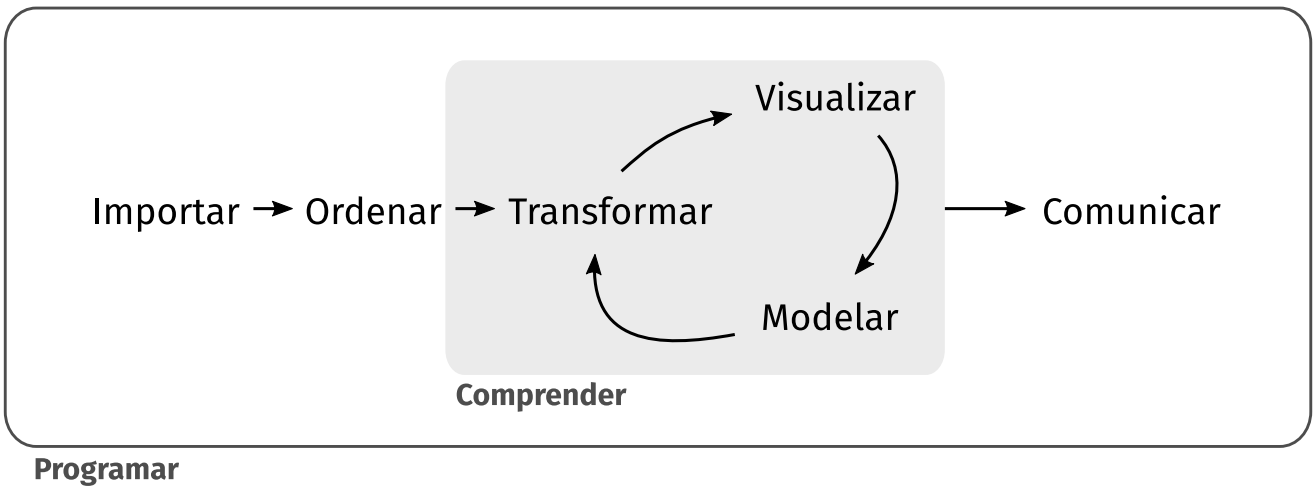
R PARA CIENCIA DE DATOS



HADLEY WICKHAM
GARETT GROLEMUND

R PARA CIENCIA DE DATOS

HADLEY WICKHAM & GARETT GROLEMUND

R Para Ciencia de Datos

HADLEY WICKHAM & GARETT GROLEMUND

Este libro se distribuye de forma gratuita en https://github.com/cienciadedatos/datos

La presente versión fue compilada el día 14 de junio de 2020.

Índice general

1 Introducción **3**
- 1.1 Lo que aprenderás . . . 3
- 1.2 Cómo está organizado este libro . . . 5
- 1.3 Qué no vas a aprender . . . 5
  - 1.3.1 Big data . . . 5
  - 1.3.2 Python, Julia y amigos . . . 6
  - 1.3.3 Datos no rectangulares . . . 6
  - 1.3.4 Confirmación de hipótesis . . . 7
- 1.4 Prerrequisitos . . . 7
  - 1.4.1 R . . . 7
  - 1.4.2 RStudio . . . 8
  - 1.4.3 El Tidyverse . . . 9
  - 1.4.4 El paquete `datos` . . . 9
  - 1.4.5 Otros paquetes . . . 9
- 1.5 Ejecutar código en R . . . 10
- 1.6 Pedir ayuda y aprender más . . . 10
- 1.7 Agradecimientos . . . 12
- 1.8 Colofón . . . 13

(PART) Explorar **16**

2 Introducción **17**

3 Visualización de datos **19**
- 3.1 Introducción . . . 19
  - 3.1.1 Prerrequisitos . . . 19
- 3.2 Primeros pasos . . . 20
  - 3.2.1 El *data frame* `millas` . . . 20
  - 3.2.2 Creando un gráfico con ggplot . . . 21
  - 3.2.3 Una plantilla de gráficos . . . 22
  - 3.2.4 Ejercicios . . . 22
- 3.3 Mapeos estéticos . . . 22
  - 3.3.1 Ejercicios . . . 26
- 3.4 Problemas comunes . . . 28
- 3.5 Separar en facetas . . . 28
  - 3.5.1 Ejercicios . . . 30
- 3.6 Objetos geométricos . . . 30
  - 3.6.1 Ejercicios . . . 36

3.7 Transformaciones estadísticas . . . 37
3.7.1 Ejercicios . . . 41
3.8 Ajustes de posición . . . 41
3.8.1 Ejercicios . . . 45
3.9 Sistemas de coordenadas . . . 46
3.9.1 Ejercicios . . . 47
3.10 La gramática de gráficos en capas . . . 48

4 Flujo de trabajo: conocimientos básicos **51**
4.1 Conocimientos básicos de programación . . . 51
4.2 La importancia de los nombres . . . 52
4.3 Usando funciones . . . 53
4.4 Ejercicios . . . 54

5 Transformación de datos **56**
5.1 Introducción . . . 56
5.1.1 Prerequisitos . . . 56
5.1.2 vuelos . . . 56
5.1.3 Lo básico de **dplyr** . . . 58
5.2 Filtrar filas con `filter()` . . . 58
5.2.1 Comparaciones . . . 59
5.2.2 Operadores lógicos . . . 60
5.2.3 Valores faltantes . . . 61
5.2.4 Ejercicios . . . 62
5.3 Reordenar las filas con `arrange()` . . . 63
5.3.1 Ejercicios . . . 64
5.4 Seleccionar columnas con `select()` . . . 64
5.4.1 Ejercicios . . . 66
5.5 Añadir nuevas variables con `mutate()` . . . 67
5.5.1 Funciones de creación útiles . . . 68
5.5.2 Ejercicios . . . 70
5.6 Resúmenes agrupados con `summarise()` . . . 71
5.6.1 Combinación de múltiples operaciones con el *pipe* . . . 71
5.6.2 Valores faltantes . . . 73
5.6.3 Conteos . . . 74
5.6.4 Funciones de resumen útiles . . . 78
5.6.5 Agrupación por múltiples variables . . . 83
5.6.6 Desagrupar . . . 84
5.6.7 Ejercicios . . . 84
5.7 Transformaciones agrupadas (y filtros) . . . 85
5.7.1 Ejercicios . . . 86

Bienvenida

Este es el sitio web de la versión en español de **"R for Data Science"**, de Hadley Wickham y Garrett Grolemund. Este texto te enseñará cómo hacer ciencia de datos con R: aprenderás a importar datos, llevarlos a la estructura más conveniente, transformarlos, visualizarlos y modelarlos. Así podrás poner en práctica las habilidades necesarias para hacer ciencia de datos. Tal como los químicos aprenden a limpiar tubos de ensayo y ordenar un laboratorio, aprenderás a limpiar datos y crear gráficos— junto a muchas otras habilidades que permiten que la ciencia de datos tenga lugar. En este libro encontrarás las mejores prácticas para desarrollar dichas tareas usando R. También aprenderás a usar la gramática de gráficos, programación letrada e investigación reproducible para ahorrar tiempo. Además, aprenderás a manejar recursos cognitivos para facilitar el hacer descubrimientos al momento de manipular, visualizar y explorar datos.

Sobre la traducción

La traducción de "R para Ciencia de Datos" es un proyecto colaborativo de la comunidad de R de Latinoamérica, que tiene por objetivo hacer R más accesible en la región. .

En la traducción del libro participaron las siguientes personas (en orden alfabético): Marcela Alfaro, Mónica Alonso, Fernando Álvarez, Zulemma Bazurto, Yanina Bellini, Juliana Benítez, María Paula Caldas, Elio Campitelli, Florencia D'Andrea, Rocío Espada, Joshua Kunst, Patricia Loto, Pamela Matías, Lina Moreno, Paola Prieto, Riva Quiroga, Lucía Rodríguez, Mauricio "Pachá" Vargas, Daniela Vázquez, Melina Vidoni, Roxana N. Villafañe. ¡Muchas gracias por su trabajo! La administración del repositorio con la traducción ha estado cargo de Mauricio "Pachá" Vargas. La coordinación general y la edición, a cargo de Riva Quiroga.

Agradecemos a todas las personas que han ayudado revisando las traducciones y haciendo sugerencias de mejora. Puedes revisar la documentación del proyecto para ver los créditos de participación. Gracias también a Marcela Alfaro por el tuit que hizo visible la necesidad de la versión en español, y a Laura Ación y Edgar Ruiz, que pusieron en contacto a las personas del equipo.

Este proyecto no solo implica la traducción del texto, sino también de los sets de datos que se utilizan a lo largo de él. Para ello, se creó el paquete `datos`, que contiene las versiones traducidas de estos. Puedes revisar su documentación acá. El paquete fue desarrollado por Edgar Ruiz, Riva Quiroga, Mauricio "Pachá" Vargas y Mauro Lepore. Para su creación se utilizaron funciones del paquete `datalang` de Edgar Ruiz y las sabias sugerencias de Hadley Wickham.
Si quieres conocer más sobre los principios que han orientado nuestro trabajo puedes leer [la docu-

mentación del proyecto] (https://github.com/cienciadedatos/documentacion-traduccion-r4ds). Para estar al tanto de novedades sobre el paquete {datos} y nuevas iniciativas del equipo, sigue nuestra cuenta en Twitter.

Sobre la versión original en inglés

Puedes consultar la versión original del libro en r4ds.had.co.nz/. Existe una edición impresa, que fue publicada por O'Reilly en enero de 2017. Puedes adquirir una copia en Amazon.

(El libro "R for Data Science" primero se llamó "Data Science with R" en "Hands-On Programming with R")

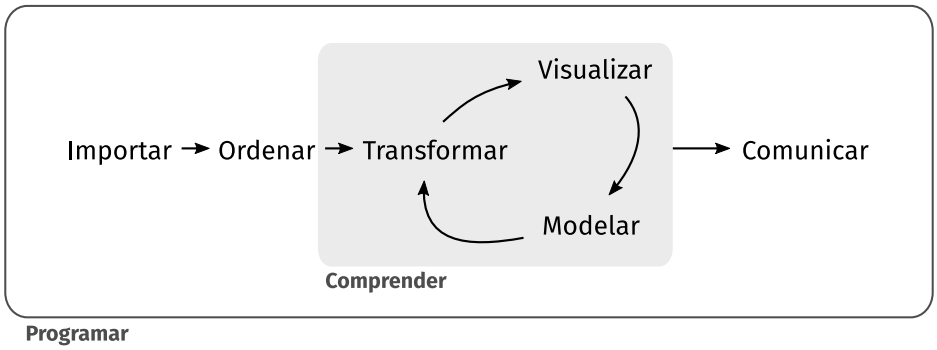
CAPÍTULO 1

Introducción

La ciencia de datos (*data science*) es una disciplina fascinante que te permite convertir datos sin procesar en entendimiento, comprensión y conocimiento. El objetivo de este libro es ayudarte a aprender las herramientas más importantes para que puedas hacer ciencia de datos en R. Luego de leerlo, tendrás las herramientas para enfrentar una gran variedad de desafíos propios de esta área, usando las mejores partes de R.

1.1. Lo que aprenderás

La ciencia de datos es un campo muy amplio y no hay manera de que puedas dominarlo leyendo un solo libro. El objetivo de este, en particular, es entregarte una base sólida acerca de las herramientas más importantes. Nuestro modelo sobre cuáles son las herramientas necesarias para un proyecto típico de ciencia de datos se ve así:



Primero, debes **importar** tus datos hacia R. Típicamente, esto implica tomar datos que están guardados en un archivo, base de datos o API y cargarlos como *data frame* en R. Si no puedes llevar tus datos a R, no puedes hacer ciencia de datos con él.

Una vez que has importado los datos, es una buena idea **ordenarlos**. Ordenar los datos significa guar-

darlos de una manera consistente que haga coincidir la semántica del set de datos con la manera en que está guardado. En definitiva, cuando tus datos están ordenados, cada columna es una variable y cada fila una observación. Tener datos ordenados es importante porque si su estructura es consistente, puedes enfocar tus esfuerzos en las preguntas sobre los datos y no en luchar para que estos tengan la forma necesaria para diferentes funciones.

Cuando tus datos están ordenados, un primer paso suele ser **transformarlos**. La transformación implica reducir las observaciones a aquellas que sean de interés (como todas las personas de una ciudad o todos los datos del último año), crear nuevas variables que sean funciones de variables ya existentes (como calcular la rapidez a partir de la velocidad y el tiempo) y calcular una serie de estadísticos de resumen (como recuentos y medias). Juntos, a ordenar y transformar, se les llama **manejar o domar** los datos, porque hacer que estos tengan la forma con la que es natural trabajarlos, suele sentirse como una lucha.

Una vez que tienes los datos ordenados con las variables que necesitas, hay dos principales fuentes generadoras de conocimiento: la visualización y el modelado. Ambas tienen fortalezas y debilidades complementarias, por lo que cualquier análisis real iterará entre ellas varias veces.

La **visualización** es una actividad humana fundamental. Una buena visualización te mostrará cosas que no esperabas o hará surgir nuevas preguntas acerca de los datos. También puede darte pistas acerca de si estás haciendo las preguntas equivocadas o si necesitas recolectar datos diferentes. Las visualizaciones pueden sorprenderte, pero no escalan particularmente bien, ya que requieren ser interpretadas por una persona.

Los **modelos** son herramientas complementarias a la visualización. Una vez que tus preguntas son lo suficientemente precisas, puedes utilizar un modelo para responderlas. Los modelos son herramientas matemáticas o computacionales, por lo que generalmente escalan bien. Incluso cuando no lo hacen, resulta más económico comprar más computadores que comprar más cerebros. Sin embargo, cada modelo tiene supuestos y, debido a su propia naturaleza, un modelo no puede cuestionar sus propios supuestos. Esto significa que un modelo, por definición, no puede sorprenderte.

El último paso de la ciencia de datos es la **comunicación**, una parte crítica de cualquier proyecto de análisis de datos. No importa qué tan bien tus modelos y visualizaciones te hayan permitido entender tus datos, a menos que también puedas comunicar esos resultados a otras personas.

Alrededor de todas estas herramientas se encuentra la **programación**. La programación es una herramienta transversal que usarás en todas las partes de tu proyecto. No necesitas ser una personas experta en programación para hacer ciencia de datos, pero aprender más sobre ella es una gran ventaja porque te permite automatizar tareas recurrentes y resolver problemas con mayor facilidad.

En cualquier proyecto de ciencia de datos tendrás que ocupar estas herramientas, pero en muchos casos estas no serán suficientes. Hay un regla aproximada de 80-20 en juego: puedes enfrentar alrededor del 80 % de cualquier proyecto usando las herramientas que aprenderás en este libro, pero necesitarás utilizar otras para abordar el 20 % restante. A lo largo del libro te iremos señalando recursos donde puedes aprender más.

1.2. Cómo está organizado este libro

La descripción anterior de las herramientas propias de las ciencia de datos está organizada aproximadamente de acuerdo al orden en que usualmente se usan en el análisis (aunque, por supuesto, tendrás que iterar entre ellas múltiples veces). Sin embargo, en nuestra experiencia esa no es la mejor manera de aprenderlas:

- Partir con la ingesta y orden de los datos no es lo óptimo porque el 80 % del tiempo es un proceso rutinario y aburrido y el 20 % restante es extraño y frustrante. ¡No es un buen lugar para aprender un tema nuevo! En cambio, partiremos con la visualización y transformación de datos que ya han sido importados y ordenados. De esta manera, cuando tengas que importar y ordenar tus propios datos, tu motivación se mantendrá alta porque sabrás que ese sufrimiento vale la pena.

- Algunos temas se explican mejor con otras herramientas. Por ejemplo, creemos que es más fácil entender qué es un modelo si ya sabes sobre visualización, datos ordenados y programación.

- Las herramientas de programación no son necesariamente interesantes en sí mismas; sin embargo, te permiten enfrentar problemas desafiantes. Te entregaremos una selección de herramientas de programación en la mitad del libro y luego verás cómo se pueden combinar con las herramientas propias de la ciencia de datos para enfrentar problemas de modelado interesantes.

En cada capítulo hemos tratado de mantener un patrón similar: partir con algunos ejemplos motivantes que te permitan ver el panorama completo y luego sumergirnos en los detalles. Cada sección del libro incluye ejercicios que te ayudarán a practicar lo que has aprendido. Pese a que puede ser tentador saltarse los ejercicios, no hay mejor manera de aprender que practicar con problemas reales.

1.3. Qué no vas a aprender

Hay algunos temas importantes que este libro no aborda. Creemos que es importante mantenernos enfocados con determinación en los aspectos esenciales, con el fin de que puedas ponerte en marcha lo más rápido posible. Eso implica que este libro no puede abordar todos los temas importantes.

1.3.1. Big data

Este libro se enfoca con orgullo en conjuntos de datos pequeños procesables en la memoria de tu computadora. Este es el lugar adecuado para partir, ya que no es posible que te enfrentes a datos de gran tamaño sin antes haber tenido experiencia con otros más pequeños. Las herramientas que aprenderás en este libro permiten manejar fácilmente datos de cientos de megabytes y, con un poco de cuidado, normalmente podrías hacerlas funcionar con 1 o 2 Gb de datos. Si habitualmente trabajas con datos más grandes (por ejemplo, 10-100 Gb), sería bueno que aprendieras sobre data.table. Este libro no enseña `data.table`, ya que su interfaz concisa hace que sea difícil de aprender por las pocas pistas lingüísticas que entrega. Sin embargo, si trabajas con datos grandes, la ventaja en términos de rendimiento hace que valga la pena el esfuerzo extra que requiere aprenderlo. Si tus datos son más grandes que eso, es importante que consideres cuidadosamente si tu problema de *big data* no es, en

realidad, un problema de datos pequeños oculto. Si bien los datos completos pueden ser grandes, muchas veces los necesarios para responder una pregunta específica son menos. Puede que encuentres un subconjunto, una submuestra o un resumen de datos que sí caben en la memoria y que de todos modos te permiten responder la pregunta que te interesa. El desafío en este caso es encontrar los datos pequeños adecuados, lo que usualmente requiere muchas iteraciones.

Otra posibilidad es que tu problema de *big data* sea realmente una suma de problemas de datos pequeños. Puede que cada uno de estos problema individuales quepa en la memoria; el problema es que tienes millones de ellos. Por ejemplo, puedes querer ajustar un modelo para cada persona de tu conjunto de datos. Eso sería trivial si tuvieras solo 10 o 100 personas, pero no si tienes un millón. Afortunadamente, cada problema es independiente del resto (una configuración a la que a veces se le llama de manera vergonzosa *paralela*), por lo que solo necesitas un sistema (como Hadoop o Spark) que te permita enviar diferentes sets de datos a diferentes computadoras para procesarlos. Una vez que hayas resuelto cómo responder la pregunta para un subset de datos usando las herramientas descritas en este libro, podrás aprender otras nuevas como `sparklyr`, `RHIPE` y `ddr` para responder la pregunta para todo el *dataset*.

1.3.2. Python, Julia y amigos

En este libro no aprenderás nada sobre Python, Julia u otros lenguajes de programación útiles para hacer ciencia de datos. No es que creamos que estas herramientas sean malas. ¡No lo son! En la práctica, la mayoría de los equipos de ciencia de datos utilizan una mezcla de lenguajes, habitualmente al menos R y Python.

Sin embargo, creemos fuertemente que es preferible dominar una sola herramienta a la vez. Mejorarás más rápido si te sumerges en un tema con profundidad, en vez de dispersarte entre muchos temas distintos. Esto no quiere decir que solo tengas que saber una cosa, solamente que aprenderás más rápido si te enfocas en una a la vez. Debes tratar de aprender cosas nuevas a lo largo de tu carrera, pero asegúrate de que tu entendimiento sea sólido antes de moverte hacia el siguiente tema interesante.

Creemos que R es un gran lugar para empezar tu camino en la ciencia de datos, ya que es un ambiente diseñado desde las bases hacia arriba para apoyarla. R no es solo un lenguaje de programación, sino también un ambiente interactivo para hacer ciencia de datos. Para apoyar esta interacción, R es mucho más flexible que muchos de sus equivalentes. Si bien esta flexibilidad tiene desventajas, su lado positivo es que permite desarrollar con facilidad gramáticas que se ajustan a las distintas partes de la ciencia de datos. Estos mini-lenguajes te ayudan a pensar los problemas como científico/a de datos, al tiempo que apoyan una interacción fluida entre tu cerebro y la computadora.

1.3.3. Datos no rectangulares

Este libro se enfoca exclusivamente en datos rectangulares, esto es, en conjuntos de valores que están asociados cada uno con una variable y una observación. Hay muchos conjuntos de datos que no se ajustan naturalmente a este paradigma, los que incluyen imágenes, sonidos, árboles y texto. Sin embargo, los *data frames* rectangulares son tan comunes en la ciencia y en la industria, que creemos que son un buen lugar para iniciar tu camino en la ciencia de datos.

1.3.4. Confirmación de hipótesis

Es posible dividir el análisis de datos en dos áreas: generación de hipótesis y confirmación de hipótesis (a veces llamada análisis confirmatorio). El foco de este libro está puesto decididamente en la generación de hipótesis o exploración de datos. Acá mirarás con profundidad los datos y, en combinación con tu propio conocimiento, generarás muchas hipótesis interesantes que ayudarán a explicar por qué se comportan como lo hacen. Evaluarás las hipótesis informalmente, usando tu escepticismo para cuestionar los datos de distintas maneras.

El complemento de la generación de hipótesis es la confirmación de hipótesis. Esta última es difícil por dos razones:

1. Necesitas un modelo matemático con el fin de generar predicciones falseables. Esto usualmente requiere considerable sofisticación estadística.

2. Cada observación puede ser utilizada una sola vez para confirmar una hipótesis. En el momento en que la usas más de una vez ya estás de vuelta haciendo análisis exploratorio. Esto quiere decir que para hacer confirmación de hipótesis tienes que haber "pre-registrado" (es decir, haber escrito con anticipación) tu plan de análisis y no desviarte de él incluso cuando hayas visto tus datos. Hablaremos un poco acerca de estrategias que puedes utilizar para hacer esto más fácil en el capítulo sobre modelos.

Es habitual pensar en el modelado como una herramienta para la confirmación de hipótesis y a la visualización como una herramienta para la generación de hipótesis. Sin embargo, esa es una falsa dicotomía: los modelos son utilizados usualmente para la exploración y, con un poco de cuidado, puedes usar la visualización para confirmar una hipótesis. La diferencia clave es qué tan seguido mires cada observación: si la miras solo una vez es confirmación; si la miras más de una vez es exploración.

1.4. Prerrequisitos

Hemos asumido algunas cosas respecto de lo que ya sabes con el fin de poder sacar mayor provecho de este libro. Tienes que tener una *literacidad* numérica general y sería útil que tuvieses algo de experiencia programando. Si nunca has programado antes y puedes leer en inglés, Hands on Programming with R escrito por Garrett podría ser un buen acompañamiento para este libro.

Hay cinco cosas que necesitas para poder ejecutar el código incluido en este libro: R, RStudio, una colección de paquetes llamada **tidyverse**, el paquete **datos** (que incluye los datos en español se se utilizan en los ejemplos y ejercicios) y una serie de otros paquetes. Los paquetes son la unidad fundamental de código reproducible de R. Incluyen funciones reutilizables, la documentación que describe cómo usarlas y datos de muestra.

