Transformación de datos

De La Cruz Cañar Kevin Santiago

2025-07-29

Introducción

La visualización es clave para generar conocimiento, pero los datos rara vez vienen en la forma necesaria. Es común crear nuevas variables, hacer resúmenes, cambiar nombres o reordenar observaciones. Este capítulo enseña cómo hacerlo usando el paquete `dplyr` y un conjunto de datos sobre vuelos desde Nueva York en 2013.

Prerrequisitos

Este capítulo se enfoca en el uso del paquete `dplyr`, parte central del tidyverse. Las ideas se ilustran con el dataset `vuelos` del paquete `datos`, y se usa `ggplot2` como apoyo para entender los datos.

```
# install.packages("datos")
library(datos)
library(tidyverse)
```

Al cargar el paquete `tidyverse`, se muestra un mensaje de conflictos indicando que `dplyr` sobrescribe funciones de R base. Para usar las versiones originales, debes especificar sus nombres completos: `stats::filter()` y `stats::lag()`.

vuelos

Para explorar los verbos básicos de manipulación de datos de `dplyr`, se utiliza el conjunto de datos `vuelos`, que contiene los 336,776 vuelos que salieron de Nueva York en 2013. Los datos provienen del Departamento de Estadísticas de Transporte de los Estados Unidos y están documentados en `?vuelos`.

```
library(datos)
vuelos
```

```
# A tibble: 336,776 x 19
    anio   mes   dia horario_salida salida_programada atraso_salida
   <int> <int> <int>          <int>              <int>         <dbl>
 1  2013     1     1            517                515             2
 2  2013     1     1            533                529             4
 3  2013     1     1            542                540             2
 4  2013     1     1            544                545            -1
 5  2013     1     1            554                600            -6
 6  2013     1     1            554                558            -4
 7  2013     1     1            555                600            -5
 8  2013     1     1            557                600            -3
 9  2013     1     1            557                600            -3
10  2013     1     1            558                600            -2
# i 336,766 more rows
# i 13 more variables: horario_llegada <int>, llegada_programada <int>,
#   atraso_llegada <dbl>, aerolinea <chr>, vuelo <int>, codigo_cola <chr>,
#   origen <chr>, destino <chr>, tiempo_vuelo <dbl>, distancia <dbl>,
#   hora <dbl>, minuto <dbl>, fecha_hora <dttm>
```

Este conjunto de datos se imprime de forma diferente porque es un tibble, que muestra solo las primeras filas y las columnas que caben en pantalla. Para ver todo el conjunto, se puede usar `View(vuelos)`. Los tibbles son data frames adaptados para funcionar mejor en el tidyverse. Las diferencias se explicarán más adelante.

Debajo de los nombres de las columnas puede aparecer una fila de tres (o cuatro) abreviaturas de letras que describen el tipo de cada variable.

- `int` significa enteros.
- `dbl` significa dobles, o números reales.
- `chr` significa vectores de caracteres o cadenas.
- `dttm` significa fechas y horas (una fecha + una hora).

Hay tres tipos comunes de variables que no se usan en este conjunto de datos, pero que aparecerán más adelante en el libro.

- `lgl` significa lógico, vectores que solo contienen TRUE (verdadero) o FALSE (falso).
- `fctr` significa factores, que R usa para representar variables categóricas con valores posibles fijos.

- `date` significa fechas.

Lo básico de dply

En este capítulo se presentan cinco funciones clave de `dplyr` para la manipulación de datos:

- `filter()`: filtrar observaciones por sus valores.
- `arrange()`: reordenar filas.
- `select()`: seleccionar variables por nombre.
- `mutate()`: crear nuevas variables a partir de otras.
- `summarise()`: resumir múltiples valores en uno solo.

Todas estas funciones pueden combinarse con `group_by()`, que modifica el alcance para operar grupo por grupo. Estas seis funciones forman los verbos del lenguaje de manipulación de datos.

Todos los verbos siguen una estructura similar:

1. El primer argumento es un data frame.
2. Los argumentos siguientes indican qué hacer, usando nombres de variables sin comillas.
3. El resultado es un nuevo data frame.

Estas características facilitan encadenar pasos simples para obtener resultados complejos.

Filtrar filas con `filter()`

`filter()` permite filtrar un subconjunto de observaciones según sus valores. El primer argumento es el nombre del data frame, y los siguientes son las expresiones que aplican el filtro. Por ejemplo, se pueden seleccionar todos los vuelos del 1 de enero con:

```
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
```

```
filter(vuelos, mes == 1, dia == 1)
```

```
# A tibble: 842 x 19
    anio   mes   dia horario_salida salida_programada atraso_salida
   <int> <int> <int>          <int>             <int>         <dbl>
 1  2013     1     1            517               515             2
 2  2013     1     1            533               529             4
 3  2013     1     1            542               540             2
 4  2013     1     1            544               545            -1
 5  2013     1     1            554               600            -6
 6  2013     1     1            554               558            -4
 7  2013     1     1            555               600            -5
 8  2013     1     1            557               600            -3
 9  2013     1     1            557               600            -3
10  2013     1     1            558               600            -2
# i 832 more rows
# i 13 more variables: horario_llegada <int>, llegada_programada <int>,
#   atraso_llegada <dbl>, aerolinea <chr>, vuelo <int>, codigo_cola <chr>,
#   origen <chr>, destino <chr>, tiempo_vuelo <dbl>, distancia <dbl>,
#   hora <dbl>, minuto <dbl>, fecha_hora <dttm>
```

Al ejecutar esa línea de código, `dplyr` realiza el filtrado y devuelve un nuevo data frame. Las funciones de `dplyr` no modifican su entrada, por lo que para guardar el resultado es necesario usar el operador de asignación `<-`.

```
ene1 <- filter(vuelos, mes == 1, dia == 1)
```

R imprime los resultados o los guarda en una variable. Si deseas hacer ambas cosas puedes escribir toda la línea entre paréntesis:

```
(dic25 <- filter(vuelos, mes == 12, dia == 25))
```

```
# A tibble: 719 x 19
    anio   mes   dia horario_salida salida_programada atraso_salida
   <int> <int> <int>          <int>             <int>         <dbl>
 1  2013    12    25            456               500            -4
 2  2013    12    25            524               515             9
 3  2013    12    25            542               540             2
 4  2013    12    25            546               550            -4
 5  2013    12    25            556               600            -4
```

```
 6  2013    12    25           557              600           -3
 7  2013    12    25           557              600           -3
 8  2013    12    25           559              600           -1
 9  2013    12    25           559              600           -1
10  2013    12    25           600              600            0
# i 709 more rows
# i 13 more variables: horario_llegada <int>, llegada_programada <int>,
#   atraso_llegada <dbl>, aerolinea <chr>, vuelo <int>, codigo_cola <chr>,
#   origen <chr>, destino <chr>, tiempo_vuelo <dbl>, distancia <dbl>,
#   hora <dbl>, minuto <dbl>, fecha_hora <dttm>
```

Comparaciones

Para filtrar de forma efectiva, se usan operadores de comparación como `>`, `>=`, `<`, `<=`, `!=` (no igual) y `==` (igual).

Un error común al comenzar con R es usar `=` en lugar de `==` al verificar igualdad, lo que genera un error informativo.

```
filter(vuelos, mes == 1)
```

```
# A tibble: 27,004 x 19
    anio   mes   dia horario_salida salida_programada atraso_salida
   <int> <int> <int>          <int>             <int>         <dbl>
 1  2013     1     1            517               515             2
 2  2013     1     1            533               529             4
 3  2013     1     1            542               540             2
 4  2013     1     1            544               545            -1
 5  2013     1     1            554               600            -6
 6  2013     1     1            554               558            -4
 7  2013     1     1            555               600            -5
 8  2013     1     1            557               600            -3
 9  2013     1     1            557               600            -3
10  2013     1     1            558               600            -2
# i 26,994 more rows
# i 13 more variables: horario_llegada <int>, llegada_programada <int>,
#   atraso_llegada <dbl>, aerolinea <chr>, vuelo <int>, codigo_cola <chr>,
#   origen <chr>, destino <chr>, tiempo_vuelo <dbl>, distancia <dbl>,
#   hora <dbl>, minuto <dbl>, fecha_hora <dttm>
```

Hay otro problema común que puedes encontrar al usar ==: los números de coma flotante. ¡Estos resultados pueden sorprenderte!

```r
sqrt(2)^2 == 2
```

```
[1] FALSE
```

```r
near(1 / 49 * 49, 1)
```

```
[1] TRUE
```

Operadores lógicos

Si usas múltiples argumentos en `filter()`, se combinan con "y": todas las condiciones deben ser verdaderas para incluir una fila. Para otras combinaciones, se usan operadores Booleanos: `&` es "y", `|` es "o" y `!` es "no". La figura `@ref(fig:bool-ops)` muestra todas las operaciones Booleanas.

