con este viaje de reestructurar conjuntos de datos y crear nuevas variables.