1.4.1. R

Para descargar R debes acceder a CRAN, llamado así por sus siglas en inglés: the **c**omprehensive **R** **a**rchive **n**etwork. CRAN está compuesto de una serie de servidores espejo repartidos alrededor del

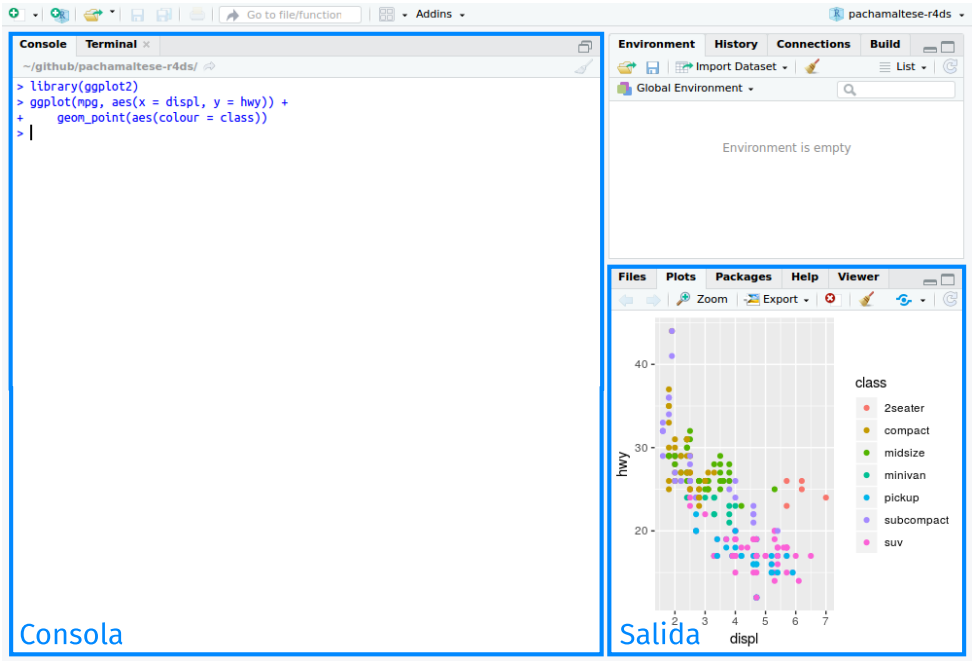
mundo y es utilizado para distribuir tanto R como los paquetes de R. No es necesario que intentes elegir un servidor que esté cerca tuyo: en su lugar, puedes utilizar el servidor en la nube, https://cloud.r-project.org, que automáticamente lo identifica por ti.

Una vez al año sale una nueva versión importante de R y hay entre 2 y 3 ediciones menores en ese período. Es una buena idea actualizarlo regularmente. El proceso puede ser un poco engorroso, especialmente en el caso de las versiones mayores, que requieren que reinstales todos los paquetes que ya tienes. Sin embargo, no hacerlo puede ser peor. Para este libro, asegúrate de tener al menos la versión 3.5.

1.4.2. RStudio

RStudio es un ambiente de desarrollo integrado (o IDE, por su sigla en inglés: Integrated Development Environment) para programar en R. Puedes descargarlo e instalarlo desde http://www.rstudio.com/download. RStudio se actualiza un par de veces al año. Cuando haya una nueva versión disponible, el mismo programa te lo hará saber. Es una buena idea mantenerlo actualizado para que puedas aprovechar las mejores y más recientes características. Para este libro, asegúrate de tener al menos la versión 1.0.0.

Cuando abras RStudio, verás en la interfaz dos regiones clave:



Por ahora, todo lo que tienes que saber es que el código de R se escribe en la Consola y que hay que presionar Enter para ejecutarlo. ¡Aprenderás más a medida que avancemos!

1.4.3. El Tidyverse

Es necesario que instales también algunos paquetes de R. Un **paquete** es una colección de funciones, datos y documentación que permite extender las capacidades de R base. Los paquetes son clave para usar R de manera exitosa. La mayoría de los paquetes que aprenderás a usar en este libro son parte del llamado "Tidyverse". Los paquetes del Tidyverse comparten una filosofía acerca de los datos y la programación en R, y están diseñados para trabajar juntos con naturalidad. Su nombre viene de la palabra en inglés "tidy", que quiere decir "ordenado".

Puedes instalar el **tidyverse** completo con una sola línea de código:

```
install.packages("tidyverse")
```

Escribe en tu computadora esa línea de código en la consola y luego presiona Enter para ejecutarla. R descargará los paquetes de CRAN y los instalará en tu computadora. Si tienes problemas durante la instalación, asegúrate que tienes conexión a Internet y que https://cloud.r-project.org/ no está bloqueado por tu firewall o proxy.

No podrás usar las funciones, objetos y archivos de ayuda de un paquete hasta que lo hayas cargado con `library()`. Una vez que has instalado un paquete, puedes cargarlo con la función `library()`:

```
library(tidyverse)
```

Este mensaje te indica que el tidyverse está cargando los paquetes **ggplot2**, **tibble**, **tidyr**, **readr**, **purrr**, **dplyr**, **stringr** y **forcats**. Estos son considerados el **corazón** del Tidyverse porque los usarás prácticamente en cualquier análisis.

Los paquetes del Tidyverse cambian con bastante frecuencia. Puedes ver si existen actualizaciones disponibles y opcionalmente instalarlas ejecutando `tidyverse_update()`.

1.4.4. El paquete `datos`

Con el fin de que este libro sea más accesible para el público hispanoparlante, además de la traducción del texto se han traducido los datos que se utilizan en los ejemplos y ejercicios.

El paquete `datos` se encuentra disponible en Github y puedes instalarlo ejecutando el siguiente código:

```
#install.packages("remotes")
remotes::install_github("cienciadedatos/datos")
```

1.4.5. Otros paquetes

Existen muchas otros paquetes excelentes que no son parte del **tidyverse** porque resuelven problemas de otros ámbitos o porque los principios en los que se basa su diseño son distintos. Esto no los hace mejores o peores, solo diferentes. En otras palabras, el complemento del **tidyverse** no es el *messyverse*

(del inglés *messy*, desordenado), sino muchos otros universos de paquetes interrelacionados. A medida que te enfrentes a más proyectos de ciencia de datos con R, aprenderás sobre nuevos paquetes y nuevas formas de pensar los datos.

1.5. Ejecutar código en R

En la sección anterior te mostramos algunos ejemplos de cómo ejecutar código en R. En el libro, el código se ve así:

```
1 + 2
```

```
[1] 3
```

```
#> [1] 3
```

Si ejecutas el mismo código en tu consola local, se verá así:

```
> 1 + 2
[1] 3
```

Hay dos diferencias principales. La primera, es que en tu consola debes escribir el código después del signo >, llamado *prompt*; en el libro no te mostraremos el *prompt*. La segunda, es que en el libro el *output*, es decir, el resultado de ejecutar el código, está comentado: #>. En tu consola, el output aparecerá directamente después del código. Estas dos diferencias implican que, como esta es una versión electrónica del libro, puedes copiar directamente el código que aparece acá y pegarlo en tu consola.

A lo largo del libro usaremos una serie consistente de convenciones para referirnos al código:

- Las funciones están escritas en una fuente para código y seguidas de paréntesis. La primera vez que son mencionadas ofreceremos una traducción al español: `sum()` (del inglés *suma*) o `mean()` (del inglés *media*).
- Otros tipos de objetos de R (como datos o argumentos de funciones) estarán en fuente para código, pero sin paréntesis: `vuelos` o `x`.
- Si queremos dejar claro de qué paquete viene un objeto, usaremos el nombre del paquete seguido de doble dos puntos: `dplyr::mutate()` o `datos::paises`. Esto también es válido para el código de R.

1.6. Pedir ayuda y aprender más

Este libro no es una isla. No existe ningún recurso que por sí mismo te permita dominar R. A medida que empieces a aplicar las técnicas descritas en este libro a tus propios datos te encontrarás con preguntas que acá no respondemos. En esta sección se describen algunas sugerencias sobre cómo pedir ayuda y cómo seguir aprendiendo.

Si en algún momento ya no puedes avanzar, empieza buscando en Google. Usualmente, agregar "R" a tu búsqueda es suficiente para que se restrinja solo a resultados relevantes. Si lo que encuentras no es útil, probablemente sea porque no hay resultados disponibles en español. Google es particularmente útil para los mensajes de error. Si te aparece uno y no tienes idea de lo que significa, ¡prueba buscando en Google! Lo más probable es que alguien más se haya confundido con ese mensaje en el pasado y que haya ayuda en la web. Si el error te aparece en español u otro idioma, ejecuta en la consola `Sys.setenv(LANGUAGE = "en")` y luego vuelve a ejecutar el código. Es más probable que encuentres ayuda si el error que arroja R está en inglés.

Si Google no ayuda, prueba con la versión en español de stackoverflow. Parte dedicando un tiempo a buscar si existe ya una respuesta a tu pregunta agregando `[R]` a tu búsqueda para restringir los resultados a preguntas y respuestas que usen R. Si no encuentras nada útil, prepara un ejemplo reproducible o **reprex**. Un buen *reprex* hace más fácil que otras personas te puedan ayudar y al prepararlo probablemente resuelvas el problema por tu cuenta.

Hay tres cosas que debes incluir para hacer que tu ejemplo sea reproducible: los paquetes necesarios, datos y código.

1. Los **paquetes** deben ser cargados al inicio del *script* (que es como se le llama a la secuencia de comandos) para que sea fácil ver cuáles se necesitan para el ejemplo. Es una buena oportunidad para chequear que estás utilizando la última versión de cada paquete. Es posible que hayas descubierto un error (o *bug*, en inglés) que ya fue resuelto desde que instalaste el paquete. Para los paquetes del **tidyverse**, la manera más fácil de hacerlo es ejecutando `tidyverse_update()`.

2. La manera más simple de incluir **datos** en una pregunta es usar `dput()` para generar el código de R que los recree. Por ejemplo, para recrear el conjunto de datos `mtautos` en R, tendríamos que realizar los siguientes pasos.

3. Cargar el paquete que contiene los datos: `library(datos)`

4. Ejecutar `dput(mtautos)` en R

5. Copiar el output

6. En tu script reproducible, escribir `mtautos <-` y luego pegar lo copiado.

Trata de buscar el subconjunto más pequeño de tus datos que te permita mostrar tu problema.

1. Dedica tiempo a asegurarte que tu **código** puede ser fácilmente leído por otras personas:

- Asegúrate de haber utilizado espacios y que los nombre de tus variables son a la vez concisos e informativos.

- Realiza comentarios que indiquen dónde se encuentra el problema.

- Haz lo posible por remover todo lo que no esté relacionado con el problema. Mientras más breve tu código, más fácil de entender y más fácil de arreglar.

Finalmente, revisa que tu ejemplo es efectivamente reproducible. Para ello, inicia una nueva sesión de R y copia y pega tu script ahí.

También deberías dedicar tiempo a prepararte para resolver problemas por tu cuenta antes de que ocurran. Invertir un poco de tiempo cada día aprendiendo R te entrega grandes beneficios a largo plazo. Una manera es siguiendo lo que hacen Hadley, Garrett y todas las personas de RStudio en el blog de RStudio. Ahí es donde se publican anuncios sobre nuevos paquetes, nuevas características del entorno de desarrollo integrado (IDE) y talleres presenciales. También te podría interesar seguir a Hadley (@hadleywickham) o a Garrett (@statgarrett) en Twitter, o a la cuenta @rstudiotips para mantenerte al día sobre las nuevas características RStudio.

Para estar al tanto acerca de la comunidad de R en general, puedes leer http://www.r-bloggers.com: que agrupa cerca de 500 blogs sobre R de todas partes del mundo, algunos incluso en español. Si tienes cuenta en Twitter, puedes seguir el hashtag `#rstatsES` (en español) o `#rstats` (en inglés). Twitter es una de las herramientas clave que usa Hadley para mantenerse al día sobre los nuevos desarrollos de la comunidad.

1.7. Agradecimientos

Este libro no es solo el producto de Hadley y Garrett, sino el resultado de muchas conversaciones (en persona y en línea) que hemos tenido con gente de la comunidad de R. Hay algunas personas a las que nos gustaría agradecer de manera particular, ya que han invertido muchas horas respondiendo preguntas tontas y ayudándonos a pensar mejor acerca de la ciencia de datos:

- Jenny Bryan y Lionel Henry, por las muchas conversaciones útiles en torno al trabajo con listas y listas-columnas.
- Los tres capítulos sobre flujo de trabajo fueron adaptados (con permiso) de http://stat545.com/block002_hello-r-workspace-wd-project.html de Jenny Bryan.
- Genevera Allen por las dicusiones acerca de modelos, modelado, la perspectiva sobre de aprendizaje estadístico y la diferencia entre generación de hipótesis y confirmación de hipótesis.
- Yihui Xie por su trabajo en el paquete bookdown, y por responder incansablemente las peticiones de nuevas características.
- Bill Behrman por la lectura atenta del libro completo y por probarlo en su curso de Ciencia de Datos en Stanford.
- A la comunidad en Twitter de #rstats que revisó todos los borradores de los capítulos y ofreció un montón de retroalimentación útil.
- Tal Galili, por aumentar su paquete **dendextend** para soportar una sección sobre *clustering* que no llegó al borrador final.

Este libro fue escrito de manera abierta y muchas personas contribuyeron con *pull requests* para resolver problemas pequeños. Especiales agradecimientos para todos quienes contribuyeron via Github:

Thanks go to all contributers in alphabetical order: adi pradhan, Ahmed ElGabbas, Ajay Deonarine, @Alex, Andrew Landgraf, bahadir cankardes, @batpigandme, @behrman, Ben Marwick, Bill Behrman, Brandon Greenwell, Brett Klamer, Christian G. Warden, Christian Mongeau, Colin Gillespie, Cooper Mo-

rris, Curtis Alexander, Daniel Gromer, David Clark, Derwin McGeary, Devin Pastoor, Dylan Cashman, Earl Brown, Eric Watt, Etienne B. Racine, Flemming Villalona, Gregory Jefferis, @harrismcgehee, Hengni Cai, Ian Lyttle, Ian Sealy, Jakub Nowosad, Jennifer (Jenny) Bryan, @jennybc, Jeroen Janssens, Jim Hester, @jjchern, Joanne Jang, John Sears, Jon Calder, Jonathan Page, @jonathanflint, Jose Roberto Ayala Solares, Julia Stewart Lowndes, Julian During, Justinas Petuchovas, Kara Woo, @kdpsingh, Kenny Darrell, Kirill Sevastyanenko, @koalabearski, @KyleHumphrey, Lawrence Wu, Matthew Sedaghatfar, Mine Cetinkaya-Rundel, @MJMarshall, Mustafa Ascha, @nate-d-olson, Nelson Areal, Nick Clark, @nickelas, Nirmal Patel, @nwaff, @OaCantona, Patrick Kennedy, @Paul, Peter Hurford, Rademeyer Vermaak, Radu Grosu, @rlzijdeman, Robert Schuessler, @robinlovelace, @robinsones, S'busiso Mkhondwane, @seamus-mckinsey, @seanpwilliams, Shannon Ellis, @shoili, @sibusiso16, @spirgel, Steve Mortimer, @svenski, Terence Teo, Thomas Klebel, TJ Mahr, Tom Prior, Will Beasley, @yahwes, Yihui Xie, @zeal626.

1.8. Colofón

La versión *online* de este libro está disponible en http://es.r4ds.hadley.nz. El original en inglés puedes encontrarlo en http://r4ds.had.co.nz. Las versiones *online* seguirán evolucionando entre reimpresiones del libro físico. La fuente del libro en español está disponible en https://github.com/cienciadedatos/r4ds y la del libro en inglés en https://github.com/hadley/r4ds. El libro funciona con https://bookdown.org, que hace fácil convertir archivos de R Markdown a HTML, PDF e EPUB.

Este libro fue construido con:

```
devtools::session_info(c("tidyverse"))
```

```
- Session info ---------------------------------------------------------
--
 setting  value
 version  R version 3.6.3 (2020-02-29)
 os       Ubuntu 18.04.4 LTS
 system   x86_64, linux-gnu
 ui       X11
 language (EN)
 collate  en_US.UTF-8
 ctype    en_US.UTF-8
 tz       America/Santiago
 date     2020-06-14

- Packages -------------------------------------------------------------
--
 package      * version  date       lib source
 askpass        1.1      2019-01-13 [1] CRAN (R 3.6.3)
 assertthat     0.2.1    2019-03-21 [1] CRAN (R 3.6.3)
 backports      1.1.7    2020-05-13 [1] CRAN (R 3.6.3)
 base64enc      0.1-3    2015-07-28 [1] CRAN (R 3.6.3)
 BH             1.72.0-3 2020-01-08 [1] CRAN (R 3.6.3)
 blob           1.2.1    2020-01-20 [1] CRAN (R 3.6.3)
```

```
broom          0.5.6    2020-04-20 [1] CRAN (R 3.6.3)
callr          3.4.3    2020-03-28 [1] CRAN (R 3.6.3)
cellranger     1.1.0    2016-07-27 [1] CRAN (R 3.6.3)
cli            2.0.2    2020-02-28 [1] CRAN (R 3.6.3)
clipr          0.7.0    2019-07-23 [1] CRAN (R 3.6.3)
colorspace     1.4-1    2019-03-18 [1] CRAN (R 3.6.3)
crayon         1.3.4    2017-09-16 [1] CRAN (R 3.6.3)
curl           4.3      2019-12-02 [1] CRAN (R 3.6.3)
DBI            1.1.0    2019-12-15 [1] CRAN (R 3.6.3)
dbplyr         1.4.4    2020-05-27 [1] CRAN (R 3.6.3)
desc           1.2.0    2018-05-01 [1] CRAN (R 3.6.3)
digest         0.6.25   2020-02-23 [1] CRAN (R 3.6.3)
dplyr        * 1.0.0    2020-05-29 [1] CRAN (R 3.6.3)
ellipsis       0.3.1    2020-05-15 [1] CRAN (R 3.6.3)
evaluate       0.14     2019-05-28 [1] CRAN (R 3.6.3)
fansi          0.4.1    2020-01-08 [1] CRAN (R 3.6.3)
farver         2.0.3    2020-01-16 [1] CRAN (R 3.6.3)
forcats      * 0.5.0    2020-03-01 [1] CRAN (R 3.6.3)
fs             1.4.1    2020-04-04 [1] CRAN (R 3.6.3)
generics       0.0.2    2018-11-29 [1] CRAN (R 3.6.3)
ggplot2      * 3.3.1    2020-05-28 [1] CRAN (R 3.6.3)
glue           1.4.1    2020-05-13 [1] CRAN (R 3.6.3)
gtable         0.3.0    2019-03-25 [1] CRAN (R 3.6.3)
haven          2.3.1    2020-06-01 [1] CRAN (R 3.6.3)
highr          0.8      2019-03-20 [1] CRAN (R 3.6.3)
hms            0.5.3    2020-01-08 [1] CRAN (R 3.6.3)
htmltools      0.4.0    2019-10-04 [1] CRAN (R 3.6.3)
httr           1.4.1    2019-08-05 [1] CRAN (R 3.6.3)
isoband        0.2.1    2020-04-12 [1] CRAN (R 3.6.3)
jsonlite       1.6.1    2020-02-02 [1] CRAN (R 3.6.3)
knitr          1.28     2020-02-06 [1] CRAN (R 3.6.3)
labeling       0.3      2014-08-23 [1] CRAN (R 3.6.3)
lattice        0.20-41  2020-04-02 [4] CRAN (R 3.6.3)
lifecycle      0.2.0    2020-03-06 [1] CRAN (R 3.6.3)
lubridate      1.7.9    2020-06-08 [1] CRAN (R 3.6.3)
magrittr       1.5      2014-11-22 [1] CRAN (R 3.6.3)
markdown       1.1      2019-08-07 [1] CRAN (R 3.6.3)
MASS           7.3-51.6 2020-04-26 [4] CRAN (R 3.6.3)
Matrix         1.2-18   2019-11-27 [4] CRAN (R 3.6.1)
mgcv           1.8-31   2019-11-09 [4] CRAN (R 3.6.1)
mime           0.9      2020-02-04 [1] CRAN (R 3.6.3)
modelr         0.1.8    2020-05-19 [1] CRAN (R 3.6.3)
munsell        0.5.0    2018-06-12 [1] CRAN (R 3.6.3)
nlme           3.1-147  2020-04-13 [4] CRAN (R 3.6.3)
openssl        1.4.1    2019-07-18 [1] CRAN (R 3.6.3)
pillar         1.4.4    2020-05-05 [1] CRAN (R 3.6.3)
pkgbuild       1.0.8    2020-05-07 [1] CRAN (R 3.6.3)
pkgconfig      2.0.3    2019-09-22 [1] CRAN (R 3.6.3)
```

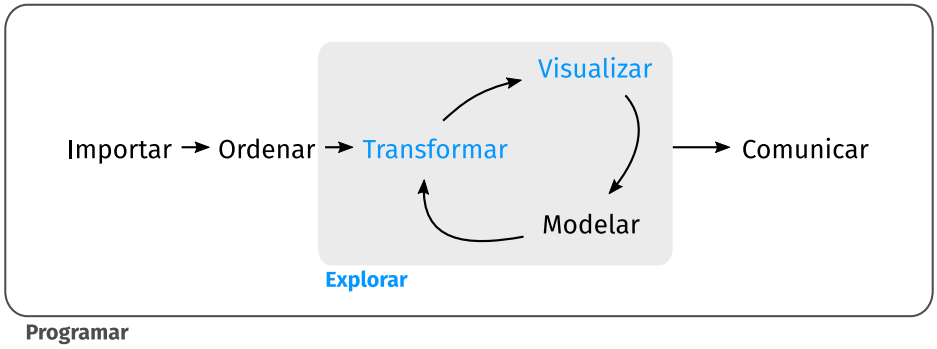
```
 pkgload         1.1.0    2020-05-29 [1] CRAN (R 3.6.3)
 plyr            1.8.6    2020-03-03 [1] CRAN (R 3.6.3)
 praise          1.0.0    2015-08-11 [1] CRAN (R 3.6.3)
 prettyunits     1.1.1    2020-01-24 [1] CRAN (R 3.6.3)
 processx        3.4.2    2020-02-09 [1] CRAN (R 3.6.3)
 progress        1.2.2    2019-05-16 [1] CRAN (R 3.6.3)
 ps              1.3.3    2020-05-08 [1] CRAN (R 3.6.3)
 purrr         * 0.3.4    2020-04-17 [1] CRAN (R 3.6.3)
 R6              2.4.1    2019-11-12 [1] CRAN (R 3.6.3)
 RColorBrewer    1.1-2    2014-12-07 [1] CRAN (R 3.6.3)
 Rcpp            1.0.4.6  2020-04-09 [1] CRAN (R 3.6.3)
 readr         * 1.3.1    2018-12-21 [1] CRAN (R 3.6.3)
 readxl          1.3.1    2019-03-13 [1] CRAN (R 3.6.3)
 rematch         1.0.1    2016-04-21 [1] CRAN (R 3.6.3)
 reprex          0.3.0    2019-05-16 [1] CRAN (R 3.6.3)
 reshape2        1.4.4    2020-04-09 [1] CRAN (R 3.6.3)
 rlang           0.4.6    2020-05-02 [1] CRAN (R 3.6.3)
 rmarkdown       2.2      2020-05-31 [1] CRAN (R 3.6.3)
 rprojroot       1.3-2    2018-01-03 [1] CRAN (R 3.6.3)
 rstudioapi      0.11     2020-02-07 [1] CRAN (R 3.6.3)
 rvest           0.3.5    2019-11-08 [1] CRAN (R 3.6.3)
 scales          1.1.1    2020-05-11 [1] CRAN (R 3.6.3)
 selectr         0.4-2    2019-11-20 [1] CRAN (R 3.6.3)
 stringi         1.4.6    2020-02-17 [1] CRAN (R 3.6.3)
 stringr       * 1.4.0    2019-02-10 [1] CRAN (R 3.6.3)
 sys             3.3      2019-08-21 [1] CRAN (R 3.6.3)
 testthat        2.3.2    2020-03-02 [1] CRAN (R 3.6.3)
 tibble        * 3.0.1    2020-04-20 [1] CRAN (R 3.6.3)
 tidyr         * 1.1.0    2020-05-20 [1] CRAN (R 3.6.3)
 tidyselect      1.1.0    2020-05-11 [1] CRAN (R 3.6.3)
 tidyverse     * 1.3.0    2019-11-21 [1] CRAN (R 3.6.3)
 tinytex         0.23     2020-05-19 [1] CRAN (R 3.6.3)
 utf8            1.1.4    2018-05-24 [1] CRAN (R 3.6.3)
 vctrs           0.3.1    2020-06-05 [1] CRAN (R 3.6.3)
 viridisLite     0.3.0    2018-02-01 [1] CRAN (R 3.6.3)
 whisker         0.4      2019-08-28 [1] CRAN (R 3.6.3)
 withr           2.2.0    2020-04-20 [1] CRAN (R 3.6.3)
 xfun            0.14     2020-05-20 [1] CRAN (R 3.6.3)
 xml2            1.3.2    2020-04-23 [1] CRAN (R 3.6.3)
 yaml            2.2.1    2020-02-01 [1] CRAN (R 3.6.3)

[1] /home/pacha/R/x86_64-pc-linux-gnu-library/3.6
[2] /usr/local/lib/R/site-library
[3] /usr/lib/R/site-library
[4] /usr/lib/R/library
```

(PART) Explorar

CAPÍTULO 2

Introducción

El objetivo de la primera parte de este libro es introducirte en las herramientas básicas de *exploración de datos* de la manera más veloz posible. La exploración de datos es el arte de mirar tus datos, generar hipótesis rápidamente, testearlas con celeridad y luego repetir el proceso de manera iterativa. El objetivo de la exploración de datos es generar muchos hallazgos prometedores que luego puedas retomar para explorarlos en mayor profundidad.