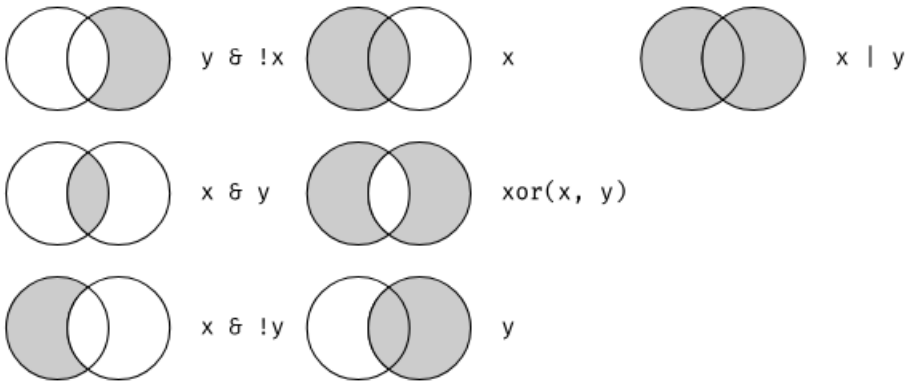


Figura 1. Set completo de operadores booleanos: `x` es el círculo de la derecha, `y` es el círculo de la izquierda y la región sombreada indica qué partes selecciona cada operador.

El siguiente código sirve para encontrar todos los vuelos que partieron en noviembre o diciembre:

```r
filter(vuelos, mes == 11 | mes == 12)
```

El orden de las operaciones no funciona como en español. Por ejemplo, `filter(vuelos, mes == (11 | 12))` no selecciona vuelos de noviembre o diciembre, sino que evalúa `11 | 12` como `TRUE`, que se convierte en 1, filtrando vuelos de enero.

Una solución útil es usar `x %in% y`, que selecciona las filas donde `x` está en el conjunto `y`. Así, el código correcto sería:

```r
nov_dic <- filter(vuelos, mes %in% c(11, 12))
```

A veces puedes simplificar subconjuntos complicados usando la ley de De Morgan: `!(x & y)` es equivalente a `!x | !y`, y `!(x | y)` equivale a `!x & !y`. Por ejemplo, para encontrar vuelos que no se retrasaron (ni en llegada ni en partida) más de dos horas, puedes usar cualquiera de estos filtros:

```r
filter(vuelos, !(atraso_llegada > 120 | atraso_salida > 120))
filter(vuelos  , atraso_llegada <= 120, atraso_salida <= 120)
```

Además de `&` y `|`, R tiene `&&` y `||`, pero no deben usarse en `filter()`. Se explicarán en el capítulo de Ejecución condicional. Al usar expresiones complejas en `filter()`, es útil convertirlas en variables explícitas, lo que facilita verificar los resultados. Aprenderás a crear nuevas variables más adelante.

Valores faltantes

Una característica importante de R que complica las comparaciones son los valores faltantes (`NA`). `NA` representa un valor desconocido y es "contagioso": casi cualquier operación que lo involucre también dará como resultado un valor desconocido.

```r
NA > 5
```

```
[1] NA
```

```r
10 == NA
```

```
[1] NA
```

```r
NA + 10
```

```
[1] NA
```

```r
NA / 2
```

```
[1] NA
```

El resultado más confuso es este:

```
NA == NA
```

```
[1] NA
```

Es más fácil entender por qué esto es cierto con un poco más de contexto:

```
# Sea x la edad de María. No sabemos qué edad tiene.
x <- NA

# Sea y la edad de Juan. No sabemos qué edad tiene.
y <- NA

# ¿Tienen Juan y María la misma edad?
x == y
```

```
[1] NA
```

```
# ¡No sabemos!
```

Si deseas determinar si falta un valor, usa `is.na()`:

```
x <- NA
is.na(x)
```

```
[1] TRUE
```

`filter()` solo incluye filas donde la condición es `TRUE`; excluye tanto `FALSE` como `NA`. Para conservar los valores perdidos, debes solicitarlos explícitamente.

```
df <- tibble(x = c(1, NA, 3))
filter(df, x > 1)
```

```
# A tibble: 1 x 1
      x
  <dbl>
1     3
```

```r
filter(df, is.na(x) | x > 1)
```

```
# A tibble: 2 x 1
      x
  <dbl>
1    NA
2     3
```

Ejercicios

1. Encuentra todos los vuelos que:

- Tuvieron un retraso de llegada de dos o más horas
- Volaron a Houston (`IAH`o `HOU`)
- Fueron operados por United, American o Delta
- Partieron en invierno del hemisferio sur (julio, agosto y septiembre)
- Llegaron más de dos horas tarde, pero no salieron tarde
- Se retrasaron por lo menos una hora, pero repusieron más de 30 minutos en vuelo
- Partieron entre la medianoche y las 6 a.m. (incluyente)

2. Otra función de **dplyr** que es útil para usar filtros es `between()`. ¿Qué hace? ¿Puedes usarla para simplificar el código necesario para responder a los desafíos anteriores?
3. ¿Cuántos vuelos tienen datos faltantes en `horario_salida`? ¿Qué otras variables tienen valores faltantes? ¿Qué representan estas filas?
4. ¿Por qué `NA ^ 0` no es faltante? ¿Por qué `NA | TRUE` no es faltante? ¿Por qué FALSE & NA no es faltante? ¿Puedes descubrir la regla general? (¡`NA * 0` es un contraejemplo complicado!)

Reordenar las filas con `arrange()`

`arrange()` funciona como `filter()`, pero en lugar de seleccionar filas, cambia su orden. Toma un data frame y nombres de columnas (o expresiones) para ordenar. Si se dan varias columnas, las adicionales se usan para desempatar los valores de las anteriores.

```r
arrange(vuelos, anio, mes, dia)
```

```
# A tibble: 336,776 x 19
    anio   mes   dia horario_salida salida_programada atraso_salida
   <int> <int> <int>          <int>             <int>         <dbl>
 1  2013     1     1            517               515             2
 2  2013     1     1            533               529             4
 3  2013     1     1            542               540             2
 4  2013     1     1            544               545            -1
 5  2013     1     1            554               600            -6
 6  2013     1     1            554               558            -4
 7  2013     1     1            555               600            -5
 8  2013     1     1            557               600            -3
 9  2013     1     1            557               600            -3
10  2013     1     1            558               600            -2
# i 336,766 more rows
# i 13 more variables: horario_llegada <int>, llegada_programada <int>,
#   atraso_llegada <dbl>, aerolinea <chr>, vuelo <int>, codigo_cola <chr>,
#   origen <chr>, destino <chr>, tiempo_vuelo <dbl>, distancia <dbl>,
#   hora <dbl>, minuto <dbl>, fecha_hora <dttm>
```

Usa `desc()` para reordenar por una columna en orden descendente:

```
arrange(vuelos, desc(atraso_salida))
```

```
# A tibble: 336,776 x 19
    anio   mes   dia horario_salida salida_programada atraso_salida
   <int> <int> <int>          <int>             <int>         <dbl>
 1  2013     1     9            641               900          1301
 2  2013     6    15           1432              1935          1137
 3  2013     1    10           1121              1635          1126
 4  2013     9    20           1139              1845          1014
 5  2013     7    22            845              1600          1005
 6  2013     4    10           1100              1900           960
 7  2013     3    17           2321               810           911
 8  2013     6    27            959              1900           899
 9  2013     7    22           2257               759           898
10  2013    12     5            756              1700           896
# i 336,766 more rows
# i 13 more variables: horario_llegada <int>, llegada_programada <int>,
#   atraso_llegada <dbl>, aerolinea <chr>, vuelo <int>, codigo_cola <chr>,
#   origen <chr>, destino <chr>, tiempo_vuelo <dbl>, distancia <dbl>,
#   hora <dbl>, minuto <dbl>, fecha_hora <dttm>
```

Los valores faltantes siempre se ordenan al final:

```
df <- tibble(x = c(5, 2, NA))
arrange(df, x)
```

```
# A tibble: 3 x 1
      x
  <dbl>
1     2
2     5
3    NA
```

```
arrange(df, desc(x))
```

```
# A tibble: 3 x 1
      x
  <dbl>
1     5
2     2
3    NA
```

Ejercicios

1. ¿Cómo podrías usar `arrange()` para ordenar todos los valores faltantes al comienzo? (Sugerencia: usa `is.na()`).
2. Ordena `vuelos` para encontrar los vuelos más retrasados. Encuentra los vuelos que salieron más temprano.
3. Ordena `vuelos` para encontrar los vuelos más rápidos (que viajaron a mayor velocidad).
4. ¿Cuáles vuelos viajaron más lejos? ¿Cuál viajó más cerca?