En esta parte del libro aprenderás algunas herramientas útiles que tienen un beneficio inmediato: * La visualización es un buen lugar en el que comenzar a programar en R, ya que el retorno que se obtiene es claro: puedes crear gráficos elegantes e informativos que te ayuden a entender los datos. En el capítulo sobre visualización de datos vas a profundizar en este tema y aprenderás la estructura básica de un gráfico en **ggplot2**,técnicas que te permitirán convertir datos en gráficos.

- La visualización por sí sola a menudo no es suficiente, por lo que en transformación de datos aprenderás los verbos clave que te permitirán seleccionar variables importantes, filtrar observaciones, crear nuevas variables y calcular estadísticos que resuman información.
- Finalmente, en [análisis exploratorio de datos (EDA)] vas a combinar visualización y transformación con tu curiosidad y escepticismo para formular y responder preguntas en torno a los datos.

Si bien modelar es un aspecto importante del proceso exploratorio, aún no tienes las habilidades para

aprenderlo con efectividad o aplicarlo. Volveremos sobre este tema en el capítulo sobre introductorio de Modelos, una vez que ya tengas las herramientas de manejo de datos y programación.

Entre estos tres capítulos que enseñan las herramientas de exploración de datos hay otros tres capítulos que se enfocan en el flujo de trabajo en R. En flujo de trabajo: conocimientos básicos, [flujo de trabajo: *Scripts*] y [flujo de trabajo: proyectos] aprenderás buenas prácticas para escribir y organizar tu código en R. Estas prácticas te prepararán para el éxito a el largo plazo, en términos de que te entregan las herramientas necesarias para organizarte cuando abordes proyectos reales.

CAPÍTULO 3

Visualización de datos

3.1. Introducción

> "Un simple gráfico ha brindado más información a la mente del analista de datos que cualquier otro dispositivo". — John Tukey

En este capítulo aprenderás cómo visualizar tus datos usando el paquete **ggplot2**. De los muchos sistemas que posee R para hacer gráficos, **ggplot2** es uno de los más elegantes y versátiles. Esto se debe a que **ggplot2** implementa un sistema coherente para describir y construir gráficos, conocido como la **gramática de gráficos**. Con **ggplot2** puedes hacer más cosas en menor tiempo, aprendiendo un único sistema y aplicándolo en diferentes ámbitos.

Si deseas obtener más información sobre los fundamentos teóricos de **ggplot2** antes de comenzar, te recomendamos leer "La gramática de gráficos en capas", http://vita.had.co.nz/papers/layered-grammar.pdf.

3.1.1. Prerrequisitos

Este capítulo se centra en **ggplot2**, uno de los paquetes principales del Tidyverse. Para acceder a sus funciones y las páginas de ayuda que utilizaremos en este capítulo, debes cargar el Tidyverse ejecutando este código:

```
library(tidyverse)
```

Esa única línea de código carga el núcleo del Tidyverse, que está compuesto por los paquetes que usarás en casi todos tus análisis de datos. Al correr esta línea también verás cuáles funciones de tidyverse pueden tener conflicto con funciones de R base (o de otros paquetes que puedas haber cargado previamente).

Si ejecutas este código y recibes el mensaje de error "Error in library(tidyverse): there is no package called 'tidyverse'" (no hay ningún paquete llamado 'tidyverse'), primero deberás instalarlo y luego ejecutar `library()`:.

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

Solo es necesario que instales los paquetes una única vez; sin embargo, tendrás que cargarlos siempre que inicies una nueva sesión.

Cuando necesitemos especificar la procedencia de una función (o un conjunto de datos), usaremos el formato especial `paquete::funcion()`. Por ejemplo, `ggplot2::ggplot()` dice explícitamente que estamos usando la función `ggplot()` del paquete ggplot2.

Además del Tidyverse, es necesario que cargue el paquete `datos`, ya que en él están contenidas las versiones en español de los datos que utilizaremos en este capítulo:

```
# install.packages("remotes")
# remotes::install_github("cienciadedatos/datos")
library(datos)
```

3.2. Primeros pasos

Usemos nuestro primer gráfico para responder una pregunta: ¿los automóviles con motores grandes consumen más combustible que los automóviles con motores pequeños? Probablemente ya tengas una respuesta, pero trata de responder de forma precisa. ¿Cómo es la relación entre el tamaño del motor y la eficiencia del combustible? ¿Es positiva? ¿Es negativa? ¿Es lineal o no lineal?

3.2.1. El *data frame* `millas`

Puedes poner a prueba tu respuesta empleando el *data frame* `millas` que se encuentra en el paquete **datos** (`datos::millas`). Un *data frame* es una colección rectangular de variables (columnas) y observaciones (filas). El *data frame* `millas` contiene observaciones para 38 modelos de automóviles recopiladas por la Agencia de Protección Ambiental de los EE. UU.

```
millas
#> # A tibble: 234 x 11
#>   fabricante modelo cilindrada  anio cilindros transmision traccion ciudad
#>   <chr>      <chr>       <dbl> <int>     <int> <chr>       <chr>     <int>
#> 1 audi       a4            1.8  1999         4 auto(l5)    d            18
#> 2 audi       a4            1.8  1999         4 manual(m5)  d            21
#> 3 audi       a4            2    2008         4 manual(m6)  d            20
#> 4 audi       a4            2    2008         4 auto(av)    d            21
#> 5 audi       a4            2.8  1999         6 auto(l5)    d            16
#> 6 audi       a4            2.8  1999         6 manual(m5)  d            18
#> # ... with 228 more rows, and 3 more variables: autopista <int>,
#> #   combustible <chr>, clase <chr>
```

Entre las variables de `millas` se encuentran:

1. `cilindrada`: tamaño del motor del automóvil, en litros.

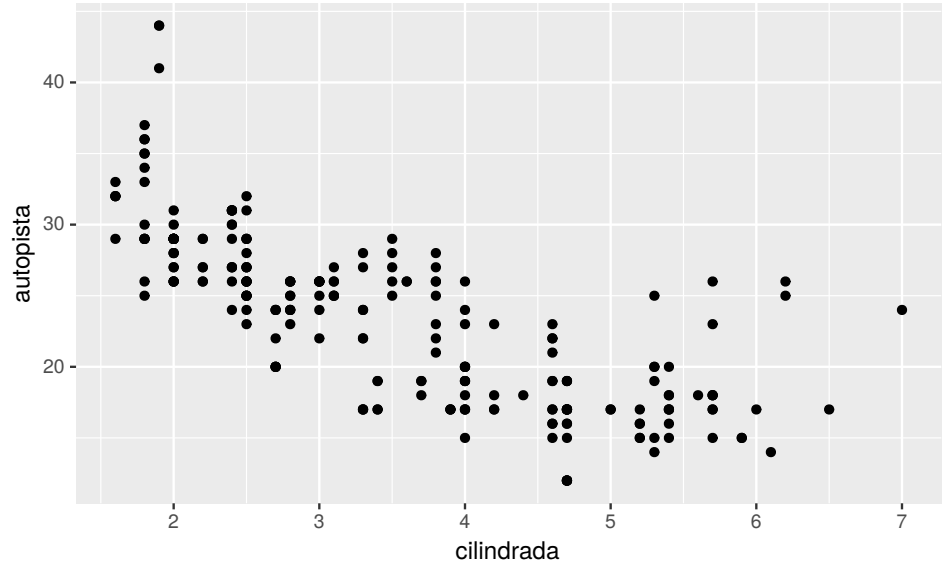
2. `autopista`: eficiencia del uso de combustible de un automóvil en carretera, en millas por galón. Al recorrer la misma distancia, un automóvil de baja eficiencia consume más combustible que un automóvil de alta eficiencia.

Para obtener más información sobre el *data frame* `millas`, abre su página de ayuda ejecutando `?millas`.

3.2.2. Creando un gráfico con ggplot

Para graficar `millas`, ejecuta este código para poner `cilindrada` en el eje x y `autopista` en el eje y:

```
ggplot(data = millas) +
  geom_point(mapping = aes(x = cilindrada, y = autopista))
```



El gráfico muestra una relación negativa entre el tamaño del motor (`cilindrada`) y la eficiencia del combustible (`autopista`). En otras palabras, los vehículos con motores grandes usan más combustible. Este resultado, ¿confirma o refuta tu hipótesis acerca de la relación entre la eficiencia del combustible y el tamaño del motor?

Para comenzar un gráfico con **ggplot2** se utiliza la función `ggplot()`. `ggplot()` crea un sistema de coordenadas al que puedes agregar capas. El primer argumento de `ggplot()` es el conjunto de datos que se utilizará en el gráfico. Si corres `ggplot(data = millas)`, obtendrás un gráfico vacío. Como no es muy interesante, no vamos a mostrarlo aquí.

Para completar tu gráfico debes agregar una o más capas a `ggplot()`. La función `geom_point()` agrega una capa de puntos al gráfico, que crea un diagrama de dispersión (o *scatterplot*). **ggplot2** incluye muchas funciones *geom*, cada una de las cuales agrega un tipo de capa diferente a un gráfico. Aprenderás muchas de ellas a lo largo de este capítulo.

Cada función geom en **ggplot2** tiene un argumento de `mapping`. Este define cómo se "mapean" o se asignan las variables del conjunto de datos a propiedades visuales. El argumento de `mapping` siempre aparece emparejado con `aes()` y los argumentos `x` e `y` dentro de `aes()` especifican qué variables asignar a estos ejes. **ggplot2** busca la variable asignada en el argumento `data`, en este caso, `millas`.

3.2.3. Una plantilla de gráficos

Convirtamos ahora este código en una plantilla reutilizable para hacer gráficos con **ggplot2**. Para hacer un gráfico, reemplaza las secciones entre corchetes en el siguiente código con un conjunto de datos, una función geom o una colección de mapeos.

```
ggplot(data = <DATOS>) +
  <GEOM_FUNCIÓN>(mapping = aes(<MAPEOS>))
```

El resto de este capítulo te mostrará cómo utilizar y adaptar esta plantilla para crear diferentes tipos de gráficos. Comenzaremos por el componente <MAPEOS>

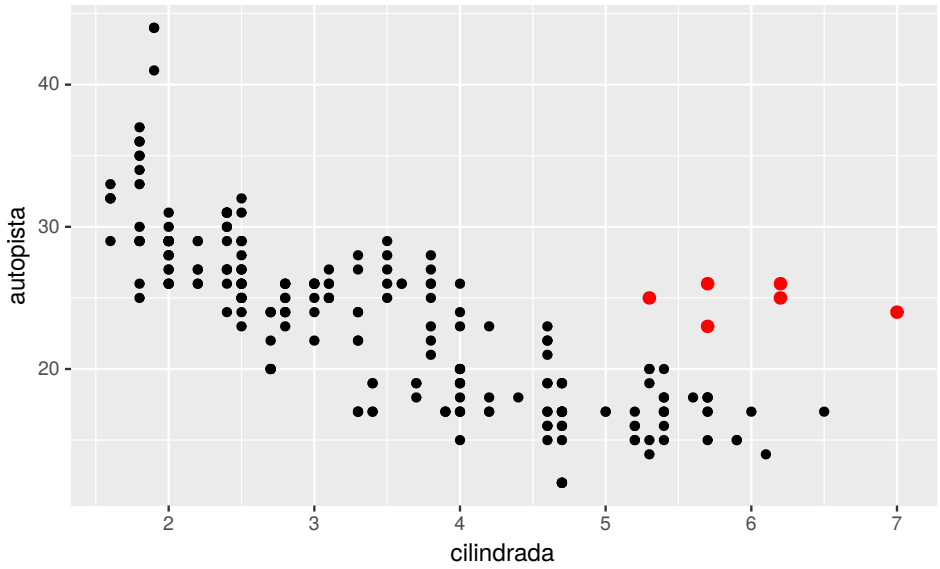
3.2.4. Ejercicios

1. Ejecuta `ggplot(data = millas)`. ¿Qué observas?
2. ¿Cuántas filas hay en `millas`? ¿Cuántas columnas?
3. ¿Qué describe la variable `traccion`? Lee la ayuda de `?millas` para encontrar la respuesta.
4. Realiza un gráfico de dispersión de `autopista` versus `cilindros`.
5. ¿Qué sucede cuando haces un gráfico de dispersión (*scatterplot*) de `clase` versus `traccion`? ¿Por qué no es útil este gráfico?

3.3. Mapeos estéticos

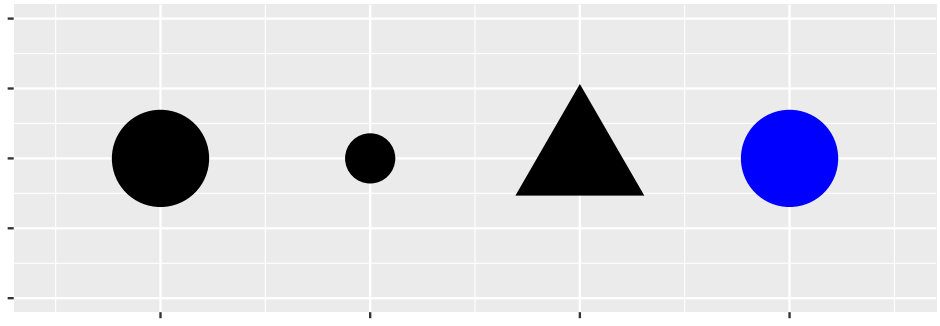
> "El mayor valor de una imagen es cuando nos obliga a observar lo que no esperábamos ver". — John Tukey

En el siguiente gráfico, un grupo de puntos (resaltados en rojo) parece quedar fuera de la tendencia lineal. Estos automóviles tienen un kilometraje mayor de lo que esperaríamos. ¿Cómo puedes explicar estos vehículos?



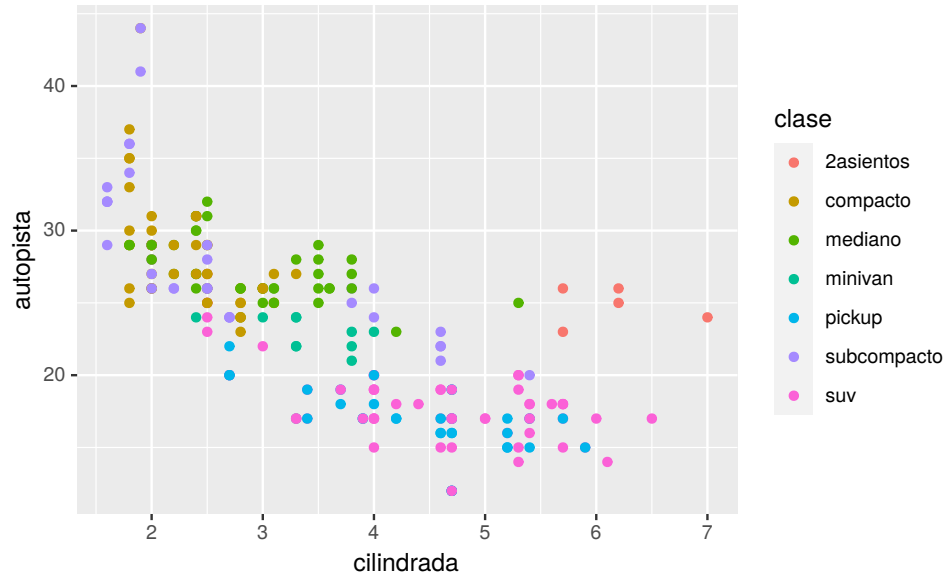
Supongamos que estos automóviles son híbridos. Una forma de probar esta hipótesis es observando la variable que indica la `clase` de cada automóvil. La variable `clase` del conjunto de datos de `millas` clasifica los autos en grupos como compacto, mediano y SUV. Si los puntos periféricos corresponden a automóviles híbridos, deberían estar clasificados como compactos o, tal vez, subcompactos (ten en cuenta que estos datos se recopilaron antes de que las camionetas híbridas y SUV se hicieran populares).

Puedes agregar una tercera variable, como `clase`, a un diagrama de dispersión bidimensional asignándolo a un parámetro **estético**. Un parámetro estético (o *estética*) es una propiedad visual de los objetos de un gráfico. Las estéticas incluye cosas como el tamaño, la forma o el color de tus puntos. Puedes mostrar un punto (como el siguiente) de diferentes maneras si cambias los valores de sus propiedades estéticas. Como ya usamos la palabra "valor" para describir los datos, usemos la palabra "nivel" para describir las propiedades estéticas. Aquí cambiamos los niveles del tamaño, la forma y el color de un punto para que el punto sea pequeño, triangular o azul:



El mapeo entre las propiedades estéticas de tu gráfico y las variables de tu *dataset* te permite comunicar información sobre tus datos. Por ejemplo, puedes asignar los colores de los puntos de acuerdo a la variable `clase` para indicar a qué clase pertenece cada automóvil.

```
ggplot(data = millas) +
  geom_point(mapping = aes(x = cilindrada, y = autopista, color = clase))
```



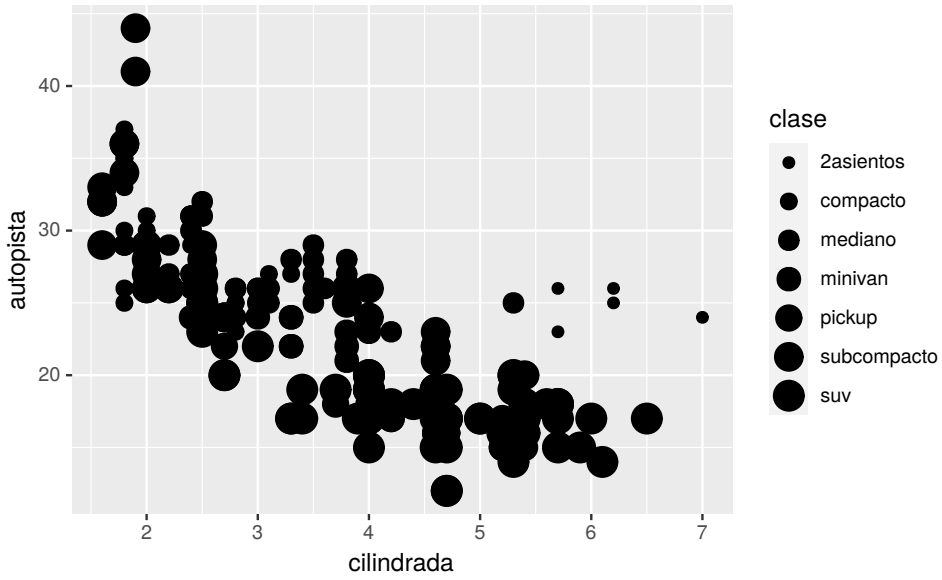
(Si prefieres el inglés británico, como Hadley, puedes usar `colour` en lugar de `color`).

Para mapear (o asignar) una estética a una variable, debes asociar el nombre de la estética al de la variable dentro de `aes()`. **ggplot2** asignará automáticamente un nivel único de la estética (en este ejemplo, un color) a cada valor único de la variable. Este proceso es conocido como **escalamiento** (*scaling*). **ggplot2** acompañará el gráfico con una leyenda que explica qué niveles corresponden a qué valores.

Los colores revelan que muchos de los puntos inusuales son automóviles de dos asientos. ¡Estos no parecen híbridos y son, de hecho, automóviles deportivos! Los automóviles deportivos tienen motores grandes, como las camionetas todo terreno o *pickups*, pero su cuerpo es pequeño, como los automóviles medianos y compactos, lo que mejora su consumo de gasolina. En retrospectiva, es poco probable que estos automóviles sean híbridos, ya que tienen motores grandes.

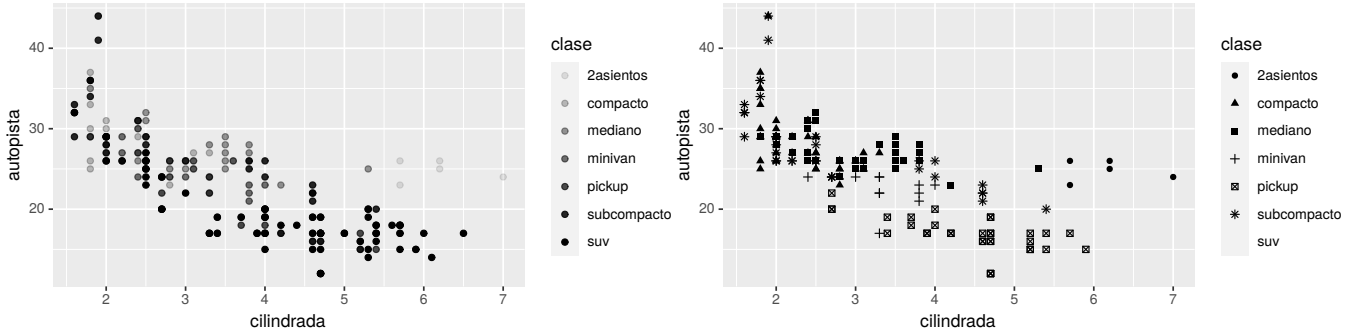
En el ejemplo anterior asignamos la variable `clase` a la estética de color, pero podríamos haberla asignado a la estética del tamaño del mismo modo. En este caso, el tamaño exacto de cada punto revelaría a qué clase pertenece. Recibimos aquí una **advertencia** (*warning*), porque mapear una variable no ordenada (`clase`) a una estética ordenada (`size`) no es una buena idea.

```
ggplot(data = millas) +
  geom_point(mapping = aes(x = cilindrada, y = autopista, size = clase))
```



También podríamos haber asignado la variable `clase` a la estética *alpha*, que controla la transparencia de los puntos, o a la estética *shape* que controla la forma (*shape*) de los puntos.

```
# Izquierda
ggplot(data = millas) +
  geom_point(mapping = aes(x = cilindrada, y = autopista, alpha = clase))

# Derecha
ggplot(data = millas) +
  geom_point(mapping = aes(x = cilindrada, y = autopista, shape = clase))
```



¿Qué pasó con los SUV? **ggplot2** solo puede usar seis formas a la vez. De forma predeterminada, los grupos adicionales no se grafican cuando se emplea la estética de la forma (*shape*).

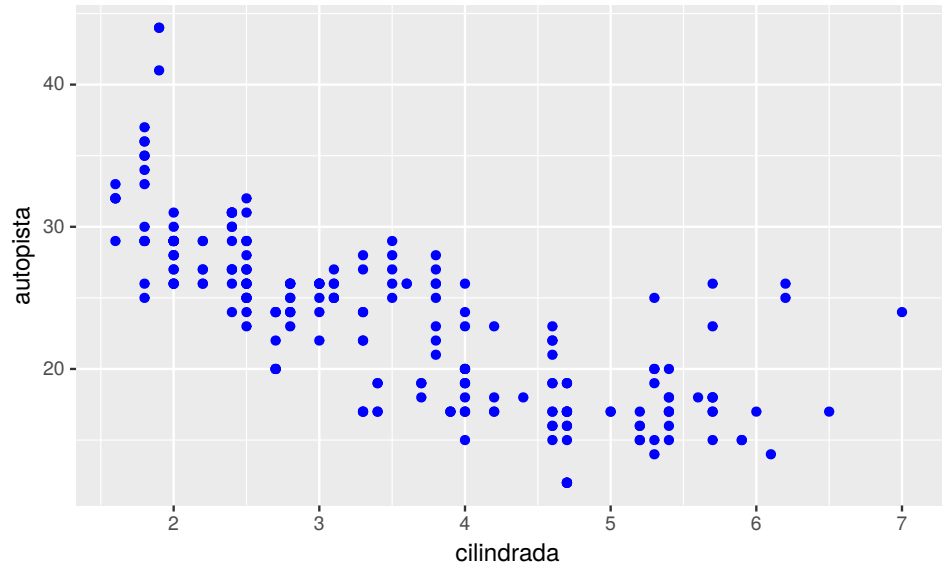
Para cada estética utilizamos `aes()` para asociar su nombre con la variable seleccionada para graficar. La función `aes()` reúne cada una de las asignaciones estéticas utilizadas por una capa y las pasa al argumento de mapeo de la capa. La sintaxis resalta una visión útil sobre x e y: las ubicaciones de x e y de un punto son en sí mismas también estéticas, es decir propiedades visuales que se puede asignar a las variables para mostrar información sobre los datos.

Una vez que asignas (o "mapeas") una estética, **ggplot2** se ocupa del resto. El paquete selecciona una escala razonable para usar con la estética elegida y construye una leyenda que explica la relación entre

niveles y valores. Para la estética x e y, **ggplot2** no crea una leyenda, pero sí una línea que delimita el eje con sus marcas de graduación y una etiqueta. La línea del eje actúa como una leyenda; explica el mapeo entre ubicaciones y valores.

También puedes *fijar* las propiedades estéticas de tu *geom* manualmente. Por ejemplo, podemos hacer que todos los puntos del gráfico sean azules:

```
ggplot(data = millas) +
 geom_point(mapping = aes(x = cilindrada, y = autopista), color = "blue")
```



Aquí, el color no transmite información sobre una variable, sino que cambia la apariencia del gráfico. Para establecer una estética de forma manual, debes usar el nombre de la estética como un argumento de la función geom; es decir, va *fuera* de `aes()`. Tendrás que elegir un nivel que tenga sentido para esa estética:

- El nombre de un color como cadena de caracteres.
- El tamaño de un punto en mm.
- La forma de un punto como un número, tal como se muestra en la Figura @ref(fig:shapes).

3.3.1. Ejercicios

1. ¿Qué no va bien en este código? ¿Por qué hay puntos que no son azules?

```
ggplot(data = millas) +
  geom_point(mapping = aes(x = cilindrada, y = autopista, color = "blue"))
```

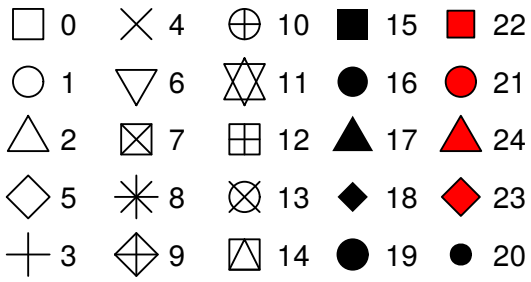
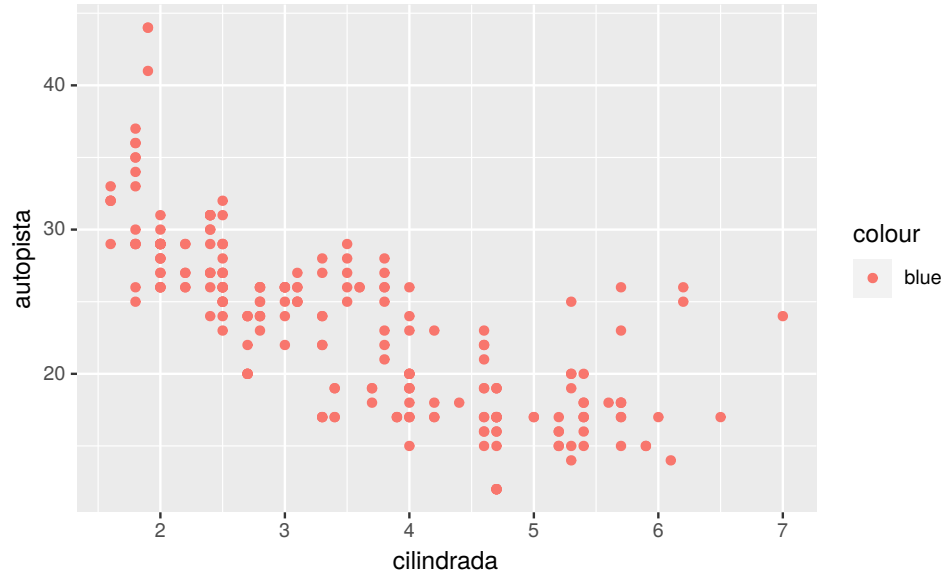


Figura 3.1: R tiene 25 formas predefinidas que están identificadas por números. Hay algunas que parecen duplicados: por ejemplo 0, 15 y 22 son todos cuadrados. La diferencia viene de la interacción entre las estéticas 'color' y 'fill' (*relleno*). Las formas vacías (0–14) tienen un borde determinado por 'color'; las formas sólidas (15–18) están rellenas con 'color'; las formas rellenas (21–24) tienen un borde de 'color' y están rellenas por 'fill'.



2. ¿Qué variables en `millas` son categóricas? ¿Qué variables son continuas? (Pista: escribe `?millas` para leer la documentación de ayuda para este conjunto de datos). ¿Cómo puedes ver esta información cuando ejecutas `millas`?

3. Asigna una variable continua a `color`, `size`, y `shape`. ¿Cómo se comportan estas estéticas de manera diferente para variables categóricas y variables continuas?

4. ¿Qué ocurre si asignas o mapeas la misma variable a múltiples estéticas?

5. ¿Qué hace la estética `stroke`? ¿Con qué formas trabaja? (Pista: consulta `?geom_point`)

6. ¿Qué ocurre si se asigna o mapea una estética a algo diferente del nombre de una variable, como `aes(color = cilindrada < 5)`?

3.4. Problemas comunes

A medida que empieces a escribir código en R, lo más probable es que te encuentres con problemas. No te preocupes, es lo más común. Hemos estado escribiendo código en R durante años, ¡y todos los días seguimos escribiendo código que no funciona!

Comienza comparando cuidadosamente el código que estás ejecutando con el código en este libro. R es extremadamente exigente y un carácter fuera de lugar puede marcar la diferencia. Asegúrate de que cada (coincida con un) y cada " esté emparejado con otro". Algunas veces ejecutarás el código y no pasará nada. Comprueba la parte izquierda de tu consola: si es un +, significa que R no cree que hayas escrito una expresión completa y está esperando que la termines. En este caso, normalmente es más fácil comenzar de nuevo desde cero presionando ESCAPE (la tecla *esc*) para cancelar el procesamiento del comando actual.

Un problema común al crear gráficos con **ggplot2** es colocar el + en el lugar equivocado: debe ubicarse al final de la línea, no al inicio. En otras palabras, asegúrate de no haber escrito accidentalmente un código como este:

```
ggplot(data = millas)
+ geom_point(mapping = aes(x = cilindrada, y = autopista))
```

Si esto no resuelve el problema, prueba con la ayuda. Puedes obtener ayuda sobre cualquier función R ejecutando `?nombre_de_la_funcion` en la consola o seleccionando el nombre de la función y presionando F1 en RStudio. No te preocupes si la ayuda no te parece tan útil, trata entonces de saltar a los ejemplos y buscar un pedazo de código que coincida con lo que intentas hacer.

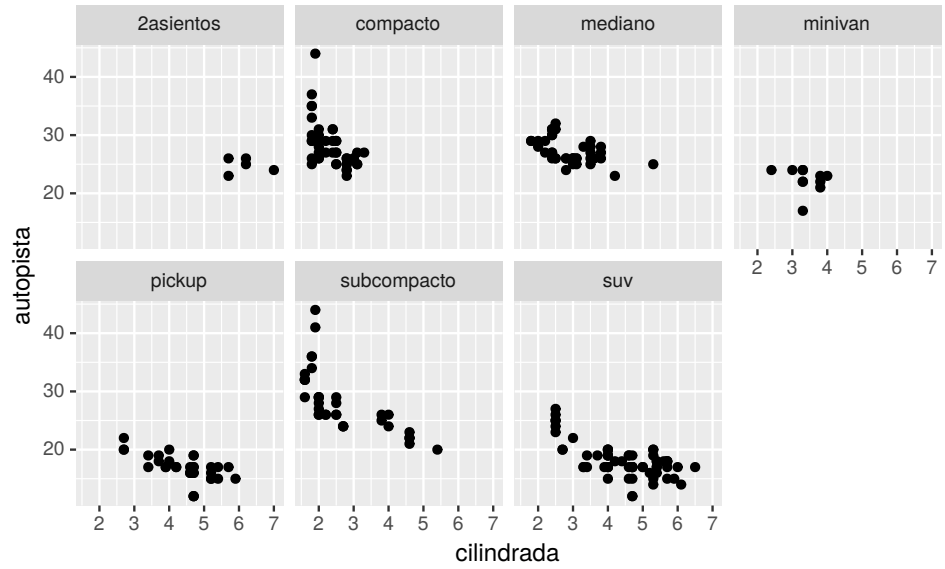
Si eso no ayuda, lee cuidadosamente el mensaje de error. ¡A veces la respuesta estará oculta allí! Sin embargo, cuando recién comienzas en R, puede que la respuesta esté en el mensaje de error, pero aún no sabes cómo entenderlo. Otra gran herramienta es Google: intenta buscar allí el mensaje de error, ya que es probable que otra persona haya tenido el mismo problema y haya obtenido ayuda en línea.

3.5. Separar en facetas

Una forma de agregar variables adicionales es con las estéticas. Otra forma particularmente útil para las variables categóricas consiste en dividir el gráfico en **facetas**, es decir, sub-gráficos que muestran cada uno un subconjunto de los datos.

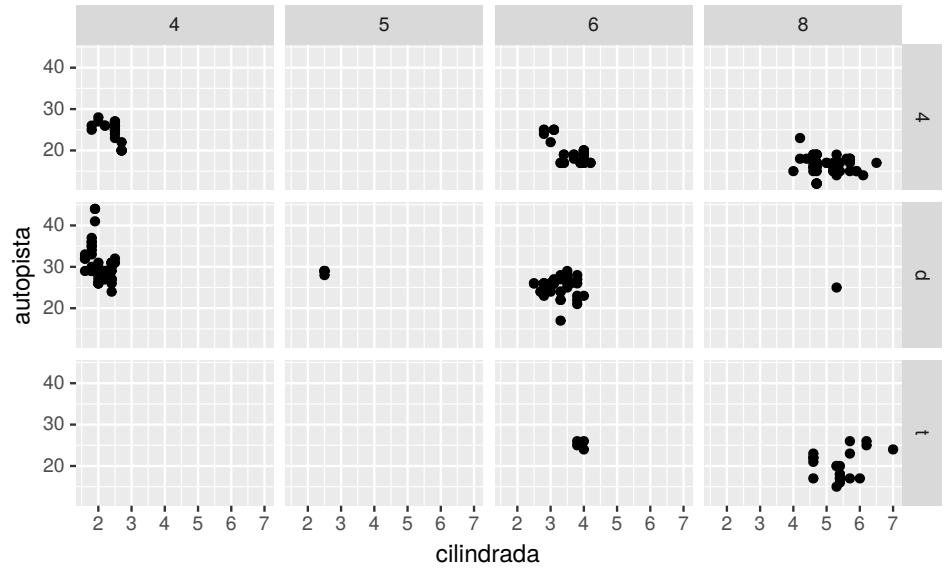
Para separar en facetas un gráfico según una sola variable, utiliza `facet_wrap()` (del inglés *envolver una faceta*). El primer argumento de `facet_wrap()` debería ser una fórmula creada con ~ seguido del nombre de una de las variable (aquí "fórmula" es el nombre de un tipo de estructura en R, no un sinónimo de "ecuación"). La variable que uses en `facet_wrap()` debe ser categórica.

```
ggplot(data = millas) +
  geom_point(mapping = aes(x = cilindrada, y = autopista)) +
  facet_wrap(~ clase, nrow = 2)
```



Para separar en facetas un gráfico según las combinaciones de dos variables, agrega `facet_grid()` a tu código del gráfico (*grid* quiere decir cuadrícula en inglés). El primer argumento de `facet_grid()` también corresponde a una fórmula. Esta vez, la fórmula debe contener dos nombres de variables separados por un ~.

```
ggplot(data = millas) +
  geom_point(mapping = aes(x = cilindrada, y = autopista)) +
  facet_grid(traccion ~ cilindros)
```



Si prefieres no separar en facetas las filas o columnas, remplaza por un `.` el nombre de alguna de las variables, por ejemplo `+ facet_grid(. ~ cilindros)`.

3.5.1. Ejercicios

1. ¿Qué ocurre si intentas separar en facetas una variable continua?

2. ¿Qué significan las celdas vacías que aparecen en el gráfico generado usando `facet_grid(traccion ~ cilindros)`? ¿Cómo se relacionan con este gráfico?

```
ggplot(data = millas) +
  geom_point(mapping = aes(x = traccion, y = cilindros))
```

3. ¿Qué grafica el siguiente código? ¿Qué hace `.` ?

```
ggplot(data = millas) +
  geom_point(mapping = aes(x = cilindrada, y = autopista)) +
  facet_grid(traccion ~ .)

ggplot(data = millas) +
  geom_point(mapping = aes(x = cilindrada, y = autopista)) +
  facet_grid(. ~ cilindros)
```

4. Mira de nuevo el primer gráfico en facetas presentado en esta sección:

```
ggplot(data = millas) +
  geom_point(mapping = aes(x = cilindrada, y = autopista)) +
  facet_wrap(~ clase, nrow = 2)
```

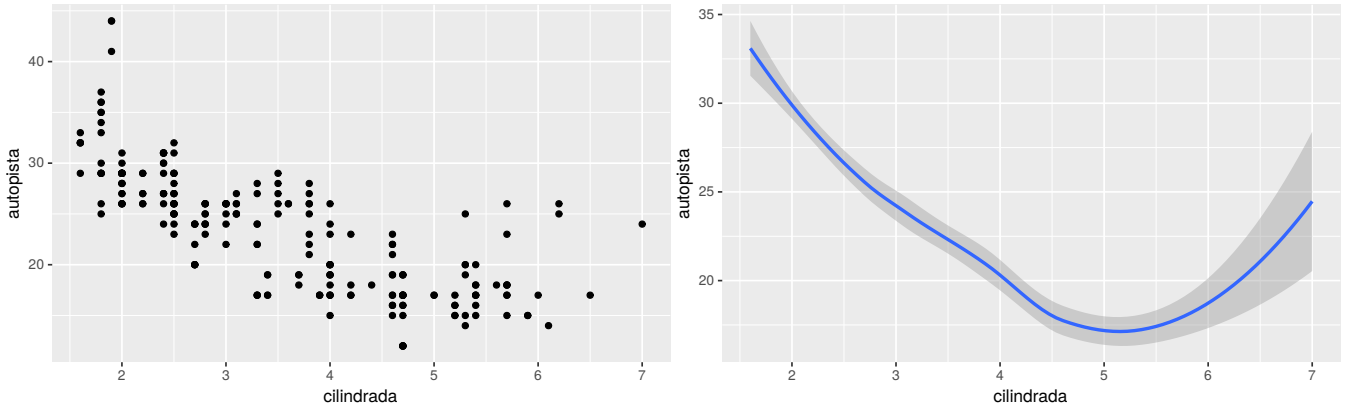
   ¿Cuáles son las ventajas de separar en facetas en lugar de aplicar una estética de color? ¿Cuáles son las desventajas? ¿Cómo cambiaría este balance si tuvieras un conjunto de datos más grande?

5. Lee `?facet_wrap`. ¿Qué hace `nrow`? ¿Qué hace `ncol`? ¿Qué otras opciones controlan el diseño de los paneles individuales? ¿Por qué `facet_grid()` no tiene argumentos `nrow` y `ncol`?

6. Cuando usas `facet_grid()`, generalmente deberías poner la variable con un mayor número de niveles únicos en las columnas. ¿Por qué?

3.6. Objetos geométricos

¿En qué sentido estos dos gráficos son similares?



Ambos gráficos contienen las mismas variables x e y, y ambos describen los mismos datos. Pero los gráficos no son idénticos. Cada uno utiliza un objeto visual diferente para representar los datos. En la sintaxis de **ggplot2**, decimos que usan diferentes **geoms**.

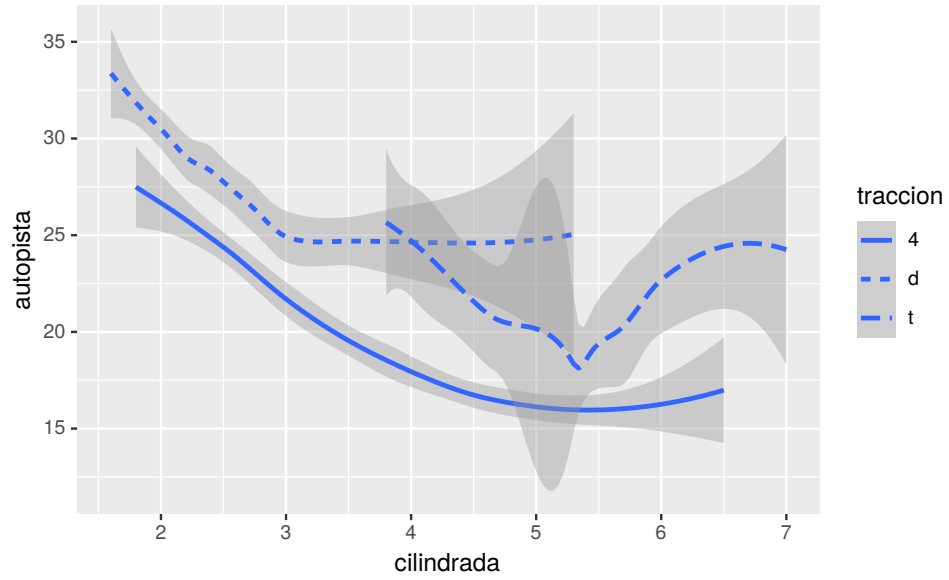
Un **geom** es el objeto geométrico usado para representar datos de forma gráfica. La gente a menudo llama a los gráficos por el tipo de geom que utiliza. Por ejemplo, los diagramas de barras usan geoms de barra (*bar*), los diagramas de líneas usan geoms de línea (*line*), los diagramas de caja usan geoms de diagrama de caja (*boxplot*), y así sucesivamente. En inglés, los diagramas de puntos (llamados *scatterplots*) rompen la tendencia; ellos usan geom de punto (o *point*). Como vemos arriba, puedes usar diferentes geoms para graficar los mismos datos. La gráfica de la izquierda usa el geom de punto (`geom_point()`), y la gráfica de la derecha usa el geom suavizado (`geom_smooth()`), una línea suavizada ajustada a los datos.

Para cambiar el geom de tu gráfico, modifica la función geom que acompaña a `ggplot()`. Por ejemplo, para hacer los gráficos que se muestran arriba, puedes usar este código:

```
# izquierda
ggplot(data = millas) +
  geom_point(mapping = aes(x = cilindrada, y = autopista))

# derecha
ggplot(data = millas) +
  geom_smooth(mapping = aes(x = cilindrada, y = autopista))
```

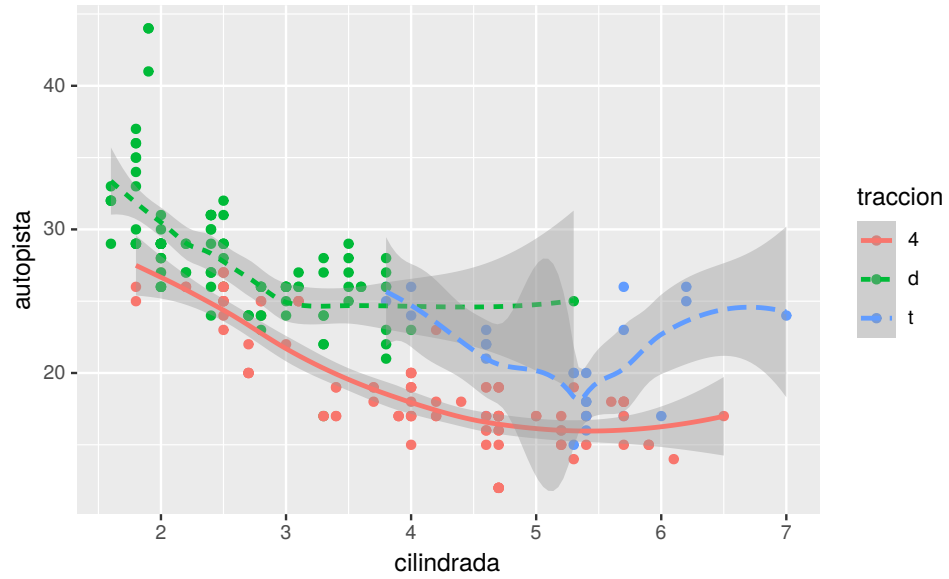
Cada función geom en **ggplot2** toma un argumento de `mapping`. Sin embargo, no todas las estéticas funcionan con todos los geom. Puedes establecer la forma para un punto, pero no puedes establecer la "forma" de una línea. Por otro lado, para una línea *podrías* elegir el *tipo* de línea (*linetype*). `geom_smooth()` dibujará una línea diferente, con un tipo de línea distinto (*linetype*), para cada valor único de la variable que asignes al tipo de línea (*linetype*).

```
ggplot(data = millas) +
  geom_smooth(mapping = aes(x = cilindrada, y = autopista, linetype = traccion))
```



Aquí `geom_smooth()` separa los automóviles en tres líneas en función de su valor de `traccion`, que describe el tipo de transmisión de un automóvil. Una línea describe todos los puntos con un valor de `4`, otra línea los de valor `d`, y una tercera línea describe los puntos con un valor `t`. Aquí, `4` significa tracción en las cuatro ruedas, `d` tracción delantera y `t` tracción trasera.