Seleccionar columnas con `select()`

No es raro tener conjuntos de datos con cientos de variables, y el primer paso suele ser reducirlas a las que realmente interesan. `select()` permite seleccionar rápidamente un subconjunto útil basado en los nombres de las variables. Aunque con los datos de vuelos, que solo tienen 19 variables, no se ve su mayor utilidad, se entiende la idea general.

```r
# Seleccionar columnas por nombre
select(vuelos, anio, mes, dia)
```

```
# A tibble: 336,776 x 3
    anio   mes   dia
   <int> <int> <int>
 1  2013     1     1
 2  2013     1     1
 3  2013     1     1
 4  2013     1     1
 5  2013     1     1
 6  2013     1     1
 7  2013     1     1
 8  2013     1     1
 9  2013     1     1
10  2013     1     1
# i 336,766 more rows
```

```r
# Seleccionar todas las columnas entre anio y dia (incluyente)
select(vuelos, anio:dia)
```

```
# A tibble: 336,776 x 3
    anio   mes   dia
   <int> <int> <int>
 1  2013     1     1
 2  2013     1     1
 3  2013     1     1
 4  2013     1     1
 5  2013     1     1
 6  2013     1     1
 7  2013     1     1
 8  2013     1     1
 9  2013     1     1
10  2013     1     1
# i 336,766 more rows
```

```r
# Seleccionar todas las columnas excepto aquellas entre anio en dia (incluyente)
select(vuelos, -(anio:dia))
```

```
# A tibble: 336,776 x 16
```

```
   horario_salida salida_programada atraso_salida horario_llegada
            <int>              <int>         <dbl>           <int>
 1            517                515             2             830
 2            533                529             4             850
 3            542                540             2             923
 4            544                545            -1            1004
 5            554                600            -6             812
 6            554                558            -4             740
 7            555                600            -5             913
 8            557                600            -3             709
 9            557                600            -3             838
10            558                600            -2             753
# i 336,766 more rows
# i 12 more variables: llegada_programada <int>, atraso_llegada <dbl>,
#   aerolinea <chr>, vuelo <int>, codigo_cola <chr>, origen <chr>,
#   destino <chr>, tiempo_vuelo <dbl>, distancia <dbl>, hora <dbl>,
#   minuto <dbl>, fecha_hora <dttm>
```

Hay funciones auxiliares que se pueden usar dentro de `select()`:

- `starts_with("abc")`: coincide con nombres que comienzan con "abc".
- `ends_with("xyz")`: coincide con nombres que terminan con "xyz".
- `contains("ijk")`: coincide con nombres que contienen "ijk".
- `matches("(.)\\1")`: selecciona nombres que coinciden con una expresión regular (como caracteres repetidos).
- `num_range("x", 1:3)`: coincide con `x1`, `x2` y `x3`.

Consulta ?select para ver más detalles.

`select()` puede usarse para cambiar el nombre de variables, pero no es muy útil porque descarta las que no se mencionan. En su lugar, se recomienda usar `rename()`, una variante de `select()` que conserva todas las variables no mencionadas.

```
rename(vuelos, cola_num = codigo_cola)
```

```
# A tibble: 336,776 x 19
    anio   mes   dia horario_salida salida_programada atraso_salida
   <int> <int> <int>          <int>              <int>         <dbl>
 1  2013     1     1            517                515             2
 2  2013     1     1            533                529             4
 3  2013     1     1            542                540             2
 4  2013     1     1            544                545            -1
 5  2013     1     1            554                600            -6
```

```
 6  2013     1     1            554              558            -4
 7  2013     1     1            555              600            -5
 8  2013     1     1            557              600            -3
 9  2013     1     1            557              600            -3
10  2013     1     1            558              600            -2
# i 336,766 more rows
# i 13 more variables: horario_llegada <int>, llegada_programada <int>,
#   atraso_llegada <dbl>, aerolinea <chr>, vuelo <int>, cola_num <chr>,
#   origen <chr>, destino <chr>, tiempo_vuelo <dbl>, distancia <dbl>,
#   hora <dbl>, minuto <dbl>, fecha_hora <dttm>
```

Otra opción es usar `select()` con el auxiliar `everything()`. Esto es útil cuando se desea mover un grupo de variables al comienzo del data frame.

```
select(vuelos, fecha_hora, tiempo_vuelo, everything())
```

```
# A tibble: 336,776 x 19
   fecha_hora          tiempo_vuelo  anio   mes   dia horario_salida
   <dttm>                     <dbl> <int> <int> <int>          <int>
 1 2013-01-01 05:00:00          227  2013     1     1            517
 2 2013-01-01 05:00:00          227  2013     1     1            533
 3 2013-01-01 05:00:00          160  2013     1     1            542
 4 2013-01-01 05:00:00          183  2013     1     1            544
 5 2013-01-01 06:00:00          116  2013     1     1            554
 6 2013-01-01 05:00:00          150  2013     1     1            554
 7 2013-01-01 06:00:00          158  2013     1     1            555
 8 2013-01-01 06:00:00           53  2013     1     1            557
 9 2013-01-01 06:00:00          140  2013     1     1            557
10 2013-01-01 06:00:00          138  2013     1     1            558
# i 336,766 more rows
# i 13 more variables: salida_programada <int>, atraso_salida <dbl>,
#   horario_llegada <int>, llegada_programada <int>, atraso_llegada <dbl>,
#   aerolinea <chr>, vuelo <int>, codigo_cola <chr>, origen <chr>,
#   destino <chr>, distancia <dbl>, hora <dbl>, minuto <dbl>
```

Ejercicios

1. Haz una lluvia de ideas sobre tantas maneras como sea posible para seleccionar horario_salida,atraso_salida,horario_llegada, yatraso_llegada de vuelos.
2. ¿Qué sucede si incluyes el nombre de una variable varias veces en una llamada a select()?

3. ¿Qué hace la función any_of()? ¡¿Por qué podría ser útil en conjunto con este vector?

```
vars <- c ("anio", "mes", "dia", "atraso_salida", "atraso_llegada")
```

4. ¿Te sorprende el resultado de ejecutar el siguiente código? ¿Cómo tratan por defecto las funciones auxiliares de select() a las palabras en mayúsculas o en minúsculas? ¿Cómo puedes cambiar ese comportamiento predeterminado?

```
select(vuelos, contains("SALIDA"))
```

Añadir nuevas variables con `mutate()`

Además de seleccionar columnas existentes, a menudo es útil crear nuevas a partir de ellas. Para eso se usa `mutate()`. Esta función siempre agrega las nuevas columnas al final del conjunto de datos, por lo que conviene trabajar con un subconjunto más pequeño. En RStudio, la forma más fácil de ver todas las columnas es usando `View()`.

```
vuelos_sml <- select(vuelos,
  anio:dia,
  starts_with("atraso"),
  distancia,
  tiempo_vuelo
)
mutate(vuelos_sml,
  ganancia = atraso_salida - atraso_llegada,
  velocidad = distancia / tiempo_vuelo * 60
)
```

```
# A tibble: 336,776 x 9
    anio   mes   dia atraso_salida atraso_llegada distancia tiempo_vuelo
   <int> <int> <int>         <dbl>          <dbl>     <dbl>        <dbl>
 1  2013     1     1             2             11      1400          227
 2  2013     1     1             4             20      1416          227
 3  2013     1     1             2             33      1089          160
 4  2013     1     1            -1            -18      1576          183
 5  2013     1     1            -6            -25       762          116
 6  2013     1     1            -4             12       719          150
 7  2013     1     1            -5             19      1065          158
 8  2013     1     1            -3            -14       229           53
 9  2013     1     1            -3             -8       944          140
```

```
10  2013     1     1            -2              8       733          138
# i 336,766 more rows
# i 2 more variables: ganancia <dbl>, velocidad <dbl>
```

Ten en cuenta que puedes hacer referencia a las columnas que acabas de crear:

```
mutate(vuelos_sml,
  ganancia = atraso_salida - atraso_llegada,
  horas = tiempo_vuelo / 60,
  ganacia_por_hora = ganancia / horas
)
```

```
# A tibble: 336,776 x 10
    anio   mes   dia atraso_salida atraso_llegada distancia tiempo_vuelo
   <int> <int> <int>         <dbl>          <dbl>     <dbl>        <dbl>
 1  2013     1     1             2             11      1400          227
 2  2013     1     1             4             20      1416          227
 3  2013     1     1             2             33      1089          160
 4  2013     1     1            -1            -18      1576          183
 5  2013     1     1            -6            -25       762          116
 6  2013     1     1            -4             12       719          150
 7  2013     1     1            -5             19      1065          158
 8  2013     1     1            -3            -14       229           53
 9  2013     1     1            -3             -8       944          140
10  2013     1     1            -2              8       733          138
# i 336,766 more rows
# i 3 more variables: ganancia <dbl>, horas <dbl>, ganacia_por_hora <dbl>
```

Si solo quieres conservar las nuevas variables, usa transmute():

```
transmute(vuelos,
  ganancia = atraso_salida - atraso_llegada,
  horas = tiempo_vuelo / 60,
  ganancia_por_hora = ganancia / horas
)
```

```
# A tibble: 336,776 x 3
   ganancia horas ganancia_por_hora
      <dbl> <dbl>             <dbl>
 1       -9 3.78              -2.38
 2      -16 3.78              -4.23
```

```
 3      -31 2.67             -11.6
 4       17 3.05               5.57
 5       19 1.93               9.83
 6      -16 2.5                -6.4
 7      -24 2.63               -9.11
 8       11 0.883              12.5
 9        5 2.33               2.14
10      -10 2.3                -4.35
# i 336,766 more rows
```

Funciones de creación útiles

Hay muchas funciones que se pueden usar con `mutate()` para crear nuevas variables. La clave es que deben ser vectorizadas: toman un vector como entrada y devuelven otro del mismo tamaño. Aunque no se pueden enumerar todas, hay una selección de funciones que suelen ser especialmente útiles.