Si esto suena extraño, podemos hacerlo más claro al superponer las líneas sobre los datos brutos y luego colorear todo según `traccion`.



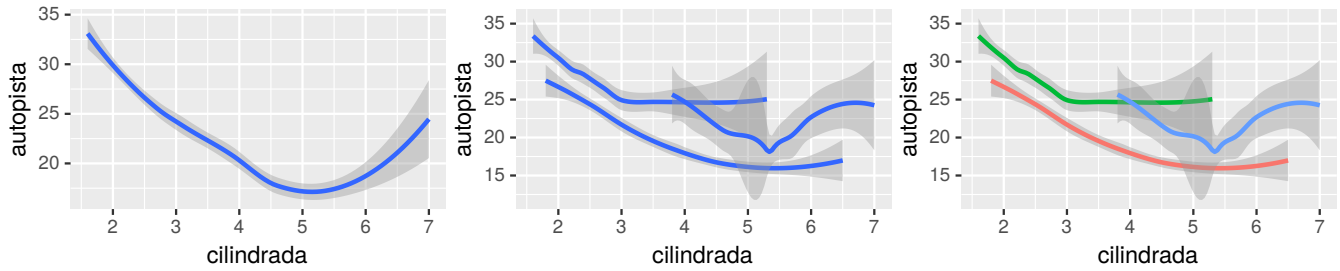
¡Observa que generamos un gráfico que contiene dos geoms! Si esto te emociona, abróchate el cinturón. En la siguiente sección aprenderemos cómo colocar múltiples geoms en el mismo gráfico.

ggplot2 proporciona más de 40 geoms y los paquetes de extensión proporcionan aún más (consulta https://exts.ggplot2.tidyverse.org/gallery/ para obtener una muestra). La mejor forma de obtener un panorama completo sobre las posibilidades que brinda **ggplot2** es consultando la hoja de referencia (o *cheatsheet*), que puedes encontrar en https://rstudio.com/resources/cheatsheets/. Para obtener

más información sobre un tipo dado de geoms, usa la ayuda: `?geom_smooth`.

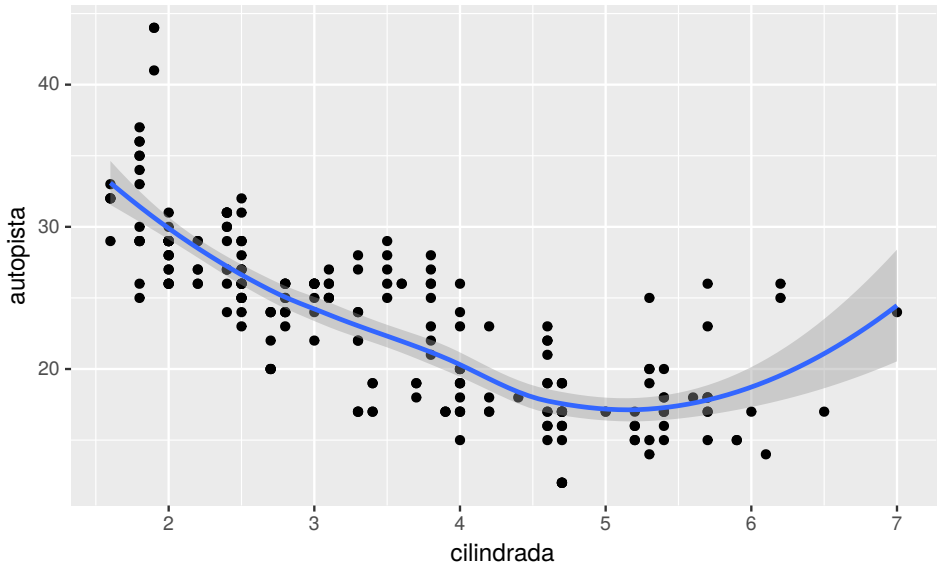
Muchos geoms, como `geom_smooth()`, usan un único objeto geométrico para mostrar múltiples filas de datos. Con estos geoms, puedes asignar la estética de `group` (**grupo**) a una variable categórica para graficar múltiples objetos. **ggplot2** representará un objeto distinto por cada valor único de la variable de agrupamiento. En la práctica, **ggplot2** agrupará automáticamente los datos para estos geoms siempre que se asigne una estética a una variable discreta (como en el ejemplo del tipo de línea o `linetype`). Es conveniente confiar en esta característica porque la estética del grupo en sí misma no agrega una leyenda o características distintivas a los geoms.

```
ggplot(data = millas) +
  geom_smooth(mapping = aes(x = cilindrada, y = autopista))

ggplot(data = millas) +
  geom_smooth(mapping = aes(x = cilindrada, y = autopista, group = traccion))

ggplot(data = millas) +
  geom_smooth(
    mapping = aes(x = cilindrada, y = autopista, color = traccion),
    show.legend = FALSE
    )
```



Para mostrar múltiples geoms en el mismo gráfico, agrega varias funciones geom a `ggplot()`:

```
ggplot(data = millas) +
 geom_point(mapping = aes(x = cilindrada, y = autopista)) +
 geom_smooth(mapping = aes(x = cilindrada, y = autopista))
```

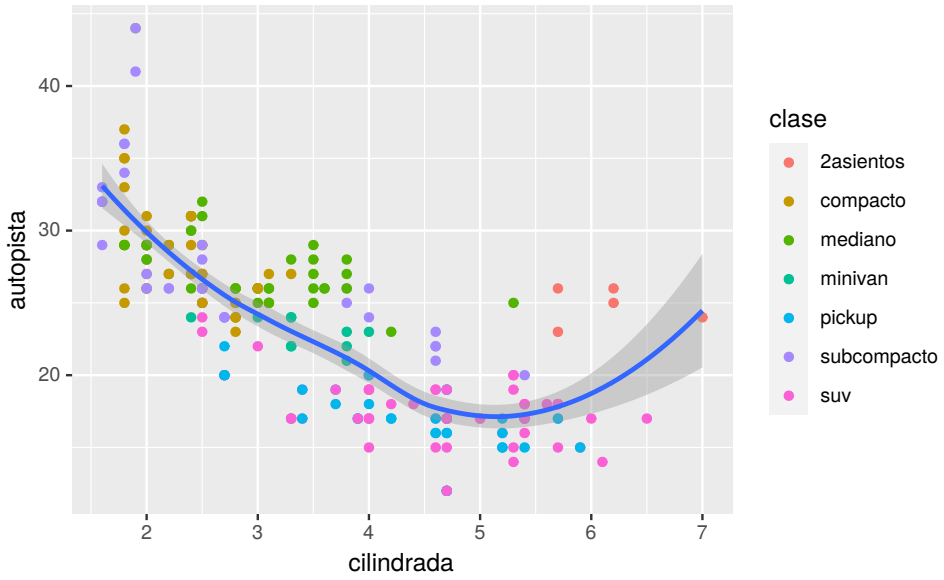


Esto introduce, sin embargo, cierta duplicación en nuestro código. Imagina que deseas cambiar el eje y para mostrar `cilindrada` en lugar de `autopista`. Necesitarías cambiar la variable en dos lugares y podrías olvidarte de actualizar uno. Puedes evitar este tipo de repetición pasando un conjunto de mapeos a `ggplot()`. **ggplot2** tratará estos mapeos como mapeos globales que se aplican a cada geom en el gráfico. En otras palabras, este código producirá la misma gráfica que el código anterior:

```
ggplot(data = millas, mapping = aes(x = cilindrada, y = autopista)) +
  geom_point() +
  geom_smooth()
```

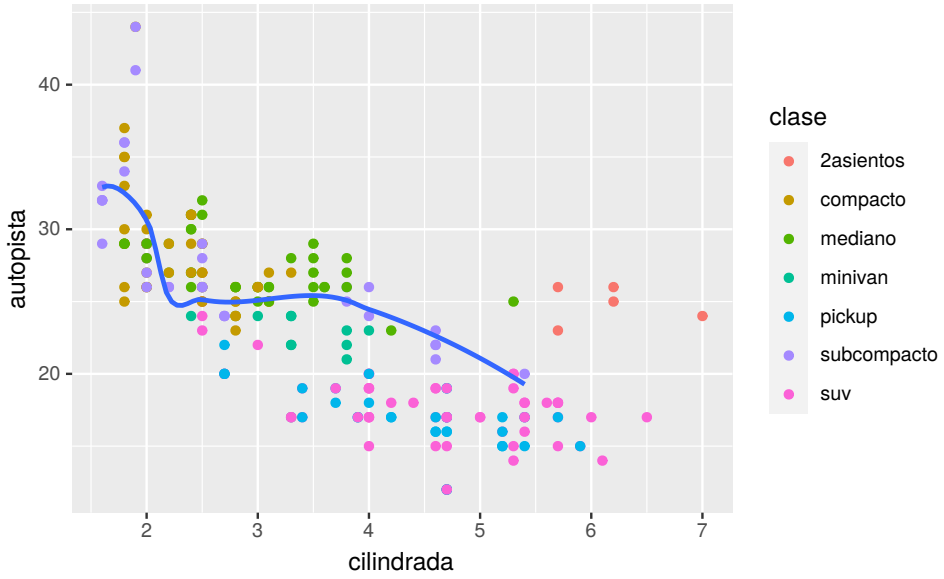
Si colocas mapeos en una función geom, ggplot2 los tratará como mapeos locales para la capa. Estas asignaciones serán usadas para extender o sobrescribir los mapeos globales *solo para esa capa*. Esto permite mostrar diferentes estéticas en diferentes capas.

```
ggplot(data = millas, mapping = aes(x = cilindrada, y = autopista)) +
  geom_point(mapping = aes(color = clase)) +
  geom_smooth()
```



La misma idea se puede emplear para especificar distintos conjuntos de datos (`data`) para cada capa. Aquí, nuestra línea suave muestra solo un subconjunto del conjunto de datos de `millas`: los autos subcompactos. El argumento local de datos en `geom_smooth()` anula el argumento de datos globales en `ggplot()` solo para esa capa.

```
ggplot(data = millas, mapping = aes(x = cilindrada, y = autopista)) +
  geom_point(mapping = aes(color = clase)) +
  geom_smooth(data = filter(millas, clase == "subcompacto"), se = FALSE)
```



(Aprenderás cómo funciona `filter()` en el próximo capítulo: por ahora, solo recuerda que este comando selecciona los automóviles subcompactos).

3.6.1. Ejercicios

1. ¿Qué geom usarías para generar un gráfico de líneas? ¿Y para un diagrama de caja? ¿Y para un histograma? ¿Y para un gráfico de área?

2. Ejecuta este código en tu mente y predice cómo se verá el *output*. Luego, ejecuta el código en R y verifica tus predicciones.

```
ggplot(data = millas, mapping = aes(x = cilindrada, y = autopista, color = tracci
  geom_point() +
  geom_smooth(se = FALSE)
```

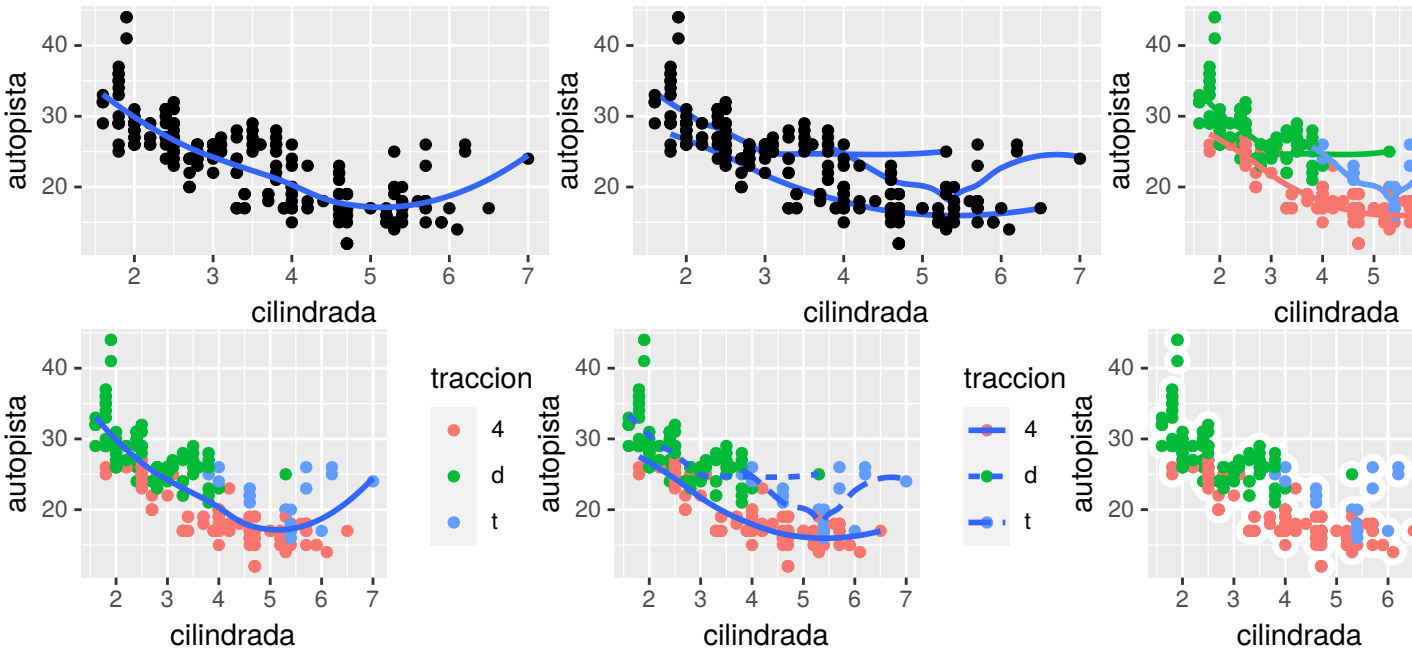
3. ¿Qué muestra `show.legend = FALSE`? ¿Qué pasa si lo quitas? ¿Por qué crees que lo utilizamos antes en el capítulo?

4. ¿Qué hace el argumento `se` en `geom_smooth()`?

5. ¿Se verán distintos estos gráficos? ¿Por qué sí o por qué no?

```
ggplot(data = millas, mapping = aes(x = cilindrada, y = autopista)) +
  geom_point() +
  geom_smooth()

ggplot() +
  geom_point(data = millas, mapping = aes(x = cilindrada, y = autopista)) +
  geom_smooth(data = millas, mapping = aes(x = cilindrada, y = autopista))
```

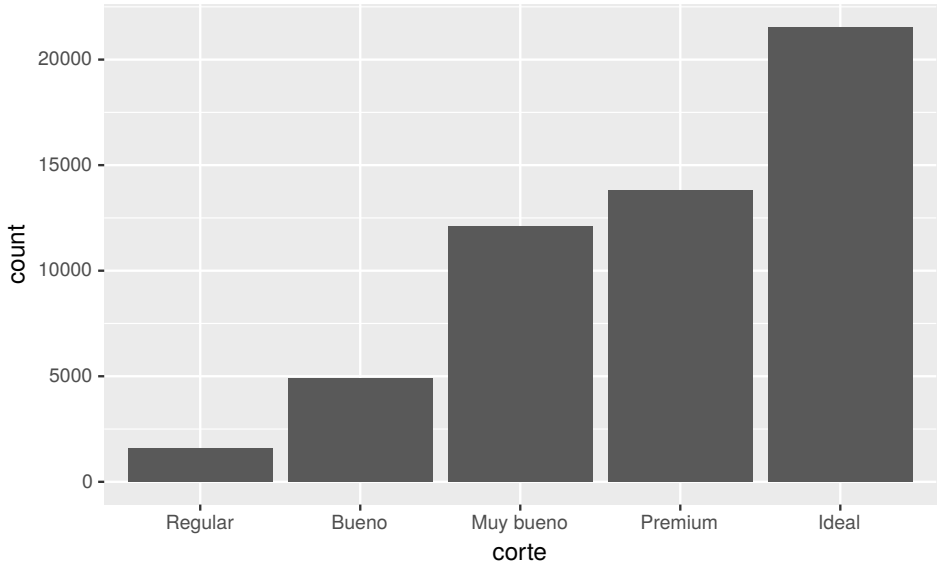
6. Recrea el código R necesario para generar los siguientes gráficos:



3.7. Transformaciones estadísticas

A continuación, echemos un vistazo a un gráfico de barras. Los gráficos de barras parecen simples, pero son interesantes porque revelan algo sutil sobre los gráficos. Considera un gráfico de barras básico, como uno realizado con `geom_bar()`. El siguiente gráfico muestra la cantidad total de diamantes en el conjunto de datos `diamantes`, agrupados por la variable `corte`. El conjunto de datos `diamantes` se encuentra en el paquete **datos** y contiene información sobre ~ 54000 diamantes, incluido el `precio`, el `quilate`, el `color`, la `claridad` y el `corte` de cada uno. El gráfico muestra que hay más diamantes disponibles con cortes de alta calidad que con cortes de baja calidad.

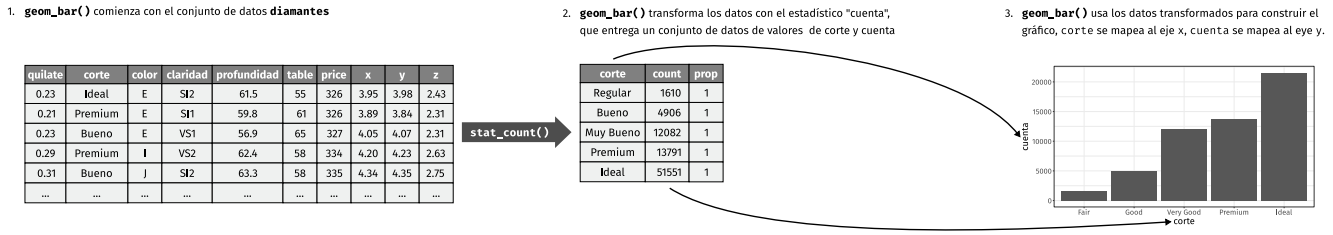
```
ggplot(data = diamantes) +
  geom_bar(mapping = aes(x = corte))
```



En el eje x, el gráfico muestra `corte`, una variable de `diamantes`. En el eje y muestra 'recuento' (*count*), ¡pero el recuento no es una variable en `diamantes`! ¿De dónde viene? Muchos gráficos, como los diagramas de dispersión (*scatterplots*), grafican los valores brutos de un conjunto de datos. Otros gráficos, como los de barras, calculan nuevos valores para presentar:

- los gráficos de barras, los histogramas y los polígonos de frecuencia almacenan los datos y luego grafican los conteos por contenedores, es decir, el número de puntos que caen en cada contenedor.
- los gráficos de líneas suavizadas (*smoothers*) ajustan un modelo a los datos y luego grafican las predicciones del modelo.
- los diagramas de caja (*boxplots*) calculan un resumen robusto de la distribución y luego muestran una caja con formato especial.

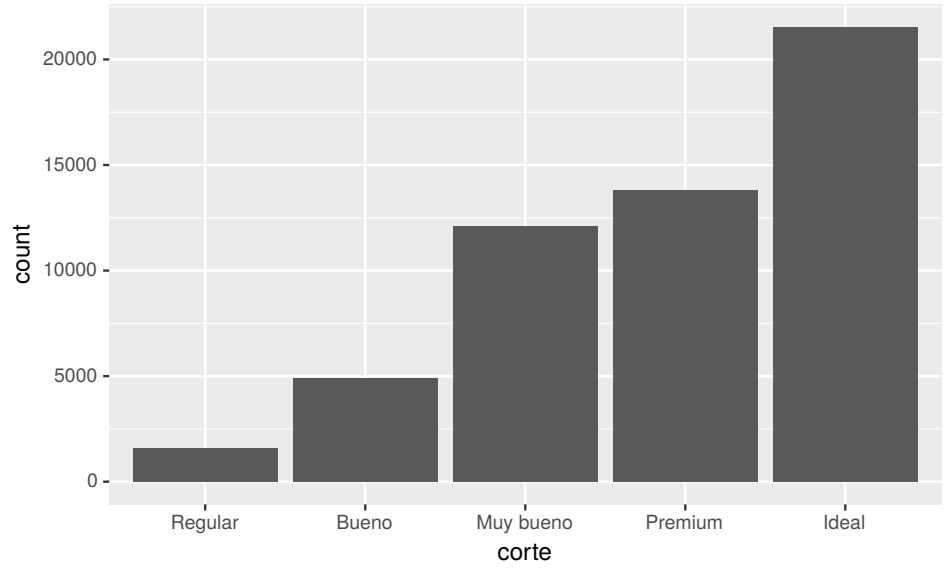
El algoritmo utilizado para calcular nuevos valores para un gráfico se llama *stat*, abreviatura en inglés de transformación estadística (***stat****istical transformation*). La siguiente figura describe cómo funciona este proceso con `geom_bar()`.



Puedes aprender acerca de qué *stat* usa cada geom inspeccionando el valor predeterminado para el argumento `stat`. Por ejemplo, `?geom_bar` muestra que el valor predeterminado para `stat` es "count", lo que significa que `geom_bar()` usa `stat_count()`. `stat_count()` está documentado en la misma página que `geom_bar()`, y si te desplazas hacia abajo puedes encontrar una sección llamada "Computed variables" (*Variables calculadas*). Ahí se describe cómo calcula dos nuevas variables: `count` y `prop`.

Por lo general, puedes usar geoms y estadísticas de forma intercambiable. Por ejemplo, puedes volver a crear la gráfica anterior usando `stat_count()` en lugar de `geom_bar()`:

```
ggplot(data = diamantes) +
  stat_count(mapping = aes(x = corte))
```



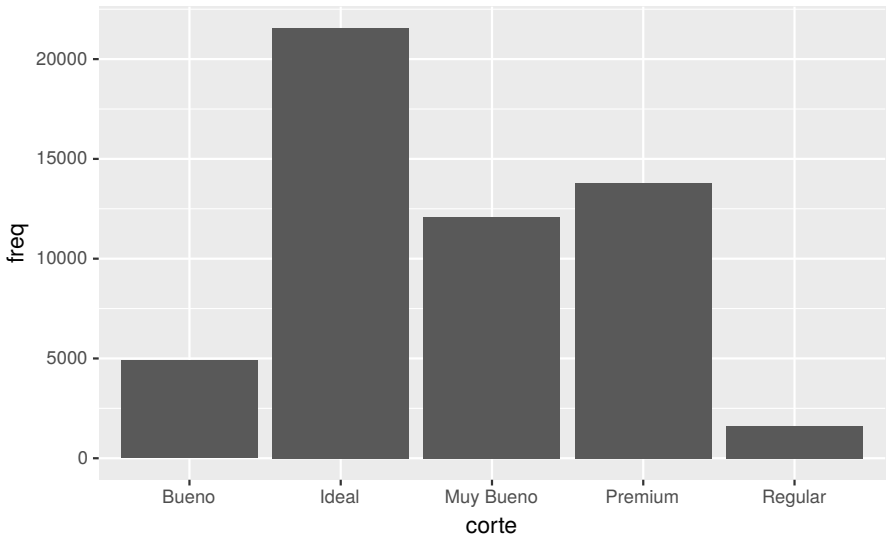
Esto funciona porque cada geom tiene una estadística predeterminada, y cada estadística tiene un geom predeterminado. Esto significa que generalmente puedes usar geoms sin preocuparte por la transformación estadística subyacente.