- Los operadores aritméticos (`+`, `-`, `*`, `/`, `^`) están vectorizados y usan reglas de reciclaje, lo que permite operar entre vectores de distintas longitudes. Esto es útil cuando uno de los argumentos es un solo número, como en `tiempo_vuelo / 60` o `horas * 60 + minuto`. También son útiles con funciones de agregación, por ejemplo: `x / sum(x)` calcula proporciones, `y - mean(y)` calcula diferencias respecto a la media.
- Aritmética modular: `%/%` (división entera) y `%%` (resto), donde `x == y * (x %/% y) + (x %% y)`. Es útil para dividir enteros en partes. Por ejemplo, en el conjunto de datos de vuelos, se puede calcular la hora y los minutos de `horario_salida` con:

```r
transmute(vuelos,
  horario_salida,
  hora = horario_salida %/% 100,
  minuto = horario_salida %% 100
)
```

```
# A tibble: 336,776 x 3
   horario_salida  hora minuto
            <int> <dbl>  <dbl>
 1            517     5     17
 2            533     5     33
 3            542     5     42
 4            544     5     44
 5            554     5     54
 6            554     5     54
```

```
 7            555     5     55
 8            557     5     57
 9            557     5     57
10            558     5     58
# i 336,766 more rows
```

- Logaritmos: `log()`, `log2()`, `log10()`. Son útiles para transformar datos con múltiples órdenes de magnitud y convertir relaciones multiplicativas en aditivas. Se recomienda usar `log2()` porque es más fácil de interpretar: una diferencia de 1 duplica el valor original y una diferencia de -1 lo divide a la mitad.
- Rezagos: `lead()` y `lag()` permiten referirse a un valor siguiente o anterior. Son útiles para calcular diferencias móviles (`x - lag(x)`) o detectar cambios (`x != lag(x)`). Son especialmente útiles junto con `group_by()`, que se abordará más adelante.

```
(x <- 1:10)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```
lag(x)
```

```
 [1] NA  1  2  3  4  5  6  7  8  9
```

```
lead(x)
```

```
 [1]  2  3  4  5  6  7  8  9 10 NA
```

- Agregados acumulativos y móviles: R ofrece funciones como `cumsum()`, `cumprod()`, `cummin()` y `cummax()` para cálculos acumulativos. `dplyr` añade `cummean()` para medias acumuladas. Para agregados móviles, se recomienda el paquete `RcppRoll`.

```
x
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```
cumsum(x)
```

```
 [1]  1  3  6 10 15 21 28 36 45 55
```

```
cummean(x)
```

```
 [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5
```

- Comparaciones lógicas: `<`, `<=`, `>`, `>=`, `!=`, vistas anteriormente. Al realizar operaciones lógicas complejas, es útil guardar los valores intermedios en nuevas variables para verificar que cada paso funcione correctamente.
- Ordenamiento: se dispone de varias funciones de ranking, pero se recomienda comenzar con `min_rank()`, que asigna posiciones como primero, segundo, tercero, etc. Por defecto, los valores más pequeños reciben la menor posición; para que los mayores la reciban, se usa `desc(x)`.

```
y <- c (1, 2, 2, NA, 3, 4)
min_rank(y)
```

```
[1]  1  2  2 NA  4  5
```

```
min_rank(desc(y))
```

```
[1]  5  3  3 NA  2  1
```

Si `min_rank()` no se ajusta a tus necesidades, puedes usar variantes como `row_number()`, `dense_rank()`, `percent_rank()`, `cume_dist()` y `quantile()`. Consulta sus páginas de ayuda para más detalles.

```
row_number(y)
```

```
[1]  1  2  3 NA  4  5
```

```
dense_rank(y)
```

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

```
percent_rank(y)
```

```
[1] 0.00 0.25 0.25   NA 0.75 1.00
```

```r
cume_dist(y)
```

```
[1] 0.2 0.6 0.6  NA 0.8 1.0
```

Ejercicios

1. Las variables `horario_salida` y `salida_programada` tienen un formato conveniente para leer, pero es difícil realizar cualquier cálculo con ellas porque no son realmente números continuos. Transfórmalas hacia un formato más conveniente como número de minutos desde la medianoche.
2. Compara `tiempo_vuelo` con `horario_llegada - horario_salida`. ¿Qué esperas ver? ¿Qué ves? ¿Qué necesitas hacer para arreglarlo?
3. Compara `horario_salida`, `salida_programada`, y `atraso_salida`. ¿Cómo esperarías que esos tres números estén relacionados?
4. Encuentra los 10 vuelos más retrasados utilizando una función de ordenamiento. ¿Cómo quieres manejar los empates? Lee atentamente la documentación de `min_rank()`.
5. ¿Qué devuelve `1:3 + 1:10`? ¿Por qué?
6. ¿Qué funciones trigonométricas proporciona R?

Resúmenes agrupados con `summarise()`

El último verbo clave es summarise() (resumir, en inglés). Se encarga de colapsar un data frame en una sola fila:

```r
summarise(vuelos, atraso = mean(atraso_salida, na.rm = TRUE))
```

```
# A tibble: 1 x 1
  atraso
   <dbl>
1   12.6
```

(Volveremos a lo que significa `na.rm = TRUE` en muy poco tiempo).

`summarise()` no es muy útil por sí sola, pero al combinarse con `group_by()` cambia la unidad de análisis a grupos. Así, los verbos de `dplyr` se aplican automáticamente por grupo. Por ejemplo, al agrupar por fecha y luego resumir, se obtiene el retraso promedio por fecha.

```r
por_dia <- group_by(vuelos, anio, mes, dia)
summarise(por_dia, atraso = mean(atraso_salida, na.rm = TRUE))
```

```
`summarise()` has grouped output by 'anio', 'mes'. You can override using the
`.groups` argument.
```

```
# A tibble: 365 x 4
# Groups:   anio, mes [12]
    anio   mes   dia atraso
   <int> <int> <int>  <dbl>
 1  2013     1     1  11.5
 2  2013     1     2  13.9
 3  2013     1     3  11.0
 4  2013     1     4   8.95
 5  2013     1     5   5.73
 6  2013     1     6   7.15
 7  2013     1     7   5.42
 8  2013     1     8   2.55
 9  2013     1     9   2.28
10  2013     1    10   2.84
# i 355 more rows
```

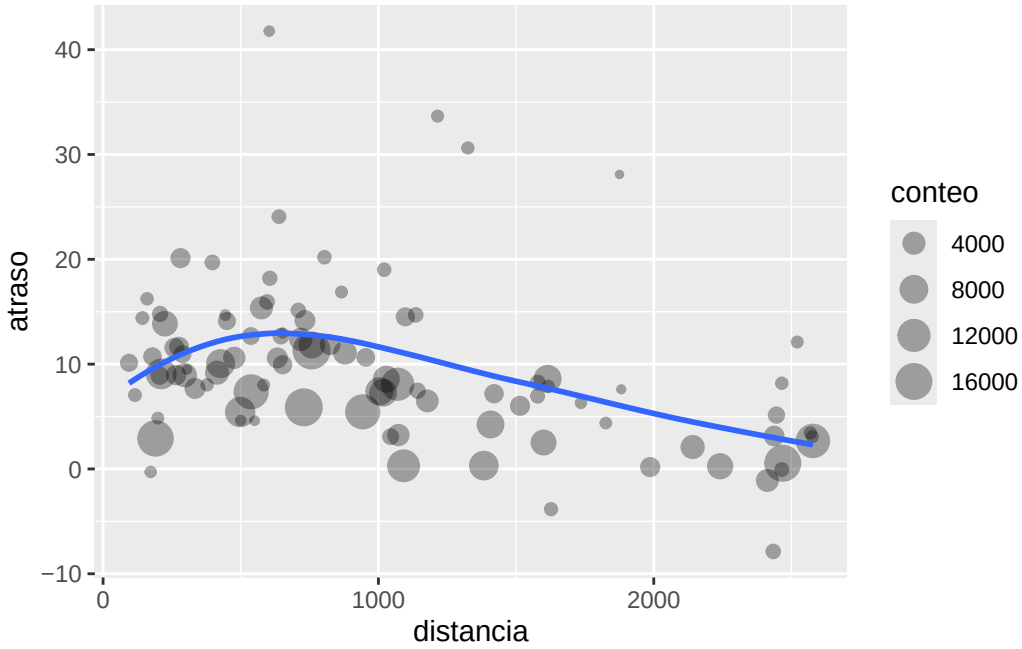
Juntos, `group_by()` y `summarise()` permiten crear resúmenes agrupados, una de las herramientas más comunes en `dplyr`. Antes de profundizar en esto, es necesario introducir una idea nueva y poderosa: el *pipe* (`|>`), que significa ducto o tubería.

Combinación de múltiples operaciones con el *pipe*

Imagina que queremos explorar la relación entre la distancia y el atraso promedio para cada ubicación. Usando lo que sabes acerca de dplyr, podrías escribir un código como este:

```r
library(ggplot2)
por_destino <- group_by(vuelos, destino)
atraso <- summarise(por_destino,
  conteo = n(),
  distancia = mean(distancia, na.rm = TRUE),
  atraso = mean(atraso_llegada, na.rm = TRUE)
)
atraso <- filter(atraso, conteo > 20, destino != "HNL")
```

```r
# Parece que las demoras aumentan con las distancias hasta ~ 750 millas
# y luego disminuyen. ¿Tal vez a medida que los vuelos se hacen más
# largos, hay más habilidad para compensar las demoras en el aire?

ggplot(data = atraso, mapping = aes(x = distancia, y = atraso)) +
  geom_point(aes(size = conteo), alpha = 1/3) +
  geom_smooth(se = FALSE)
```



Hay tres pasos para preparar esta información:

1. Agrupar los vuelos por destino.
2. Resumir para calcular la distancia, la demora promedio y el número de vuelos en cada grupo.
3. Filtrar para eliminar puntos ruidosos y el aeropuerto de Honolulu, que está casi dos veces más lejos que el próximo aeropuerto más cercano.