Hay tres razones por las que podrías necesitar usar una estadística explícitamente:

1. Es posible que desees anular la estadística predeterminada. En el siguiente código, cambiamos en `geom_bar()` la estadística recuento ("count", el valor predeterminado) a identidad ("identity"). Esto nos permite asignar la altura de las barras a los valores brutos de una variable y. Desafortunadamente, cuando las personas hablan de gráficos de barras de manera informal, podrían estar refiriéndose a este tipo de gráfico de barras, en el que la altura de la barra ya está presente en los datos, o bien, al gráfico de barras anterior, en el que la altura de la barra se

determina contando filas.

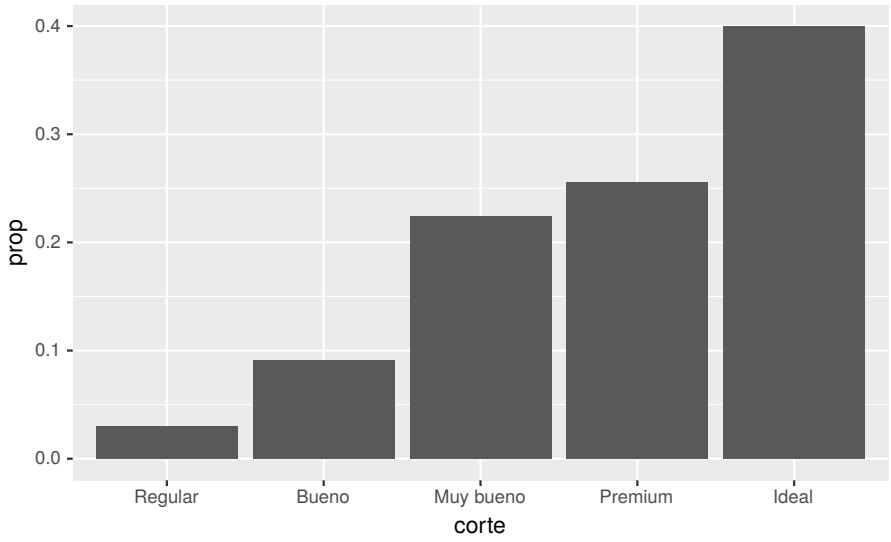
```
demo <- tribble(
  ~corte,      ~freq,
  "Regular",    1610,
  "Bueno",      4906,
  "Muy Bueno", 12082,
  "Premium",    13791,
  "Ideal",      21551
)

ggplot(data = demo) +
  geom_bar(mapping = aes(x = corte, y = freq), stat = "identity")
```



(No te preocupes si nunca has visto `<-` o `tribble()`. Puede que seas capaz de adivinar su significado por el contexto y ¡pronto aprenderás qué es lo que hacen exactamente!)

2. Es posible que desees anular el mapeo predeterminado de las variables transformadas a las estéticas. Por ejemplo, es posible que desees mostrar un gráfico de barras de proporciones, en lugar de un recuento:

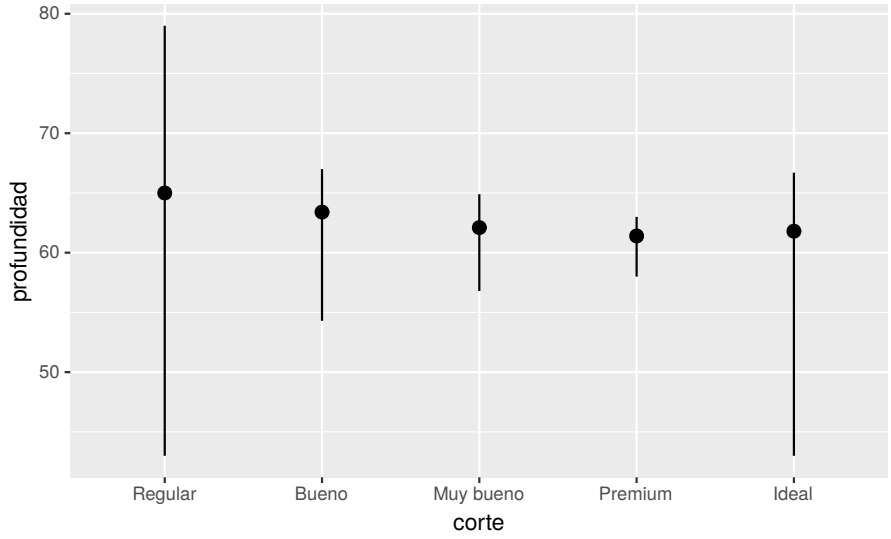
```
ggplot(data = diamantes) +
  geom_bar(mapping = aes(x = corte, y = stat(prop), group = 1))
```



Para encontrar las variables calculadas por *stat*, busca la sección de ayuda titulada "Compute Variables".

3. Es posible que desees resaltar la transformación estadística en tu código. Por ejemplo, puedes usar `stat_summary()`, que resume los valores de y para cada valor único de x, para así resaltar el resumen que se está computando:

```
ggplot(data = diamantes) +
  stat_summary(
    mapping = aes(x = corte, y = profundidad),
    fun.min = min,
    fun.max = max,
    fun = median
  )
```



ggplot2 proporciona más de 20 transformaciones estadísticas para que uses. Cada *stat* es una función, por lo que puedes obtener ayuda de la manera habitual, por ejemplo: `?stat_bin`. Para ver una lista

completa de transformaciones estadísticas disponibles para **ggplot2**, consulta la hoja de referencia.

3.7.1. Ejercicios

1. ¿Cuál es el geom predeterminado asociado con `stat_summary()`? ¿Cómo podrías reescribir el gráfico anterior para usar esa función geom en lugar de la función stat?

2. ¿Qué hace `geom_col()`? ¿En qué se diferencia de `geom_bar()`?

3. La mayoría de los geoms y las transformaciones estadísticas vienen en pares que casi siempre se usan en conjunto. Lee la documentación y haz una lista de todos los pares. ¿Qué tienen en común?

4. ¿Qué variables calcula `stat_smooth()`? ¿Qué parámetros controlan su comportamiento?

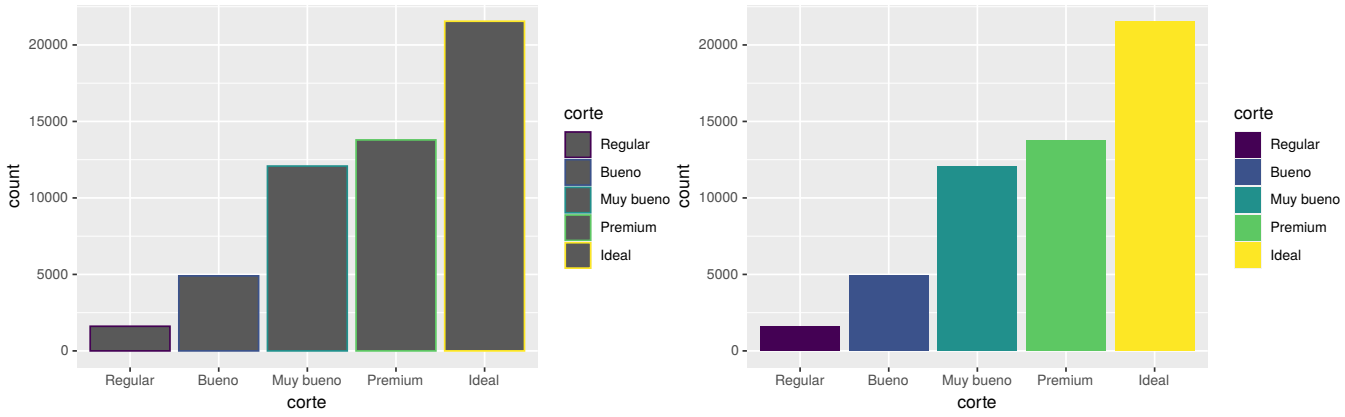
5. En nuestro gráfico de barras de proporción necesitamos establecer `group = 1`. ¿Por qué? En otras palabras, ¿cuál es el problema con estos dos gráficos?

```
ggplot(data = diamantes) +
  geom_bar(mapping = aes(x = corte, y = ..prop..))

ggplot(data = diamantes) +
  geom_bar(mapping = aes(x = corte, fill = color, y = ..prop..))
```

3.8. Ajustes de posición

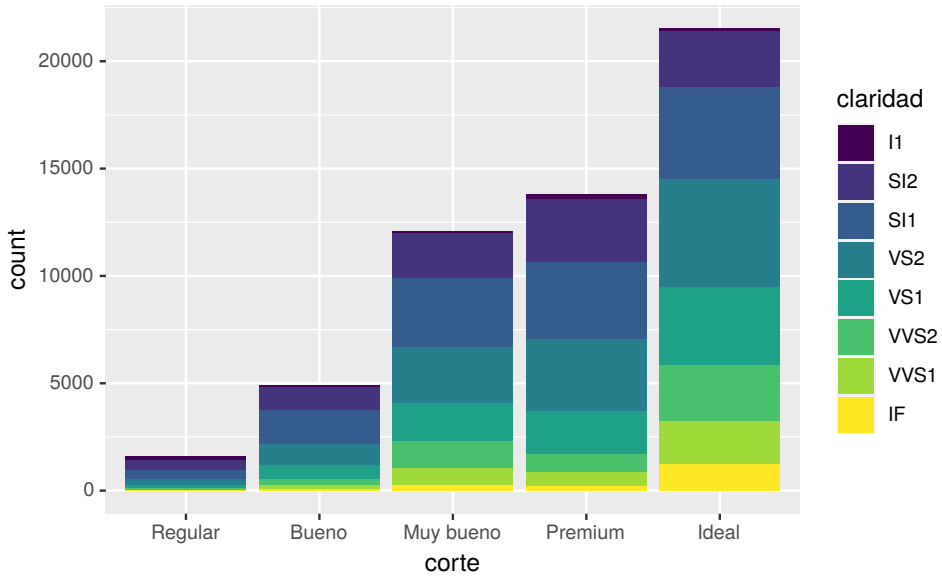
Hay una pieza más de magia asociada con los gráficos de barras. Puedes colorear un gráfico de barras usando tanto la estética de `color` como la más útil `fill` (*relleno*):

```
ggplot(data = diamantes) +
  geom_bar(mapping = aes(x = corte, colour = corte))

ggplot(data = diamantes) +
  geom_bar(mapping = aes(x = corte, fill = corte))
```



Mira lo que sucede si asignas la estética de relleno (*fill*) a otra variable, como `claridad`: las barras se apilan automáticamente. Cada rectángulo de color representa una combinación de `corte` y `claridad`.

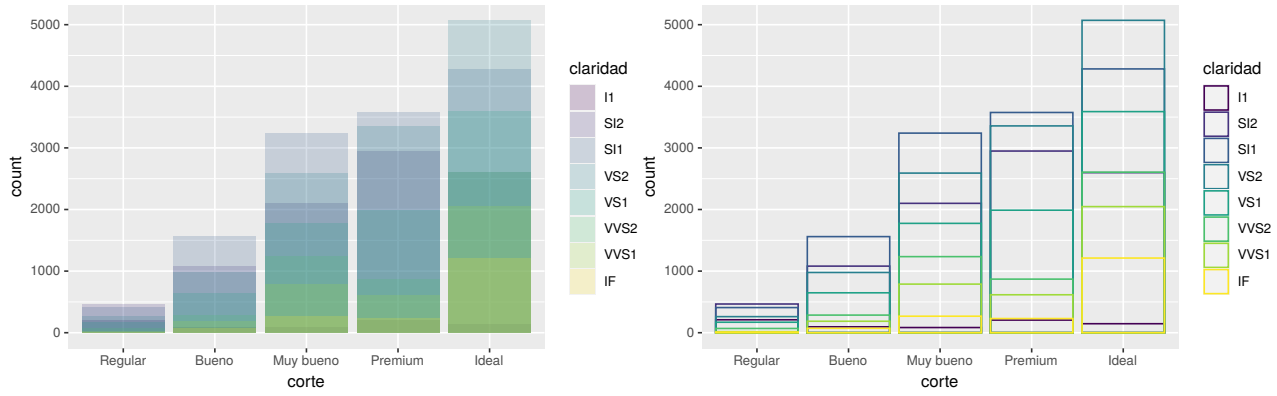
```
ggplot(data = diamantes) +
  geom_bar(mapping = aes(x = corte, fill = claridad))
```



El apilamiento se realiza automáticamente mediante el ajuste de posición especificado por el argumento `position`. Si no deseas un gráfico de barras apiladas (`"stack"`), puedes usar una de las otras tres opciones: `ïdentity"`, `"dodge"` o `"fill"`.

- `position = ïdentity"` colocará cada objeto exactamente donde cae en el contexto del gráfico. Esto no es muy útil al momento de graficar barras, ya que las superpone. Para ver esa superposición, debemos hacer que las barras sean ligeramente transparentes configurando el `alpha` a un valor pequeño, o completamente transparente al establecer `fill = NA`.

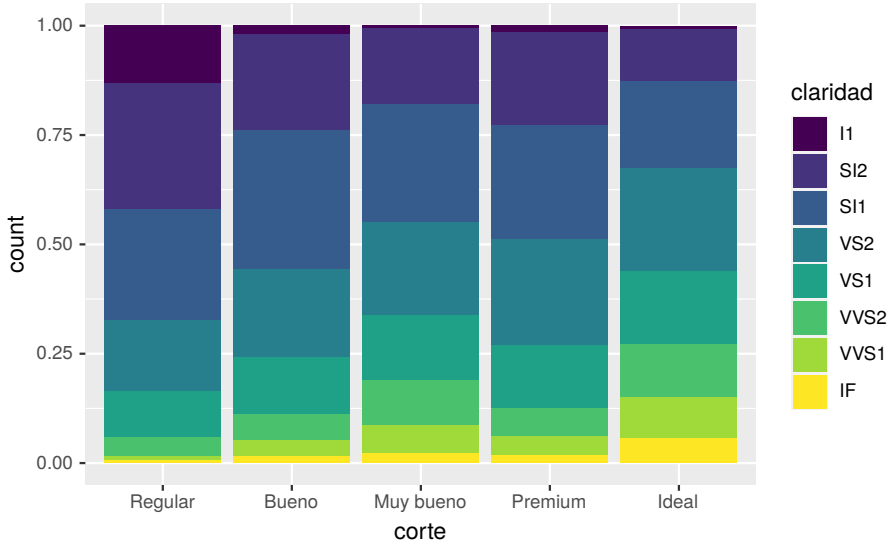
```
ggplot(data = diamantes, mapping = aes(x = corte, fill = claridad)) +
  geom_bar(alpha = 1/5, position = "identity")

ggplot(data = diamantes, mapping = aes(x = corte, colour = claridad)) +
  geom_bar(fill = NA, position = "identity")
```



El ajuste de `position = identity` es más útil para geoms 2D, como puntos, donde es la opción predeterminada.

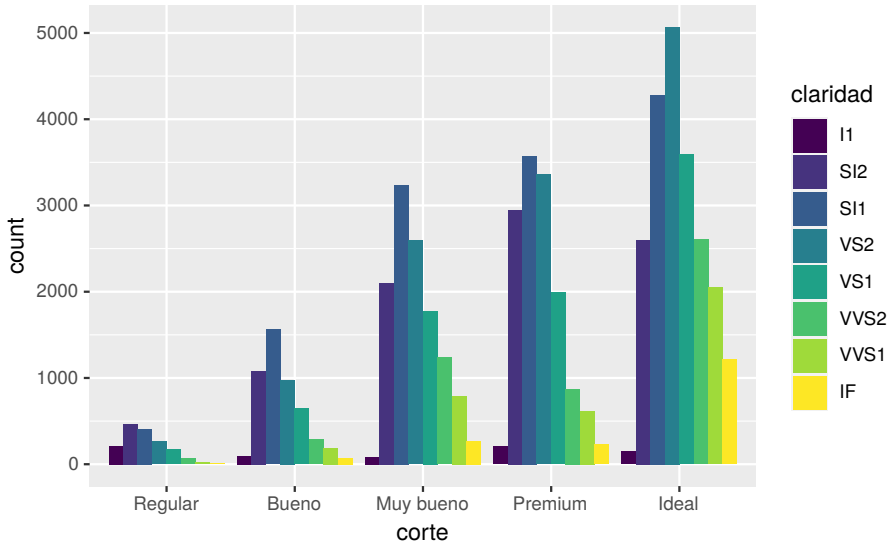
- `position = "fill"` funciona como el apilamiento de `position = "stack"`, pero hace que cada conjunto de barras apiladas tenga la misma altura. Esto hace que sea más fácil comparar proporciones entre grupos.

```
ggplot(data = diamantes) +
  geom_bar(mapping = aes(x = corte, fill = claridad), position = "fill")
```

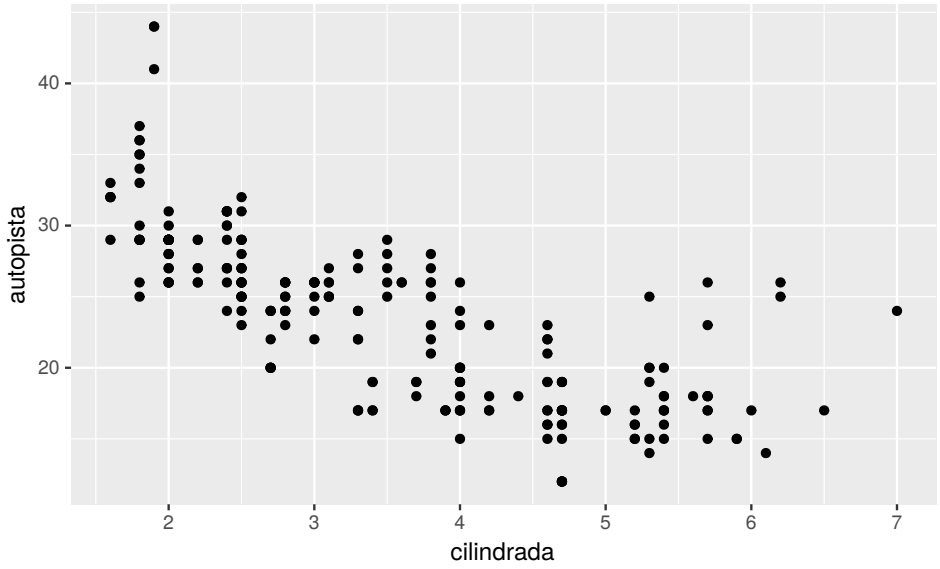


- `position = "dodge"` coloca los objetos superpuestos uno al lado del otro. Esto hace que sea más fácil comparar valores individuales.

```
ggplot(data = diamantes) +
  geom_bar(mapping = aes(x = corte, fill = claridad), position = "dodge")
```



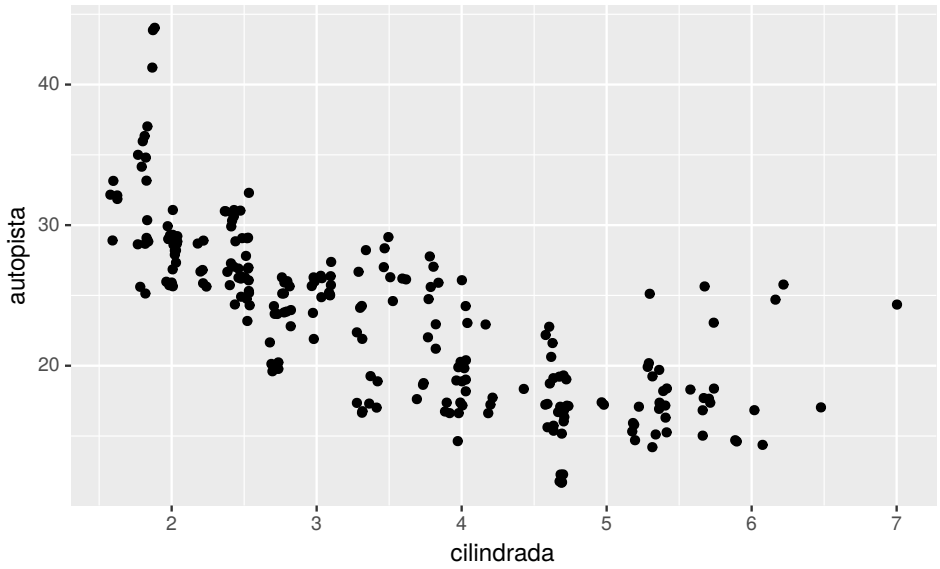
Hay otro tipo de ajuste que no es útil para gráficos de barras, pero que puede ser muy útil para diagramas de dispersión. Recuerda nuestro primer diagrama de dispersión. ¿Notaste que mostraba solo 126 puntos, a pesar de que hay 234 observaciones en el conjunto de datos?



Los valores de las variables `autopista` y `cilindrada` se redondean de modo que los puntos aparecen en una cuadrícula y muchos se superponen entre sí. Este problema se conoce como **solapamiento** (*overplotting*). Esta disposición hace que sea difícil ver dónde está la masa de datos. ¿Los puntos de datos se distribuyen equitativamente a lo largo de la gráfica, o hay una combinación especial de `autopista` y `cilindrada` que contiene 109 valores?

Puedes evitar esto estableciendo el ajuste de posición en "jitter". `position = "jitter"` agrega una pequeña cantidad de ruido aleatorio a cada punto. Esto dispersa los puntos, ya que es poco probable que dos puntos reciban la misma cantidad de ruido aleatorio.

```
ggplot(data = millas) +
  geom_point(mapping = aes(x = cilindrada, y = autopista), position = "jitter")
```



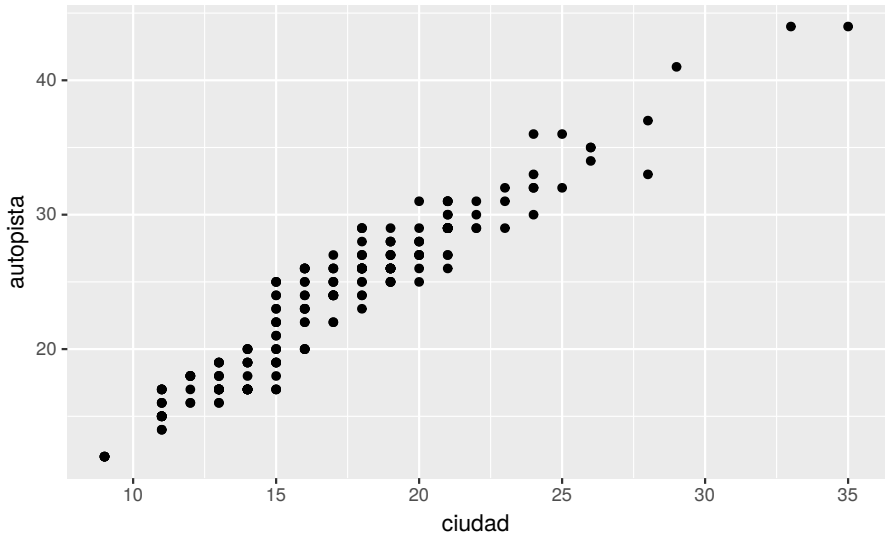
Agregar aleatoriedad a los puntos puede parecer una forma extraña de mejorar tu gráfico. Si bien hace que sea menos preciso a escalas pequeñas, lo hace ser más revelador a gran escala. Como esta es una operación tan útil, ggplot2 incluye una abreviatura de `geom_point(position = "jitter")`: `geom_jitter()`.