Es frustrante tener que nombrar cada data frame intermedio, especialmente cuando no nos interesa conservarlos. Nombrar cosas puede ser difícil y ralentiza el análisis.

Hay otra forma de abordar el mismo problema con el pipe, `%>%`:

```r
atrasos <- vuelos %>%
  group_by(destino) %>%
  summarise(
    conteo = n(),
    distancia = mean(distancia, na.rm = TRUE),
    atraso = mean(atraso_llegada, na.rm = TRUE)
  ) %>%
  filter(conteo > 20, destino != "HNL")
```

Este código se enfoca en las transformaciones, no en lo que se transforma, lo que facilita su lectura. Puede interpretarse como una serie de instrucciones: agrupa, luego resume, luego filtra. Una buena forma de leer `%>%` es como "luego".

En términos técnicos, `x %>% f(y)` se convierte en `f(x, y)`, y `x %>% f(y) %>% g(z)` se convierte en `g(f(x, y), z)`. El pipe permite reescribir operaciones para leer de izquierda a derecha, de arriba hacia abajo. Se usará con frecuencia porque mejora la legibilidad.

El uso del pipe es clave en el tidyverse, con la excepción de `ggplot2`, que fue creado antes. Su sucesor, `ggvis`, aún no está listo para uso general.

Valores faltantes

Es posible que te hayas preguntado sobre el argumento na.rm que utilizamos anteriormente. ¿Qué pasa si no lo configuramos?

```r
vuelos %>%
  group_by(anio, mes, dia) %>%
  summarise(mean = mean(atraso_salida))
```

```
`summarise()` has grouped output by 'anio', 'mes'. You can override using the
`.groups` argument.
```

```
# A tibble: 365 x 4
# Groups:   anio, mes [12]
    anio   mes   dia  mean
   <int> <int> <int> <dbl>
 1  2013     1     1    NA
 2  2013     1     2    NA
 3  2013     1     3    NA
 4  2013     1     4    NA
 5  2013     1     5    NA
```

```
 6  2013     1     6    NA
 7  2013     1     7    NA
 8  2013     1     8    NA
 9  2013     1     9    NA
10  2013     1    10    NA
# i 355 more rows
```

¡Obtenemos muchos valores faltantes! Esto ocurre porque las funciones de agregación siguen la regla de que, si hay un valor faltante en la entrada, el resultado también será faltante. Afortunadamente, todas estas funciones tienen el argumento `na.rm` para eliminar los valores faltantes antes de calcular.

```
vuelos %>%
  group_by(anio, mes, dia) %>%
  summarise(mean = mean(atraso_salida, na.rm = TRUE))
```

```
`summarise()` has grouped output by 'anio', 'mes'. You can override using the
`.groups` argument.
```

```
# A tibble: 365 x 4
# Groups:   anio, mes [12]
    anio   mes   dia  mean
   <int> <int> <int> <dbl>
 1  2013     1     1 11.5
 2  2013     1     2 13.9
 3  2013     1     3 11.0
 4  2013     1     4  8.95
 5  2013     1     5  5.73
 6  2013     1     6  7.15
 7  2013     1     7  5.42
 8  2013     1     8  2.55
 9  2013     1     9  2.28
10  2013     1    10  2.84
# i 355 more rows
```

En este caso, donde los valores faltantes corresponden a vuelos cancelados, una opción es eliminarlos antes del análisis. Se puede guardar este conjunto de datos depurado para reutilizarlo en los siguientes ejemplos.

```r
no_cancelados <- vuelos %>%
  filter(!is.na(atraso_salida), !is.na(atraso_llegada))

no_cancelados %>%
  group_by(anio, mes, dia) %>%
  summarise(mean = mean(atraso_salida))
```

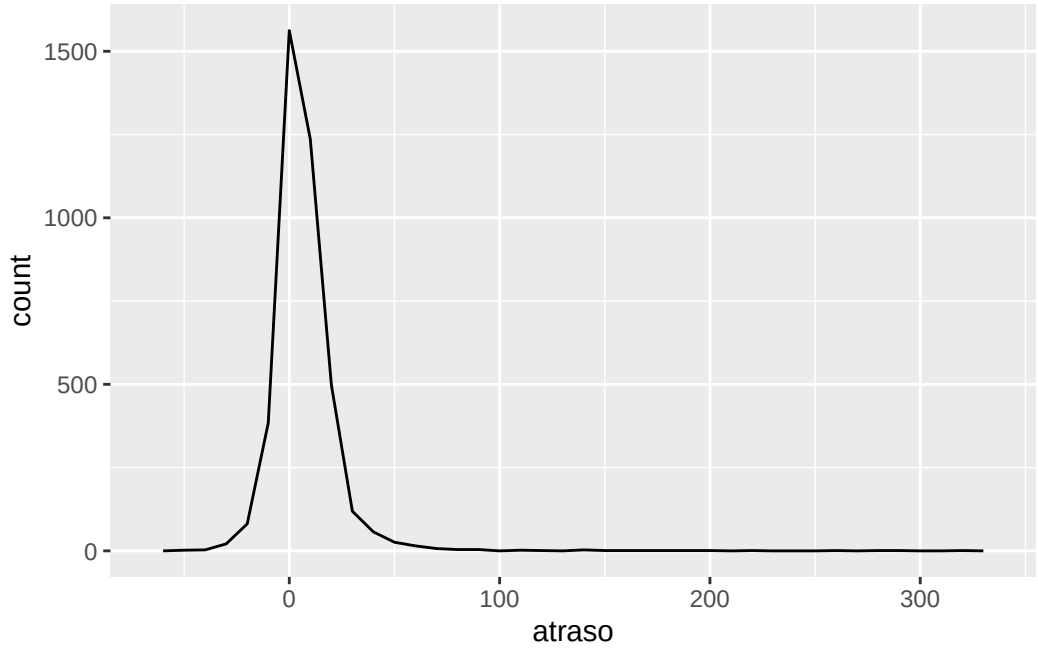
```
`summarise()` has grouped output by 'anio', 'mes'. You can override using the
`.groups` argument.
```

```
# A tibble: 365 x 4
# Groups:   anio, mes [12]
    anio   mes   dia  mean
   <int> <int> <int> <dbl>
 1  2013     1     1 11.4
 2  2013     1     2 13.7
 3  2013     1     3 10.9
 4  2013     1     4  8.97
 5  2013     1     5  5.73
 6  2013     1     6  7.15
 7  2013     1     7  5.42
 8  2013     1     8  2.56
 9  2013     1     9  2.30
10  2013     1    10  2.84
# i 355 more rows
```

Conteos

Siempre que realices una agregación, conviene incluir un conteo con `n()` o un recuento de valores no faltantes con `sum(!is.na(x))`. Esto permite asegurarte de no basar conclusiones en pocos datos. Por ejemplo, se puede usar para identificar los aviones (por número de cola) con mayores demoras promedio.

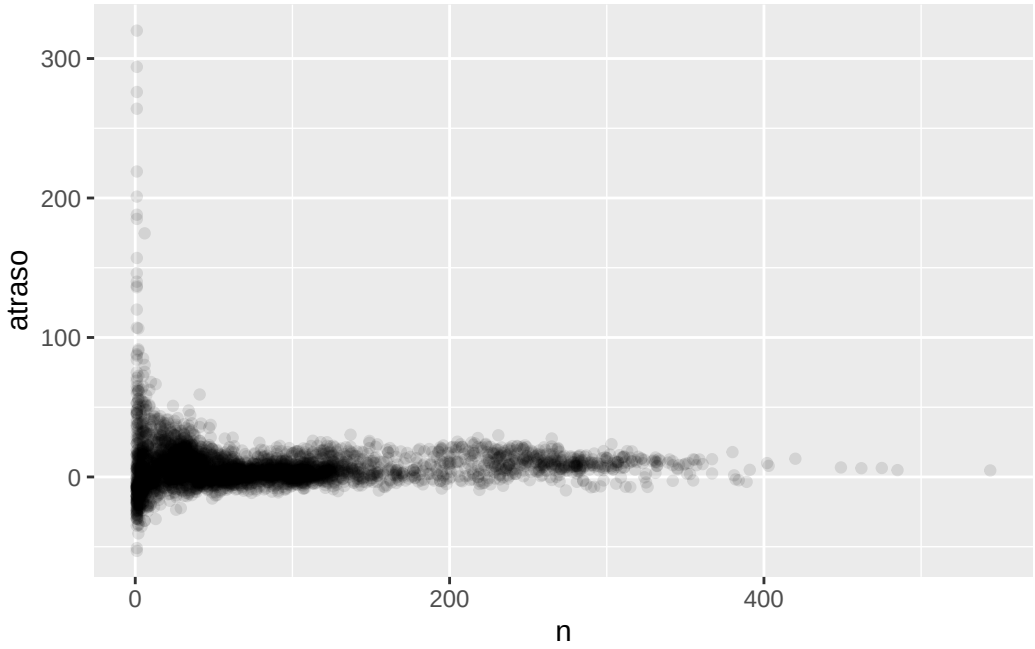
```r
atrasos <- no_cancelados %>%
  group_by(codigo_cola) %>%
  summarise(
   atraso = mean(atraso_llegada)
  )

ggplot(data = atrasos, mapping = aes(x = atraso)) +
  geom_freqpoly(binwidth = 10)
```



¡Hay algunos aviones que tienen una demora promedio de 5 horas (300 minutos)!

La historia es en realidad un poco más matizada. Podemos obtener más información si hacemos un diagrama de dispersión del número de vuelos contra la demora promedio:

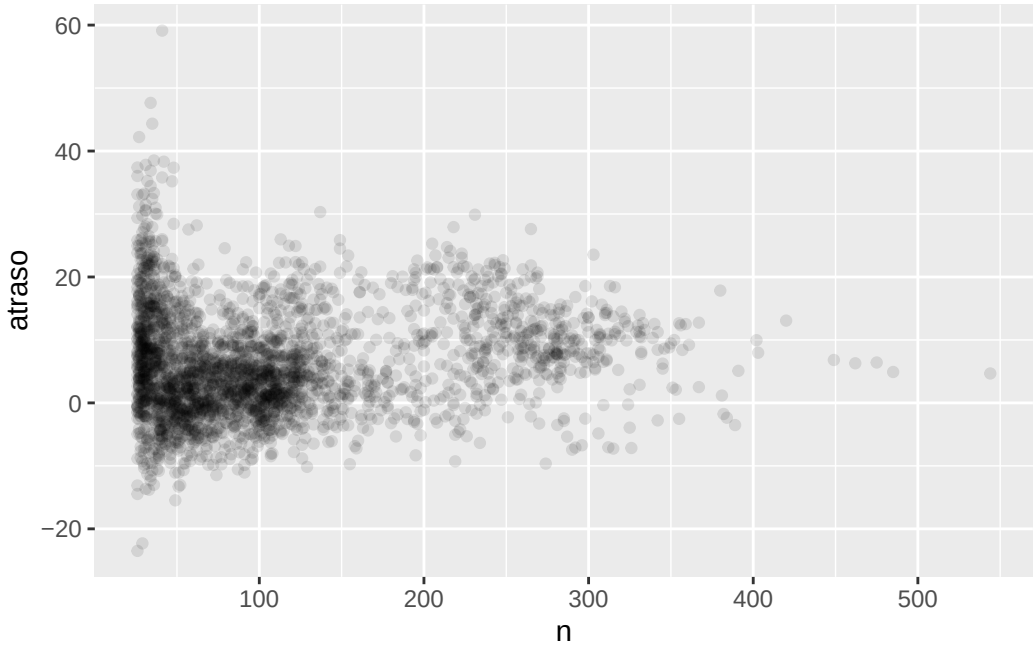
```
atrasos <- no_cancelados %>%
  group_by(codigo_cola) %>%
  summarise(
    atraso = mean(atraso_llegada),
    n = n()
  )

ggplot(data = atrasos, mapping = aes(x = n, y = atraso)) +
  geom_point(alpha = 1/10)
```



No es sorprendente que haya mayor variación en el promedio de retraso cuando hay pocos vuelos. Este tipo de gráfico tiene una forma característica: la variación disminuye a medida que aumenta el tamaño de muestra.

En estos casos, es útil eliminar los grupos con pocas observaciones para observar mejor el patrón general y reducir el efecto de la variación extrema. El siguiente bloque de código hace eso y muestra cómo integrar `ggplot2` en el flujo de trabajo de `dplyr`. Aunque cambiar de `%>%` a `+` puede parecer incómodo al principio, resulta bastante conveniente una vez que se entiende.

```
atrasos %>%
  filter(n > 25) %>%
  ggplot(mapping = aes(x = n, y = atraso)) +
    geom_point(alpha = 1/10)
```



RStudio tip: el atajo **Cmd/Ctrl + Shift + P** reenvía el bloque previamente enviado desde el editor a la consola. Es útil, por ejemplo, al explorar el valor de `n`. Envías el bloque completo con **Cmd/Ctrl + Enter**, modificas `n`, y luego lo reenvías con **Cmd/Ctrl + Shift + P**.

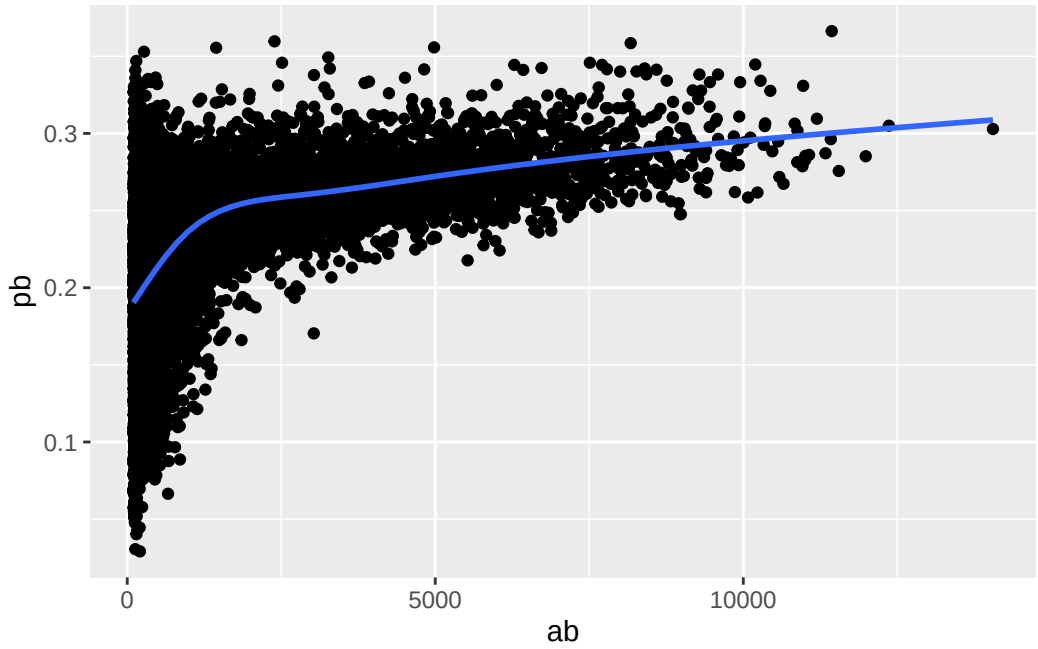
Una variación común de este patrón se observa al analizar el rendimiento promedio de los bateadores en béisbol. Usando el conjunto de datos de bateadores, se calcula el promedio de bateo (bateos / intentos) para cada jugador de las Grandes Ligas.

Al graficar el promedio de bateo (`pb`) frente al número de turnos al bate (`ab`), se observan dos patrones.

1. Como en el ejemplo anterior, la variación en el estadístico de resumen disminuye a medida que aumenta el número de observaciones.
2. Existe una correlación positiva entre la habilidad (`pb`) y las oportunidades para golpear la pelota (`ab`), porque los equipos deciden quién juega y eligen a sus mejores jugadores.

```
# Convierte a tibble para puedas imprimirlo de una manera legible
bateo <- as_tibble(datos::bateadores)

rendimiento_bateadores <- bateo %>%
  group_by(id_jugador) %>%
  summarise(
    pb = sum(golpes, na.rm = TRUE) / sum(al_bate, na.rm = TRUE),
    ab = sum(al_bate, na.rm = TRUE)
```

```
  )

rendimiento_bateadores %>%
  filter(ab > 100) %>%
  ggplot(mapping = aes(x = ab, y = pb)) +
  geom_point() +
  geom_smooth(se = FALSE)
```



Esto tiene implicaciones importantes para la clasificación. Si se ordena ingenuamente con `desc(pb)`, las personas con los mejores promedios de bateo pueden parecer destacadas por suerte, pero no necesariamente por habilidad.

```
rendimiento_bateadores %>%
  arrange(desc(pb))
```

```
# A tibble: 20,730 x 3
   id_jugador    pb    ab
   <chr>      <dbl> <int>
 1 abramge01      1     1
 2 alberan01      1     1
 3 banisje01      1     1
 4 bartocl01      1     1
```

```
 5 bassdo01       1     1
 6 birasst01      1     2
 7 bruneju01      1     1
 8 burnscb01      1     1
 9 cammaer01      1     1
10 campsh01       1     1
# i 20,720 more rows
```

Puedes encontrar una buena explicación de este problema en este artículo sobre béisbol y este otro sobre cómo no ordenar por promedio.

Funciones de resumen útiles

Solo el uso de medias, conteos y sumas puede ser útil, pero R ofrece muchas otras funciones de resumen valiosas.