Para obtener más información sobre ajustes de posición, busca la página de ayuda asociada con cada ajuste: `?position_dodge`, `?position_fill`, `?position_identity`, `?position_jitter` y `?position_stack`.

3.8.1. Ejercicios

1. ¿Cuál es el problema con este gráfico? ¿Cómo podrías mejorarlo?

```
ggplot(data = millas, mapping = aes(x = ciudad, y = autopista)) +
  geom_point()
```



2. ¿Qué parámetros de `geom_jitter()` controlan la cantidad de ruido?

3. Compara y contrasta `geom_jitter()` con `geom_count()`

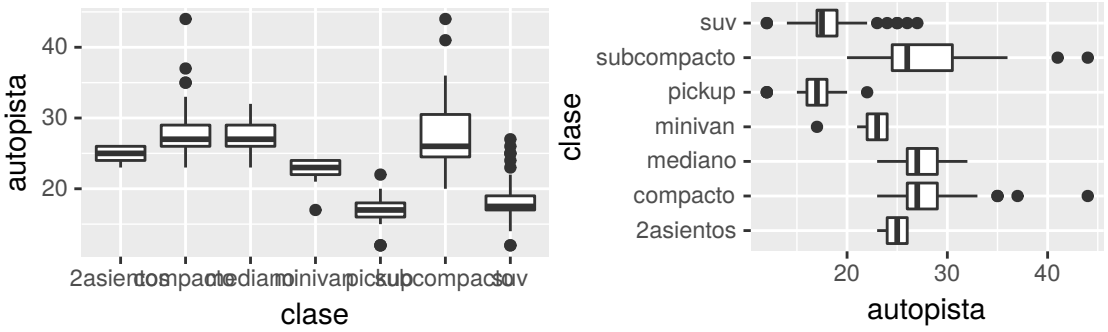
4. ¿Cuál es el ajuste de posición predeterminado de `geom_boxplot()`? Crea una visualización del conjunto de datos de `millas` que lo demuestre.

3.9. Sistemas de coordenadas

Los sistemas de coordenadas son probablemente la parte más complicada de ggplot2. El sistema predeterminado es el sistema de coordenadas cartesianas, donde las posiciones x e y actúan independientemente para determinar la ubicación de cada punto. Hay varios otros sistemas de coordenadas que ocasionalmente son útiles.

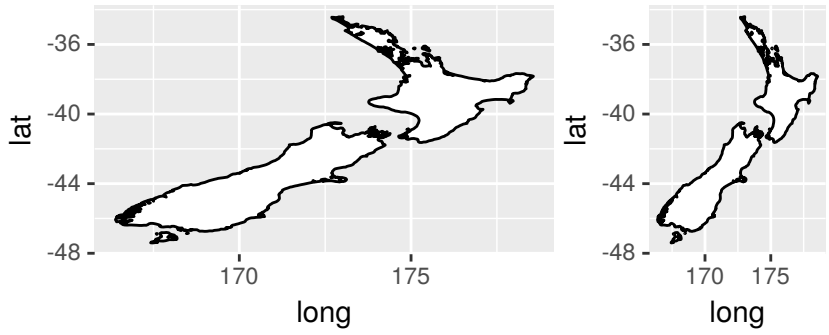
- `coord_flip()` cambia los ejes x e y. Esto es útil, por ejemplo, si quieres diagramas de caja horizontales. También es útil para etiquetas largas: es difícil ajustarlas sin que se superpongan en el eje x.

```
ggplot(data = millas, mapping = aes(x = clase, y = autopista)) +
  geom_boxplot()

ggplot(data = millas, mapping = aes(x = clase, y = autopista)) +
  geom_boxplot() +
  coord_flip()
```



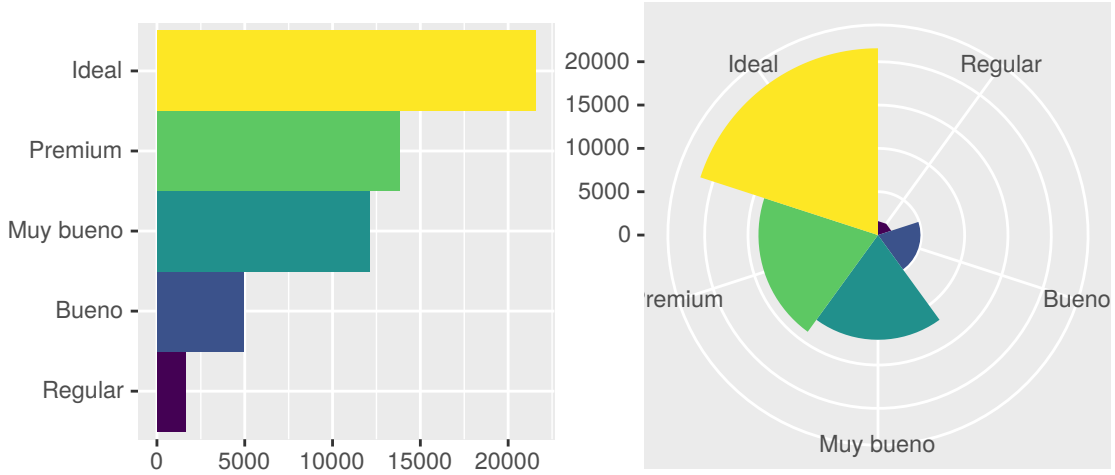
- `coord_quickmap()` establece correctamente la relación de aspecto para los mapas. Esto es muy importante si gráficas datos espaciales con ggplot2 (tema para el que, desafortunadamente, no contamos con espacio para desarrollar en este libro).

```
nz <- map_data("nz")

ggplot(nz, aes(long, lat, group = group)) +
  geom_polygon(fill = "white", colour = "black")

ggplot(nz, aes(long, lat, group = group)) +
  geom_polygon(fill = "white", colour = "black") +
  coord_quickmap()
```



- `coord_polar()` usa coordenadas polares. Las coordenadas polares revelan una conexión interesante entre un gráfico de barras y un gráfico de Coxcomb.

```
bar <- ggplot(data = diamantes) +
  geom_bar(
    mapping = aes(x = corte, fill = corte),
    show.legend = FALSE,
    width = 1
  ) +
  theme(aspect.ratio = 1) +
  labs(x = NULL, y = NULL)

bar + coord_flip()
bar + coord_polar()
```

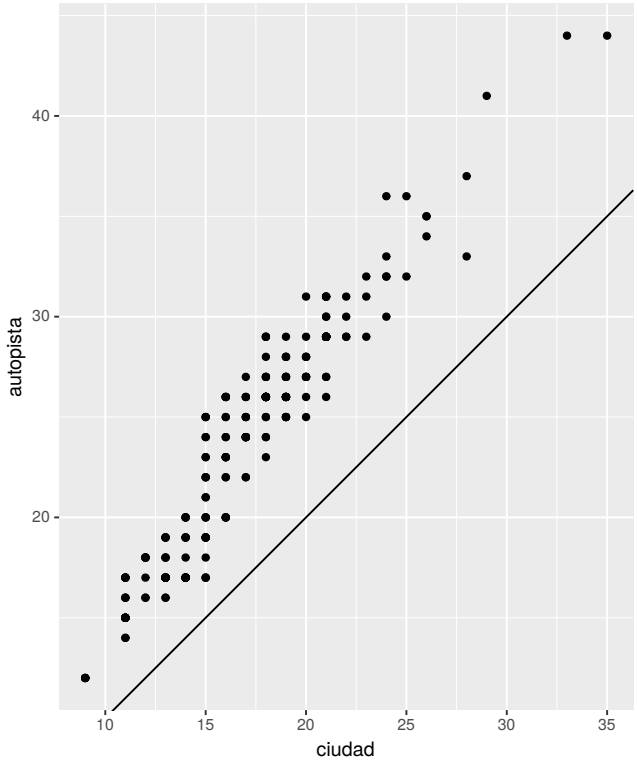


3.9.1. Ejercicios

1. Convierte un gráfico de barras apiladas en un gráfico circular usando `coord_polar()`.
2. ¿Qué hace `labs()`? Lee la documentación.
3. ¿Cuál es la diferencia entre `coord_quickmap()` y `coord_map()`?

4. ¿Qué te dice la gráfica siguiente sobre la relación entre `ciudad` y `autopista`? ¿Por qué es `coord_fixed()` importante? ¿Qué hace `geom_abline()`?

```
ggplot(data = millas, mapping = aes(x = ciudad, y = autopista)) +
  geom_point() +
  geom_abline() +
  coord_fixed()
```



3.10. La gramática de gráficos en capas

En las secciones anteriores aprendiste mucho más que solo hacer diagramas de dispersión, gráficos de barras y diagramas de caja. Aprendiste una base que se puede usar para hacer cualquier tipo de gráfico con **ggplot2**. Para ver esto, agreguemos ajustes de posición, transformaciones estadísticas, sistemas de coordenadas y facetas a nuestra plantilla de código:

```
ggplot(data = <DATOS>) +
 <GEOM_FUNCIÓN>(
   mapping = aes(<MAPEOS>),
   stat = <ESTADÍSTICAS>,
   position = <POSICIÓN>
 ) +
 <FUNCIÓN_COORDENADAS> +
 <FUNCIÓN_FACETAS>
```

Nuestra nueva plantilla tiene siete parámetros que se corresponden con las palabras entre corchetes

que aparecen en la plantilla. En la práctica, rara vez necesitas proporcionar los siete parámetros para hacer un gráfico porque **ggplot2** proporcionará valores predeterminados útiles para todos, excepto para los datos, el mapeo y la función geom.

Los siete parámetros en la plantilla componen la gramática de los gráficos, un sistema formal de construcción de gráficos. La gramática de los gráficos se basa en la idea de que puedes describir de manera única *cualquier* gráfico como una combinación de un conjunto de datos, un geom, un conjunto de mapeos, una estadística, un ajuste de posición, un sistema de coordenadas y un esquema de facetado.

Para ver cómo funciona esto, considera cómo podrías construir un gráfico básico desde cero: podrías comenzar con un conjunto de datos y luego transformarlo en la información que deseas mostrar (con un *stat*).

1. **`geom_bar()`** comienza con el conjunto de datos **`diamantes`**

2. Calcula la cuenta para cada valor de corte con **`stat_count()`**.

quilate	corte	color	claridad	profundidad	table	price	x	y	z
0.23	Ideal	E	SI2	61.5	55	326	3.95	3.98	2.43
0.21	Premium	E	SI1	59.8	61	326	3.89	3.84	2.31
0.23	Bueno	E	VS1	56.9	65	327	4.05	4.07	2.31
0.29	Premium	I	VS2	62.4	58	334	4.20	4.23	2.63
0.31	Bueno	J	SI2	63.3	58	335	4.34	4.35	2.75
...	...	...	...	...	...	...	...	...	...

`stat_count()`

corte	count	prop
Regular	1610	1
Bueno	4906	1
Muy Bueno	12082	1
Premium	13791	1
Ideal	51551	1

A continuación, podrías elegir un objeto geométrico para representar cada observación en los datos transformados. Luego, podrías usar las propiedades estéticas de los geoms para representar variables de los datos. Asignarías los valores de cada variable a los niveles de una estética.

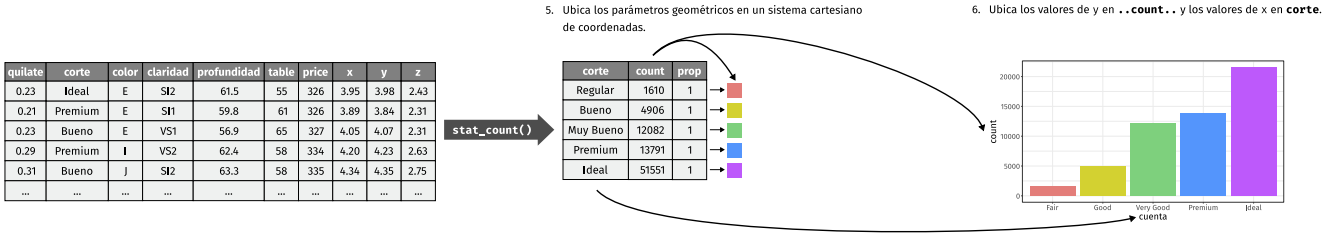
2. Representa cada observación por medio de una columna.
3. Mapea el **`relleno`** de cada columna a la variable **`..count..`**

quilate	corte	color	claridad	profundidad	table	price	x	y	z
0.23	Ideal	E	SI2	61.5	55	326	3.95	3.98	2.43
0.21	Premium	E	SI1	59.8	61	326	3.89	3.84	2.31
0.23	Bueno	E	VS1	56.9	65	327	4.05	4.07	2.31
0.29	Premium	I	VS2	62.4	58	334	4.20	4.23	2.63
0.31	Bueno	J	SI2	63.3	58	335	4.34	4.35	2.75
...	...	...	...	...	...	...	...	...	...

`stat_count()`

corte	count	prop
Regular	1610	1
Bueno	4906	1
Muy Bueno	12082	1
Premium	13791	1
Ideal	51551	1

Posteriormente, podrías seleccionar un sistema de coordenadas para colocar los geoms. Podrías utilizar la ubicación de los objetos (que es en sí misma una propiedad estética) para mostrar los valores de las variables x e y. Ya en este punto podrías tener un gráfico completo, pero también podrías ajustar aún más las posiciones de los geoms dentro del sistema de coordenadas (un ajuste de posición) o dividir el gráfico en facetas. También podrías extender el gráfico agregando una o más capas adicionales, donde cada capa adicional usaría un conjunto de datos, un geom, un conjunto de mapeos, una estadística y un ajuste de posición.



quilate	corte	color	claridad	profundidad	table	price	x	y	z
0.23	Ideal	E	SI2	61.5	55	326	3.95	3.98	2.43
0.21	Premium	E	SI1	59.8	61	326	3.89	3.84	2.31
0.23	Bueno	E	VS1	56.9	65	327	4.05	4.07	2.31
0.29	Premium	I	VS2	62.4	58	334	4.20	4.23	2.63
0.31	Bueno	J	SI2	63.3	58	335	4.34	4.35	2.75
...	...	...	...	...	...	...	...	...	...

corte	count	prop
Regular	1610	1
Bueno	4906	1
Muy Bueno	12082	1
Premium	13791	1
Ideal	51551	1

Puedes usar este método para construir *cualquier* gráfico que imagines. En otras palabras, puedes usar la plantilla de código que aprendiste en este capítulo para construir cientos de miles de gráficos únicos.

CAPÍTULO 4

Flujo de trabajo: conocimientos básicos

Ya tienes un poco de experiencia ejecutando código R. No te hemos dado demasiados detalles, pero es evidente que has podido resolver lo básico ¡o ya habrías arrojado lejos este libro en un acceso de frustración! Es natural frustrarte cuando empiezas a programar en R ya que es muy estricto en cuanto a la puntuación: incluso un único caracter fuera de lugar provocará que se queje. Si bien deberías esperar sentir un poco de frustración, confía en que esta sensación es normal y transitoria: le pasa a todas las personas y la única forma de superarla es seguir intentando.

Antes de avanzar vamos a asegurarnos de que tengas una base sólida ejecutando código R, y que conozcas algunas de las características más útiles de RStudio.

4.1. Conocimientos básicos de programación

Revisemos algunos conocimientos básicos que omitimos hasta ahora para que pudieras empezar a hacer gráficos lo más rápido posible. Puedes usar R como una calculadora:

```
1 / 200 * 30
#> [1] 0.15
(59 + 73 + 2) / 3
#> [1] 44.7
sin(pi / 2)
#> [1] 1
```

(`sin` calcula por defecto la función trigonométrica *seno*)

Puedes crear objetos nuevos usando `<-`:

```
x <- 3 * 4
```

Todas las instrucciones en R en las que crees objetos, es decir, las instrucciones de **asignación**, tienen la misma estructura:

```
nombre_objeto <- valor
```

Cuando leas esa línea de código di mentalmente “nombre_objeto recibe valor”.

Harás una gran cantidad de asignaciones y <- es incómodo de escribir. Que no te gane la pereza de usar =: sí, funcionará, pero provocará confusión más adelante. En cambio, usa el atajo de teclado de RStudio Alt + - (signo menos). RStudio automágicamente rodeará <- con espacios, lo que es un buena costumbre para dar formato al código. El código puede ser horrible para leer incluso en un buen día, por lo que ayudará a tu vista usar espacios.

4.2. La importancia de los nombres

Los nombres de los objetos deben comenzar con una letra y solo pueden contener letras, números, _ y .. Es mejor que los nombres sean descriptivos. Por eso necesitarás una convención para usar más de una palabra. Nosotros recomendamos **guion_bajo** (o *snake_case*) en el que las palabras en minúscula y sin tilde se separan con _.

```
yo_uso_guion_bajo
OtraGenteUsaMayusculas
algunas.personas.usan.puntos
Y_algunasPocas.Personas_RENIEGANdelasconvenciones
```

Volveremos a tratar el estilo del código más adelante, en [funciones].

Puedes examinar un objeto escribiendo su nombre:

```
x
#> [1] 12
```

Hagamos otra asignación:

```
este_es_un_nombre_muy_largo <- 2.5
```

Para examinar este objeto utiliza la capacidad de RStudio para completar: escribe “este”, presiona TAB, agrega caracteres hasta conseguir una única opción y finaliza apretando Enter.

¡Oh, cometiste un error! `este_es_un_nombre_muy_largo` debería valer 3.5 y no 2.5. Usa otro atajo del teclado para corregirlo. Escribe “este”, luego presiona Cmd/Ctrl + ↑. Aparecerá una lista con todas los comandos que has escrito que empiezan con esas letras. Usa las flechas para navegar y presiona Enter para reescribir el comando elegida. Cambia 2.5 por 3.5 y vuelve a ejecutarlo.

Hagamos una asignación más:

```
viva_r <- 2 ^ 3
```

Probemos examinar el objeto

```
viv_r
#> Error: object 'viv_r' not found
viva_R
#> Error: object 'viva_R' not found
```

Los mensajes de error señalan que R no encontró entre los objetos definidos ninguno que se llame `viv_r` ni `viva_R`.

Existe un acuerdo implícito entre tú y R: R hará todos los tediosos cálculos por ti, pero a cambio tú debes dar las instrucciones con total precisión. Importa si hay errores ortotipográficos (*typos*). Importa si algo está en mayúscula o minúscula.

4.3. Usando funciones

R tiene una gran colección de funciones integradas que se usan así:

```
nombre_funcion(arg1 = val1, arg2 = val2, ...)
```

Probemos usar `seq()` que construye **sec**uencias regulares de números y, mientras tanto, aprendamos otras características útiles de RStudio. Escribe `se` y presiona TAB. Una ventana emergente te mostrará opciones para completar tu instrucción. Especifica `seq()` agregando caracteres que permitan desambiguar (agrega una `q`), o usando las flechas ↑/↓. Si necesitas ayuda, presiona F1 para obtener información detallada en la pestaña de ayuda del panel inferior derecho.

Presiona TAB una vez más cuando hayas seleccionado la función que quieras. RStudio colocará por tí paréntesis de apertura (`(`) y cierre (`)`) de a pares. Escribe los argumentos `1, 10` y presiona Enter.

```
seq(1, 10)
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

Escribe este código y observa que RStudio también te asiste al utilizar comillas:

```
x <- "hola mundo"
```

Comillas y paréntesis siempre se usan de a pares. RStudio hace lo mejor que puede para ayudarte; sin embargo puede ocurrir que nos enredemos y terminemos con una disparidad. Si esto ocurre R te mostrará el caracter de continuación "+":

```
> x <- "hola
+
```

El + te indica que R está esperando que completes la instrucción; no cree que hayas terminado. Usualmente esto implica que olvidaste escribir `"` o `)`. Puedes agregar el caracter par faltante o presionar ESCAPE para abandonar la expresión y escribirla de nuevo.

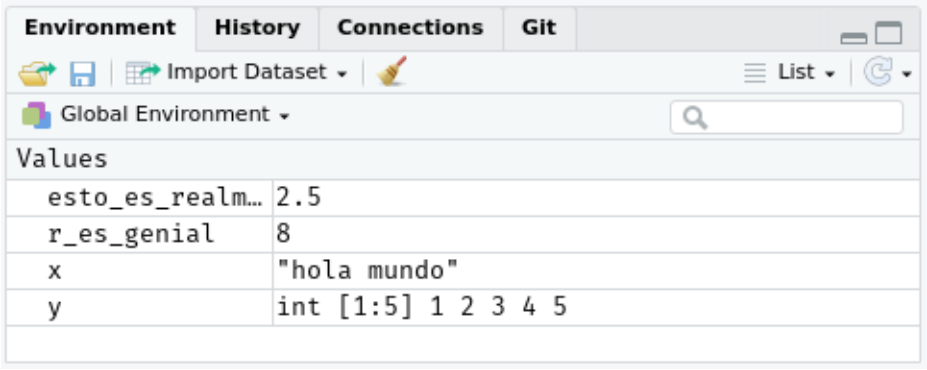
Cuando realizas una asignación no se imprime en la consola el valor asignado. Es una tentación confirmar inmediatamente el resultado:

```
y <- seq(1, 10, length.out = 5)
y
#> [1]  1.00  3.25  5.50  7.75 10.00
```

Esta acción común puede acortarse rodeando la instrucción con paréntesis, lo que resulta en una asignación e "impresión en la pantalla".

```
(y <- seq(1, 10, length.out = 5))
#> [1]  1.00  3.25  5.50  7.75 10.00
```

Mira tu entorno de trabajo en el panel superior derecho:



Allí puedes ver todos los objetos que creaste.

4.4. Ejercicios

1. ¿Por qué no funciona este código?

   ```
   mi_variable <- 10
   mi_varıable
   #> Error in eval(expr, envir, enclos): object 'mi_varıable' not found
   ```

   ¡Mira detenidamente! (Esto puede parecer un ejercicio inútil, pero entrenar tu cerebro para detectar incluso las diferencias más pequeñas será muy útil cuando comiences a programar.)

2. Modifica cada una de las instrucciones de R a continuación para que puedan ejecutarse correctamente:

   ```
   library(tidyverse)

   ggplot(dota = millas) +
     geom_point(mapping = aes(x = cilindrada, y = autopista))

   fliter(millas, cilindros = 8)
   filter(diamante, quilate > 3)
   ```

3. Presiona Alt + Shift + K. ¿Qué ocurrió? ¿Cómo puedes llegar al mismo lugar utilizando los menús?

CAPÍTULO 5

Transformación de datos

5.1. Introducción

La visualización es una herramienta importante para para la generación de conocimiento; sin embargo, es raro que obtengas los datos exactamente en la forma correcta que los necesitas. A menudo tendrás que crear algunas variables nuevas o resúmenes, o tal vez solo quieras cambiar el nombre de las variables o reordenar las observaciones para facilitar el trabajo con los datos. En este capítulo aprenderás cómo hacer todo eso (¡y más!), incluyendo cómo transformar tus datos utilizando el paquete **dplyr** y el uso de un nuevo conjunto de datos sobre salida de vuelos de la ciudad de Nueva York en el año 2013.

5.1.1. Prerequisitos

En este capítulo nos enfocaremos en cómo usar el paquete **dplyr**, otro miembro central del tidyverse. Ilustraremos las ideas clave con el *dataset vuelos* que está contenido en el paquete **datos**. Utilizaremos **ggplot2** para ayudarnos a comprender los datos.