- Medidas de centralidad: además de `mean(x)`, `median(x)` también es muy útil. La media es la suma dividida por el número de observaciones; la mediana es el valor que divide a `x` en dos mitades iguales. A veces conviene combinar la agregación con un subconjunto lógico, tema que se abordará más adelante en [selección de subconjuntos].

```
no_cancelados %>%
  group_by(anio, mes, dia) %>%
  summarise(
    prom_atraso1 = mean(atraso_llegada),
    prom_atraso2 = mean(atraso_llegada[atraso_llegada > 0]) # el promedio de atrasos positivo
  )
```

```
`summarise()` has grouped output by 'anio', 'mes'. You can override using the
`.groups` argument.
```

```
# A tibble: 365 x 5
# Groups:   anio, mes [12]
    anio   mes   dia prom_atraso1 prom_atraso2
   <int> <int> <int>        <dbl>        <dbl>
 1  2013     1     1       12.7           32.5
 2  2013     1     2       12.7           32.0
 3  2013     1     3        5.73          27.7
 4  2013     1     4       -1.93          28.3
 5  2013     1     5       -1.53          22.6
 6  2013     1     6        4.24          24.4
```

```
 7  2013     1     7       -4.95          27.8
 8  2013     1     8       -3.23          20.8
 9  2013     1     9       -0.264         25.6
10  2013     1    10       -5.90          27.3
# i 355 more rows
```

- Medidas de dispersión: `sd(x)`, `IQR(x)`, `mad(x)`. La desviación estándar `sd(x)` mide la dispersión típica. `IQR(x)` (rango intercuartil) y `mad(x)` (desviación media absoluta) son alternativas robustas, más útiles cuando hay valores atípicos.

```r
# ¿Por qué la distancia a algunos destinos es más variable que la de otros?
no_cancelados %>%
  group_by(destino) %>%
  summarise(distancia_sd = sd(distancia)) %>%
  arrange(desc(distancia_sd))
```

```
# A tibble: 104 x 2
   destino distancia_sd
   <chr>          <dbl>
 1 EGE            10.5
 2 SAN            10.4
 3 SFO            10.2
 4 HNL            10.0
 5 SEA             9.98
 6 LAS             9.91
 7 PDX             9.87
 8 PHX             9.86
 9 LAX             9.66
10 IND             9.46
# i 94 more rows
```

- Medidas de rango: `min(x)`, `quantile(x, 0.25)`, `max(x)`. Los cuantiles generalizan la mediana; por ejemplo, `quantile(x, 0.25)` da un valor mayor que el 25% de los datos y menor que el 75% restante.

```r
# ¿Cuándo salen los primeros y los últimos vuelos cada día?
no_cancelados %>%
  group_by(anio, mes, dia) %>%
  summarise(
    primero = min(horario_salida),
    ultimo = max(horario_salida)
  )
```

```
`summarise()` has grouped output by 'anio', 'mes'. You can override using the
`.groups` argument.
```

```
# A tibble: 365 x 5
# Groups:   anio, mes [12]
    anio   mes   dia primero ultimo
   <int> <int> <int>   <int>  <int>
 1  2013     1     1     517   2356
 2  2013     1     2      42   2354
 3  2013     1     3      32   2349
 4  2013     1     4      25   2358
 5  2013     1     5      14   2357
 6  2013     1     6      16   2355
 7  2013     1     7      49   2359
 8  2013     1     8     454   2351
 9  2013     1     9       2   2252
10  2013     1    10       3   2320
# i 355 more rows
```

Medidas de posición: `first(x)`, `nth(x, 2)`, `last(x)`. Funcionan como `x[1]`, `x[2]` y `x[length(x)]`, pero permiten definir un valor predeterminado si la posición no existe. Por ejemplo, se pueden usar para encontrar la primera y última salida de cada día.

```
no_cancelados %>%
  group_by(anio, mes, dia) %>%
  summarise(
    primera_salida = first(horario_salida),
    ultima_salida = last(horario_salida)
  )
```

```
`summarise()` has grouped output by 'anio', 'mes'. You can override using the
`.groups` argument.
```

```
# A tibble: 365 x 5
# Groups:   anio, mes [12]
    anio   mes   dia primera_salida ultima_salida
   <int> <int> <int>          <int>         <int>
 1  2013     1     1            517          2356
 2  2013     1     2             42          2354
 3  2013     1     3             32          2349
 4  2013     1     4             25          2358
```

```
 5  2013     1     5             14          2357
 6  2013     1     6             16          2355
 7  2013     1     7             49          2359
 8  2013     1     8            454          2351
 9  2013     1     9              2          2252
10  2013     1    10              3          2320
# i 355 more rows
```

Estas funciones son complementarias al filtrado en rangos. El filtrado te proporciona todas las variables, con cada observación en una fila distinta:

```
no_cancelados %>%
  group_by(anio, mes, dia) %>%
  mutate(r = min_rank(desc(horario_salida))) %>%
  filter(r %in% range(r))
```

```
# A tibble: 770 x 20
# Groups:   anio, mes, dia [365]
    anio   mes   dia horario_salida salida_programada atraso_salida
   <int> <int> <int>          <int>             <int>         <dbl>
 1  2013     1     1            517               515             2
 2  2013     1     1           2356              2359            -3
 3  2013     1     2             42              2359            43
 4  2013     1     2           2354              2359            -5
 5  2013     1     3             32              2359            33
 6  2013     1     3           2349              2359           -10
 7  2013     1     4             25              2359            26
 8  2013     1     4           2358              2359            -1
 9  2013     1     4           2358              2359            -1
10  2013     1     5             14              2359            15
# i 760 more rows
# i 14 more variables: horario_llegada <int>, llegada_programada <int>,
#   atraso_llegada <dbl>, aerolinea <chr>, vuelo <int>, codigo_cola <chr>,
#   origen <chr>, destino <chr>, tiempo_vuelo <dbl>, distancia <dbl>,
#   hora <dbl>, minuto <dbl>, fecha_hora <dttm>, r <int>
```

- Conteos: `n()` devuelve el tamaño del grupo actual y no requiere argumentos. Para contar valores no faltantes, se usa `sum(!is.na(x))`. Para contar valores distintos, se usa `n_distinct(x)`.

```r
# ¿Qué destinos tienen la mayoría de las aerolíneas?
no_cancelados %>%
  group_by(destino) %>%
  summarise(aerolineas = n_distinct(aerolinea)) %>%
  arrange(desc(aerolineas))
```

```
# A tibble: 104 x 2
   destino aerolineas
   <chr>        <int>
 1 ATL              7
 2 BOS              7
 3 CLT              7
 4 ORD              7
 5 TPA              7
 6 AUS              6
 7 DCA              6
 8 DTW              6
 9 IAD              6
10 MSP              6
# i 94 more rows
```

Los conteos son tan útiles que dplyr proporciona un ayudante simple si todo lo que quieres es un conteo:

```r
no_cancelados %>%
  count(destino)
```

```
# A tibble: 104 x 2
   destino     n
   <chr>   <int>
 1 ABQ       254
 2 ACK       264
 3 ALB       418
 4 ANC         8
 5 ATL     16837
 6 AUS      2411
 7 AVL       261
 8 BDL       412
 9 BGR       358
10 BHM       269
# i 94 more rows
```

Opcionalmente puedes proporcionar una variable de ponderación. Por ejemplo, podrías usar esto para "contar" (sumar) el número total de millas que voló un avión:

```
no_cancelados %>%
  count(codigo_cola, wt = distancia)
```

```
# A tibble: 4,037 x 2
   codigo_cola      n
   <chr>        <dbl>
 1 D942DN        3418
 2 N0EGMQ      239143
 3 N10156      109664
 4 N102UW       25722
 5 N103US       24619
 6 N104UW       24616
 7 N10575      139903
 8 N105UW       23618
 9 N107US       21677
10 N108UW       32070
# i 4,027 more rows
```

- Conteos y proporciones de valores lógicos: `sum(x > 10)`, `mean(y == 0)`. En operaciones numéricas, `TRUE` se convierte en 1 y `FALSE` en 0, por lo que `sum(x)` cuenta los `TRUE` y `mean(x)` da la proporción de ellos.

```
# ¿Cuántos vuelos salieron antes de las 5 am?
# (estos generalmente son vuelos demorados del día anterior)
no_cancelados %>%
  group_by(anio, mes, dia) %>%
  summarise(n_temprano = sum(horario_salida < 500))
```

```
`summarise()` has grouped output by 'anio', 'mes'. You can override using the
`.groups` argument.
```

```
# A tibble: 365 x 4
# Groups:   anio, mes [12]
    anio   mes   dia n_temprano
   <int> <int> <int>      <int>
 1  2013     1     1          0
 2  2013     1     2          3
 3  2013     1     3          4
```

```
 4  2013     1     4          3
 5  2013     1     5          3
 6  2013     1     6          2
 7  2013     1     7          2
 8  2013     1     8          1
 9  2013     1     9          3
10  2013     1    10          3
# i 355 more rows
```

```
# ¿Qué proporción de vuelos se retrasan más de una hora?
no_cancelados %>%
  group_by(anio, mes, dia) %>%
  summarise(hora_prop = mean(atraso_llegada > 60))
```

```
`summarise()` has grouped output by 'anio', 'mes'. You can override using the
`.groups` argument.
```

```
# A tibble: 365 x 4
# Groups:   anio, mes [12]
    anio   mes   dia hora_prop
   <int> <int> <int>     <dbl>
 1  2013     1     1    0.0722
 2  2013     1     2    0.0851
 3  2013     1     3    0.0567
 4  2013     1     4    0.0396
 5  2013     1     5    0.0349
 6  2013     1     6    0.0470
 7  2013     1     7    0.0333
 8  2013     1     8    0.0213
 9  2013     1     9    0.0202
10  2013     1    10    0.0183
# i 355 more rows
```

Agrupación por múltiples variables

Cuando agrupas por múltiples variables, cada resumen se desprende de un nivel de la agrupación. Eso hace que sea más fácil acumular progresivamente en un conjunto de datos:

```
diario <- group_by(vuelos, anio, mes, dia)
(por_dia   <- summarise(diario, vuelos = n()))
```

```
`summarise()` has grouped output by 'anio', 'mes'. You can override using the
`.groups` argument.

# A tibble: 365 x 4
# Groups:   anio, mes [12]
    anio   mes   dia vuelos
   <int> <int> <int>  <int>
 1  2013     1     1    842
 2  2013     1     2    943
 3  2013     1     3    914
 4  2013     1     4    915
 5  2013     1     5    720
 6  2013     1     6    832
 7  2013     1     7    933
 8  2013     1     8    899
 9  2013     1     9    902
10  2013     1    10    932
# i 355 more rows
```

```
(por_mes <- summarise(por_dia, vuelos = sum(vuelos)))
```

```
`summarise()` has grouped output by 'anio'. You can override using the
`.groups` argument.

# A tibble: 12 x 3
# Groups:   anio [1]
    anio   mes vuelos
   <int> <int>  <int>
 1  2013     1  27004
 2  2013     2  24951
 3  2013     3  28834
 4  2013     4  28330
 5  2013     5  28796
 6  2013     6  28243
 7  2013     7  29425
 8  2013     8  29327
 9  2013     9  27574
10  2013    10  28889
11  2013    11  27268
12  2013    12  28135
```

```r
(por_anio  <- summarise(por_mes, vuelos = sum(vuelos)))
```

```
# A tibble: 1 x 2
   anio vuelos
  <int>  <int>
1  2013 336776
```

Ten cuidado al acumular resúmenes progresivamente: funciona bien con sumas y recuentos, pero con medias y varianzas se debe considerar la ponderación, y no es posible hacerlo exactamente con estadísticas basadas en rangos como la mediana. Es decir, la suma de sumas agrupadas da la suma total, pero la mediana de medianas agrupadas no es la mediana general.

Desagrupar

Si necesitas eliminar la agrupación y regresar a las operaciones en datos desagrupados, usa `ungroup()`.

```r
diario %>%
  ungroup() %>%             # ya no está agrupado por fecha
  summarise(vuelos = n())   # todos los vuelos
```

```
# A tibble: 1 x 1
  vuelos
   <int>
1 336776
```

Ejercicios

1. Haz una lluvia de ideas de al menos 5 formas diferentes de evaluar las características de un retraso típico de un grupo de vuelos. Considera los siguientes escenarios:

- Un vuelo llega 15 minutos antes 50% del tiempo, y 15 minutos tarde 50% del tiempo.
- Un vuelo llega siempre 10 minutos tarde.
- Un vuelo llega 30 minutos antes 50% del tiempo, y 30 minutos tarde 50% del tiempo.
- Un vuelo llega a tiempo en el 99% de los casos. 1% de las veces llega 2 horas tarde.

¿Qué es más importante: retraso de la llegada o demora de salida?

2. Sugiere un nuevo enfoque que te dé el mismo output que `no_cancelados %>% count(destino)` y `no_cancelado %>% count(codigo_cola, wt = distancia)` (sin usar `count()`).

3. Nuestra definición de vuelos cancelados (`is.na(atraso_salida) | is.na (atraso_llegada)`) es un poco subóptima. ¿Por qué? ¿Cuál es la columna más importante?

4. Mira la cantidad de vuelos cancelados por día. ¿Hay un patrón? ¿La proporción de vuelos cancelados está relacionada con el retraso promedio?

5. ¿Qué compañía tiene los peores retrasos? Desafío: ¿puedes desenredar el efecto de malos aeropuertos vs. el efecto de malas aerolíneas? ¿Por qué o por qué no? (Sugerencia: piensa en `vuelos %>% group_by(aerolinea, destino) %>% summarise(n())`)

¿Qué hace el argumento `sort` a `count()`. ¿Cuándo podrías usarlo?

Transformaciones agrupadas (y filtros)

La agrupación es más útil cuando se combina con `summarise()`, pero también permite realizar operaciones útiles con `mutate()` y `filter()`.

- Encuentra los peores miembros de cada grupo:

```
vuelos_sml %>%
  group_by(anio, mes, dia) %>%
  filter(rank(desc(atraso_llegada)) < 10)
```

```
# A tibble: 3,306 x 7
# Groups:   anio, mes, dia [365]
    anio   mes   dia atraso_salida atraso_llegada distancia tiempo_vuelo
   <int> <int> <int>         <dbl>          <dbl>     <dbl>        <dbl>
 1  2013     1     1           853            851       184           41
 2  2013     1     1           290            338      1134          213
 3  2013     1     1           260            263       266           46
 4  2013     1     1           157            174       213           60
 5  2013     1     1           216            222       708          121
 6  2013     1     1           255            250       589          115
 7  2013     1     1           285            246      1085          146
 8  2013     1     1           192            191       199           44
 9  2013     1     1           379            456      1092          222
10  2013     1     2           224            207       550           94
# i 3,296 more rows
```

- Encuentra todos los grupos más grandes que un determinado umbral:

```
destinos_populares <- vuelos %>%
  group_by(destino) %>%
  filter(n() > 365)
destinos_populares
```

```
# A tibble: 332,577 x 19
# Groups:   destino [77]
    anio   mes   dia horario_salida salida_programada atraso_salida
   <int> <int> <int>          <int>              <int>         <dbl>
 1  2013     1     1            517                515             2
 2  2013     1     1            533                529             4
 3  2013     1     1            542                540             2
 4  2013     1     1            544                545            -1
 5  2013     1     1            554                600            -6
 6  2013     1     1            554                558            -4
 7  2013     1     1            555                600            -5
 8  2013     1     1            557                600            -3
 9  2013     1     1            557                600            -3
10  2013     1     1            558                600            -2
# i 332,567 more rows
# i 13 more variables: horario_llegada <int>, llegada_programada <int>,
#   atraso_llegada <dbl>, aerolinea <chr>, vuelo <int>, codigo_cola <chr>,
#   origen <chr>, destino <chr>, tiempo_vuelo <dbl>, distancia <dbl>,
#   hora <dbl>, minuto <dbl>, fecha_hora <dttm>
```

- Estandariza para calcular las métricas por grupo:

```
destinos_populares %>%
  filter(atraso_llegada > 0) %>%
  mutate(prop_atraso = atraso_llegada / sum(atraso_llegada)) %>%
  select(anio:dia, destino, atraso_llegada, prop_atraso)
```

```
# A tibble: 131,106 x 6
# Groups:   destino [77]
    anio   mes   dia destino atraso_llegada prop_atraso
   <int> <int> <int> <chr>            <dbl>       <dbl>
 1  2013     1     1 IAH                 11   0.000111
 2  2013     1     1 IAH                 20   0.000201
 3  2013     1     1 MIA                 33   0.000235
```

```
 4  2013     1     1 ORD                12   0.0000424
 5  2013     1     1 FLL                19   0.0000938
 6  2013     1     1 ORD                 8   0.0000283
 7  2013     1     1 LAX                 7   0.0000344
 8  2013     1     1 DFW                31   0.000282
 9  2013     1     1 ATL                12   0.0000400
10  2013     1     1 DTW                16   0.000116
# i 131,096 more rows
```

Un filtro agrupado es una transformación agrupada seguida de un filtro desagrupado. Aunque puede ser útil para manipulaciones rápidas, generalmente se prefiere evitarlo porque dificulta verificar los resultados.

Las funciones que se aplican naturalmente en transformaciones agrupadas y filtros se llaman **funciones de ventana** (*window functions*), a diferencia de las funciones de resumen. Puedes aprender más sobre ellas en la viñeta: `vignette("window-functions")`.

Ejercicios

1. Remítete a las listas de funciones útiles de mutación y filtrado. Describe cómo cambia cada operación cuando las combinas con la agrupación.
2. ¿Qué avión (`codigo_cola`) tiene el peor registro de tiempo?
3. ¿A qué hora del día deberías volar si quieres evitar lo más posible los retrasos?
4. Para cada destino, calcula los minutos totales de demora. Para cada vuelo, calcula la proporción de la demora total para su destino.
5. Los retrasos suelen estar temporalmente correlacionados: incluso una vez que el problema que causó el retraso inicial se ha resuelto, los vuelos posteriores se retrasan para permitir que salgan los vuelos anteriores. Usando `lag()`, explora cómo el retraso de un vuelo está relacionado con el retraso del vuelo inmediatamente anterior.
6. Mira cada destino. ¿Puedes encontrar vuelos sospechosamente rápidos? (es decir, vuelos que representan un posible error de entrada de datos). Calcula el tiempo en el aire de un vuelo relativo al vuelo más corto a ese destino. ¿Cuáles vuelos se retrasaron más en el aire?
7. Encuentra todos los destinos que son volados por al menos dos operadores. Usa esta información para clasificar a las aerolíneas.
8. Para cada avión, cuenta el número de vuelos antes del primer retraso de más de 1 hora.