```
#remotes::install_github("cienciadedatos/datos")
library(datos)
library(tidyverse)
```

Toma nota acerca del mensaje de conflictos que se imprime cuando cargas el paquete **tidyverse**. Te indica que **dplyr** sobrescribe algunas funciones de R base. Si deseas usar la versión base de estas funciones después de cargar **dplyr**, necesitarás usar sus nombres completos: `stats::filter()` y `stats::lag()`.

5.1.2. vuelos

Para explorar los verbos básicos de manipulación de datos de **dplyr**, usaremos `vuelos`. Este conjunto de datos contiene los 336, 776 vuelos que partieron de la ciudad de Nueva York durante el 2013. Los datos provienen del Departamento de Estadísticas de Transporte de los Estados Unidos, y están

documentados en `?vuelos`.

```
vuelos
#> # A tibble: 336,776 x 19
#>    anio   mes   dia horario_salida salida_programa~ 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
#> # ... with 336,770 more rows, and 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>
```

Es posible que observes que este conjunto de datos se imprime de una forma un poco diferente a otros que podrías haber utilizado en el pasado: solo muestra las primeras filas y todas las columnas que caben en tu pantalla. Para ver todo el conjunto de datos, puedes ejecutar `View(vuelos)` que abrirá el conjunto de datos en el visor de RStudio. En este caso se imprime de manera diferente porque es un **tibble**. Los *tibbles* son *data frames*, pero ligeramente ajustados para que funcionen mejor en el tidyverse. Por ahora, no necesitas preocuparte por las diferencias; hablaremos en más detalle de los *tibbles* en Manejar o domar datos.

También podrás haber notado la fila de tres (o cuatro) abreviaturas de letras debajo de los nombres de las columnas. Estos 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 otros tres tipos comunes de variables que no se usan en este conjunto de datos, pero que encontrarás 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.

5.1.3. Lo básico de dplyr

En este capítulo, aprenderás las cinco funciones clave de **dplyr** que te permiten resolver la gran mayoría de tus desafíos de manipulación de datos:

- Filtrar o elegir las observaciones por sus valores (`filter()` — del inglés filtrar).
- Reordenar las filas (`arrange()` — del inglés organizar).
- Seleccionar las variables por sus nombres (`select()` — del inglés seleccionar).
- Crear nuevas variables con transformaciones de variables existentes (`mutate()` — del inglés mutar o transformar).
- Contraer muchos valores en un solo resumen (`summarise()` — del inglés resumir).

Todas estas funciones se pueden usar junto con `group_by()` (del inglés *agrupar por*), que cambia el alcance de cada función para que actúe ya no sobre todo el conjunto de datos sino de grupo en grupo. Estas seis funciones proporcionan los verbos para este lenguaje de manipulación de datos.

Todos los verbos funcionan de manera similar:

1. El primer argumento es un *data frame*.

2. Los argumentos posteriores describen qué hacer con el *data frame* usando los nombres de las variables (sin comillas).

3. El resultado es un nuevo *data frame*.

En conjunto, estas propiedades hacen que sea fácil encadenar varios pasos simples para lograr un resultado complejo. Sumerjámosnos y veamos cómo funcionan estos verbos.

5.2. Filtrar filas con `filter()`

`filter()` te permite filtrar un subconjunto de observaciones según sus valores. El primer argumento es el nombre del *data frame*. El segundo y los siguientes argumentos son las expresiones que lo filtran. Por ejemplo, podemos seleccionar todos los vuelos del 1 de enero con:

```
filter(vuelos, mes == 1, dia == 1)
#> # A tibble: 842 x 19
#>    anio   mes   dia horario_salida salida_programa~ 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
#> # ... with 836 more rows, and 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>
```

Cuando ejecutas esa línea de código, **dplyr** ejecuta la operación de filtrado y devuelve un nuevo *data frame*. Las funciones de **dplyr** nunca modifican su *input*, por lo que si deseas guardar el resultado, necesitarás 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_programa~ 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
#> # ... with 713 more rows, and 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>
```

5.2.1. Comparaciones

Para usar el filtrado de manera efectiva, debes saber cómo seleccionar las observaciones que deseas utilizando los operadores de comparación. R proporciona el conjunto estándar: >, >=, <, <=, != (no igual) y == (igual).

Cuando comienzas con R, el error más fácil de cometer es usar = en lugar de == cuando se busca igualdad. Cuando esto suceda, obtendrás un error informativo:

```
filter(vuelos, mes = 1)
#> Error: Problem with filter() input ..1.
#> x Input ..1 is named.
#> i This usually means that you've used = instead of ==.
#> i Did you mean mes == 1?
```

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
1 / 49 * 49 == 1
#> [1] FALSE
```

Las computadoras usan aritmética de precisión finita (obviamente, no pueden almacenar una cantidad infinita de dígitos), así que recuerda que cada número que ves es una aproximación. En lugar de confiar en ==, usa `near()` (cercano, en inglés):

```r
near(sqrt(2)^2, 2)
#> [1] TRUE
near(1 / 49 * 49, 1)
#> [1] TRUE
```

5.2.2. Operadores lógicos

Si tienes múltiples argumentos para `filter()` estos se combinan con "y": cada expresión debe ser verdadera para que una fila se incluya en el *output*. Para otros tipos de combinaciones necesitarás usar operadores Booleanos: `&` es "y", `|` es "o", y `!` es "no". La figura @ref(fig:bool-ops) muestra el conjunto completo de operaciones Booleanas.

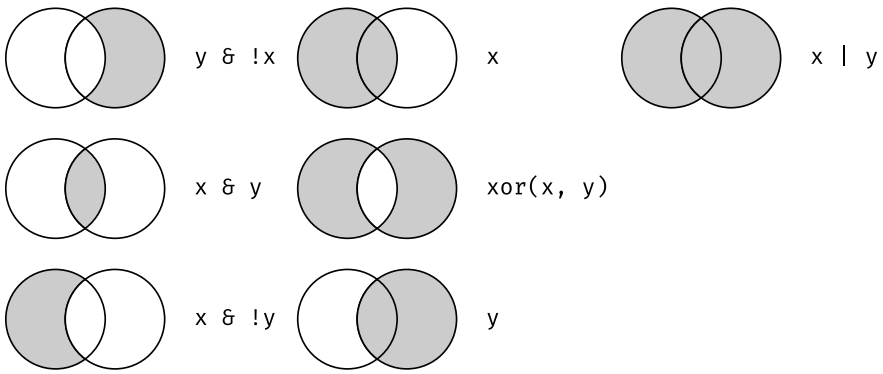


Figura 5.1: Complete set of boolean operations. 'x' is the left-hand circle, 'y' is the right-hand circle, and the shaded region show which parts each operator selects.

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. No puedes escribir `filter(vuelos, mes == (11 | 12))`, que literalmente puede traducirse como "encuentra todos los vuelos que partieron en noviembre o diciembre". En cambio, encontrará todos los meses que son iguales a `11 | 12`, una expresión que resulta en 'TRUE' (verdadero). En un contexto numérico (como aquí), 'TRUE' se convierte en uno, por lo que encuentra todos los vuelos en enero, no en noviembre o diciembre. ¡Esto es bastante confuso!

Una manera rápida y útil para resolver este problema es `x %in% y` (es decir, x *en* y). Esto seleccionará cada fila donde x es uno de los valores en y. Podríamos usarlo para reescribir el código de arriba:

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

A veces puedes simplificar subconjuntos complicados al recordar la ley de De Morgan: `!(x & y)` es lo mismo que `!x | !y`, y `!(x | y)` es lo mismo que `!x & !y`. Por ejemplo, si deseas encontrar vuelos que no se retrasaron (en llegada o partida) en más de dos horas, puedes usar cualquiera de los dos filtros siguientes:

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

Además de & y |, R también tiene && y ||. ¡No los uses aquí! Aprenderás cuándo deberías usarlos en [Ejecución condicional].

Siempre que empieces a usar en `filter()` expresiones complejas que tengan varias partes, considera convertirlas en variables explícitas. Eso hace que sea mucho más fácil verificar tu trabajo. Aprenderás cómo crear nuevas variables en breve.

5.2.3. Valores faltantes

Una característica importante de R que puede hacer que la comparación sea difícil son los valores faltantes, o NAs (del inglés "no disponibles"). NA representa un valor desconocido, lo que hace que los valores perdidos sean "contagiosos": casi cualquier operación que involucre un valor desconocido también será desconocida.

```
NA > 5
#> [1] NA
10 == NA
#> [1] NA
NA + 10
#> [1] NA
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()`:

```
is.na(x)
#> [1] TRUE
```

`filter()` solo incluye filas donde la condición es TRUE; excluye tanto los valores FALSE como NA. Si deseas conservar valores perdidos, solicítalos explícitamente:

```
df <- tibble(x = c(1, NA, 3))
filter(df, x > 1)
#> # A tibble: 1 x 1
#>       x
#>   <dbl>
#> 1     3
filter(df, is.na(x) | x > 1)
#> # A tibble: 2 x 1
#>       x
#>   <dbl>
#> 1    NA
#> 2     3
```

5.2.4. Ejercicios

1. Encuentra todos los vuelos que:
2. Tuvieron un retraso de llegada de dos o más horas
3. Volaron a Houston (`IAH` oHOU)
4. Fueron operados por United, American o Delta
5. Partieron en invierno (julio, agosto y septiembre)
6. Llegaron más de dos horas tarde, pero no salieron tarde
7. Se retrasaron por lo menos una hora, pero repusieron más de 30 minutos en vuelo
8. Partieron entre la medianoche y las 6 a.m. (incluyente)
9. 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?
10. ¿Cuántos vuelos tienen datos faltantes en `horario_salida`? ¿Qué otras variables tienen valores

faltantes? ¿Qué representan estas filas?

11. ¿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!)

5.3. Reordenar las filas con `arrange()`

`arrange()` funciona de manera similar a `filter()` excepto que en lugar de seleccionar filas, cambia su orden. La función toma un *data frame* y un conjunto de nombres de columnas (o expresiones más complicadas) para ordenar según ellas. Si proporcionas más de un nombre de columna, cada columna adicional se utilizará para romper empates en los valores de las columnas anteriores:

```
arrange(vuelos, anio, mes, dia)
#> # A tibble: 336,776 x 19
#>    anio   mes   dia horario_salida salida_programa~ 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
#> # ... with 336,770 more rows, and 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_programa~ 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
#> # ... with 336,770 more rows, and 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
```

5.3.1. 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?

5.4. Seleccionar columnas con `select()`

No es raro obtener conjuntos de datos con cientos o incluso miles de variables. En este caso, el primer desafío a menudo se reduce a las variables que realmente te interesan. `select()` te permite seleccionar rápidamente un subconjunto útil utilizando operaciones basadas en los nombres de las variables.

`select()` no es muy útil con los datos de los vuelos porque solo tenemos 19 variables, pero de todos modos se entiende la idea general:

```
# 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
#> # ... with 336,770 more rows
# 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
#> # ... with 336,770 more rows
# Seleccionar todas las columnas excepto aquellas entre anio en dia (incluyente)
select(vuelos, -(anio:dia))
#> # A tibble: 336,776 x 16
#>   horario_salida salida_programa~ atraso_salida horario_llegada llegada_program~
#>            <int>            <int>         <dbl>           <int>            <int>
#> 1            517              515             2             830              819
#> 2            533              529             4             850              830
#> 3            542              540             2             923              850
#> 4            544              545            -1            1004             1022
#> 5            554              600            -6             812              837
#> 6            554              558            -4             740              728
#> # ... with 336,770 more rows, and 11 more variables: 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 una serie de funciones auxiliares que puedes usar dentro de `select()`:

- `starts_with(".ªbc")`: coincide con los nombres que comienzan con “abc”.
- `ends_with("xyz")`: coincide con los nombres que terminan con “xyz”.
- `contains(ïjk")`: coincide con los nombres que contienen “ijk”.
- `matches("(.)\\1")`: selecciona variables que coinciden con una expresión regular. Esta en particular coincide con cualquier variable que contenga caracteres repetidos. Aprenderás más sobre expresiones regulares en [Cadenas de caracteres].
- `num_range("x", 1:3)`: coincide con `x1`,`x2` y `x3`.

Consulta `?select` para ver más detalles.

`select()` se puede usar para cambiar el nombre de las variables, pero rara vez es útil porque descarta todas las variables que no se mencionan explícitamente. En su lugar, utiliza `rename()`, que es una variante de `select()` que mantiene todas las variables que no se mencionan explícitamente:

```
rename(vuelos, cola_num = codigo_cola)
#> # A tibble: 336,776 x 19
#>    anio   mes   dia horario_salida salida_programa~ 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
#> # ... with 336,770 more rows, and 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()` junto con el auxiliar `everything()` (*todo*, en inglés). Esto es útil si tienes un grupo de variables que te gustaría mover 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
#> # ... with 336,770 more rows, and 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>
```

5.4.1. Ejercicios

1. Haz una lluvia de ideas sobre tantas maneras como sea posible para seleccionar `horario_salida`,`atraso_salida`,`horario_llegada`, y`atraso_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 `one_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"))
```

5.5. Añadir nuevas variables con `mutate()`

Además de seleccionar conjuntos de columnas existentes, a menudo es útil crear nuevas columnas en función de columnas existentes. Ese es el trabajo de `mutate()` (del inglés *mutar* o *transformar*).

`mutate()` siempre agrega nuevas columnas al final de un conjunto de datos, así que comenzaremos creando un conjunto de datos más pequeño para que podamos ver las nuevas variables. Recuerda que cuando usas RStudio, la manera más fácil de ver todas las columnas es `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 ganancia
#>   <int> <int> <int>         <dbl>          <dbl>     <dbl>        <dbl>    <dbl>
#> 1  2013     1     1             2             11      1400          227       -9
#> 2  2013     1     1             4             20      1416          227      -16
#> 3  2013     1     1             2             33      1089          160      -31
#> 4  2013     1     1            -1            -18      1576          183       17
#> 5  2013     1     1            -6            -25       762          116       19
#> 6  2013     1     1            -4             12       719          150      -16
#> # ... with 336,770 more rows, and 1 more variable: 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 ganancia
#>   <int> <int> <int>         <dbl>          <dbl>     <dbl>        <dbl>    <dbl>
#> 1  2013     1     1             2             11      1400          227       -9
#> 2  2013     1     1             4             20      1416          227      -16
#> 3  2013     1     1             2             33      1089          160      -31
#> 4  2013     1     1            -1            -18      1576          183       17
#> 5  2013     1     1            -6            -25       762          116       19
#> 6  2013     1     1            -4             12       719          150      -16
#> # ... with 336,770 more rows, and 2 more variables: 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
#> # ... with 336,770 more rows
```

5.5.1. Funciones de creación útiles

Hay muchas funciones para crear nuevas variables que puedes usar con `mutate()`. La propiedad clave es que la función debe ser vectorizada: debe tomar un vector de valores como *input*, y devolver un vector con el mismo número de valores como *output*. No hay forma de enumerar todas las posibles funciones que podrías usar, pero aquí hay una selección de funciones que frecuentemente son útiles:

- Operadores aritméticos: `+`, `-`,`*`,`/`,`^`. Todos están vectorizados usando las llamadas "reglas de reciclaje". Si un parámetro es más corto que el otro, se extenderá automáticamente para tener la misma longitud. Esto es muy útil cuando uno de los argumentos es un solo número: `tiempo_vuelo / 60`, `horas * 60 + minuto`, etc.

  Los operadores aritméticos también son útiles junto con las funciones de agregar que aprenderás más adelante. Por ejemplo, `x / sum(x)` calcula la proporción de un total, y `y - mean(y)` calcula la diferencia de la media.

- Aritmética modular: `%/%` (división entera) y `%%` (resto), donde `x == y * (x %/% y) + (x %% y)`. La aritmética modular es una herramienta útil porque te permite dividir enteros en par-

tes. Por ejemplo, en el conjunto de datos de vuelos, puedes calcular `hora` y `minutos` de `horario_salida` con:

```
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
#> # ... with 336,770 more rows
```

- Logaritmos: `log()`, `log2()`, `log10()`. Los logaritmos son increíblemente útiles como transformación para trabajar con datos con múltiples órdenes de magnitud. También convierten las relaciones multiplicativas en aditivas, una característica que retomaremos en los capítulos sobre modelos.

  En igualdad de condiciones, recomendamos usar `log2()` porque es más fácil de interpretar: una diferencia de 1 en la escala de registro corresponde a la duplicación de la escala original y una diferencia de -1 corresponde a dividir a la mitad.

- Rezagos: `lead()` y `lag()` te permiten referirte a un valor adelante o un valor atrás (con rezago). Esto te permite calcular as diferencias móviles (por ejemplo, `x - lag(x)`) o encontrar cuándo cambian los valores (`x != lag(x)`). Estos comandos son más útiles cuando se utilizan junto con `group_by()`, algo que aprenderás en breve.

```
(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 proporciona funciones para ejecutar sumas, productos, mínimos y máximos: `cumsum()`, `cumprod()`, `cummin()`, `cummax()`; **dplyr**, por su parte, proporciona `cummean()` para las medias acumuladas. Si necesitas calcular agregados móviles (es decir, una suma calculada en una ventana móvil), prueba 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] 1.00 1.00 1.33 1.75 2.20 2.67 3.14 3.62 4.11 4.60
```

- Comparaciones lógicas: <, <=, >, >=, != sobre las cuales aprendiste antes. Si estás haciendo una secuencia compleja de operaciones lógicas, es a menudo una buena idea almacenar los valores provisionales en nuevas variables para que puedas comprobar que cada paso funciona como se espera.

- Ordenamiento: hay una serie de funciones de ordenamiento (ranking), pero deberías comenzar con `min_rank()`. Esta función realiza el tipo más común de ordenamiento (por ejemplo, primero, segundo, tercero, etc.). El valor predeterminado otorga la menor posición a los valores más pequeños; usa `desc(x)` para dar la menor posición a los valores más grandes.

```
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 hace lo que necesitas, consulta las variantes `row_number()`, `dense_rank()`, `percent_rank()`, `cume_dist()`,`quantile()`. Revisa 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
cume_dist(y)
#> [1] 0.2 0.6 0.6  NA 0.8 1.0
```

5.5.2. 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?

5.6. Resúmenes agrupados con `summarise()`

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

```
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 a menos que lo enlacemos con `group_by()`. Esto cambia la unidad de análisis del conjunto de datos completo a grupos individuales. Luego, cuando uses los verbos **dplyr** en un *data frame* agrupado, estos se aplicarán automáticamente "por grupo". Por ejemplo, si aplicamos exactamente el mismo código a un *data frame* agrupado por fecha, obtenemos el retraso promedio por fecha:

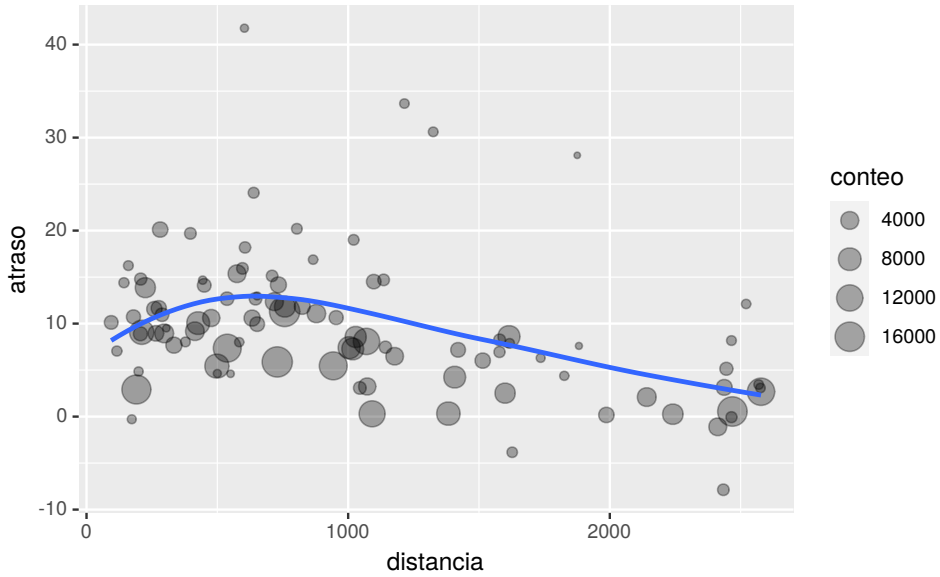
```
por_dia <- group_by(vuelos, anio, mes, dia)
summarise(por_dia, atraso = mean(atraso_salida, na.rm = TRUE))
#> # A tibble: 365 x 4
#> # Groups:   anio, mes [12]
#>    anio   mes   dia atraso
#>   <int> <int> <int>  <dbl>
#> 1  2013     1     1  11.5
#> 2  2013     1     2  13.9
#> 3  2013     1     3  11.0
#> 4  2013     1     4   8.95
#> 5  2013     1     5   5.73
#> 6  2013     1     6   7.15
#> # ... with 359 more rows
```

Juntos `group_by()` y `summarise()` proporcionan una de las herramientas que más comúnmente usarás cuando trabajes con **dplyr**: resúmenes agrupados. Pero antes de ir más allá con esto, tenemos que introducir una idea nueva y poderosa: el *pipe* (pronunciado /paip/, que en inglés significa ducto o tubería).

5.6.1. 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
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")

# 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 un poco frustrante escribir este código porque tenemos que dar un nombre a cada *data frame* intermedio, incluso si el *data frame* en sí mismo no nos importa. Nombrar cosas es difícil y enlentece nuestro análisis.

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

```
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 está transformando, lo que hace que sea más fácil de leer. Puedes leerlo como una serie de declaraciones imperativas: agrupa, luego resume y luego filtra. Como sugiere esta lectura, una buena forma de pronunciar `%>%` cuando se lee el código es "luego".

Lo que ocurre detrás del código, es que `x %>% f(y)` se convierte en `f(x, y)`, y `x %>% f(y) %>% g(z)` se convierte en `g(f(x, y), z)` y así sucesivamente. Puedes usar el *pipe* para reescribir múltiples operaciones de forma que puedas leer de izquierda a derecha, de arriba hacia abajo. Usaremos *pipes* con frecuencia a partir de ahora porque mejora considerablemente la legibilidad del código. Volveremos a este tema con más detalles en [pipes].

Trabajar con el *pipe* es uno de los criterios clave para pertenecer al tidyverse. La única excepción es **ggplot2**: se escribió antes de que se descubriera el *pipe*. Lamentablemente, la siguiente iteración de **ggplot2**, ggvis, que sí utiliza el *pipe*, aún no está lista para el horario estelar.

5.6.2. Valores faltantes

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

```
vuelos %>%
  group_by(anio, mes, dia) %>%
  summarise(mean = mean(atraso_salida))
#> # 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
#> # ... with 359 more rows
```

¡Obtenemos muchos valores faltantes! Esto se debe a que las funciones de agregación obedecen la regla habitual de valores faltantes: si hay uno en el *input*, el *output* también será un valor faltante. Afortunadamente, todas las funciones de agregación tienen un argumento `na.rm` que elimina los va-

lores faltantes antes del cálculo:

```
vuelos %>%
  group_by(anio, mes, dia) %>%
  summarise(mean = mean(atraso_salida, na.rm = TRUE))
#> # 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
#> # ... with 359 more rows
```

En este caso, en el que los valores faltantes representan vuelos cancelados, también podríamos abordar el problema eliminando primero este tipo de vuelos. Guardaremos este conjunto de datos para poder reutilizarlo en los siguientes ejemplos.

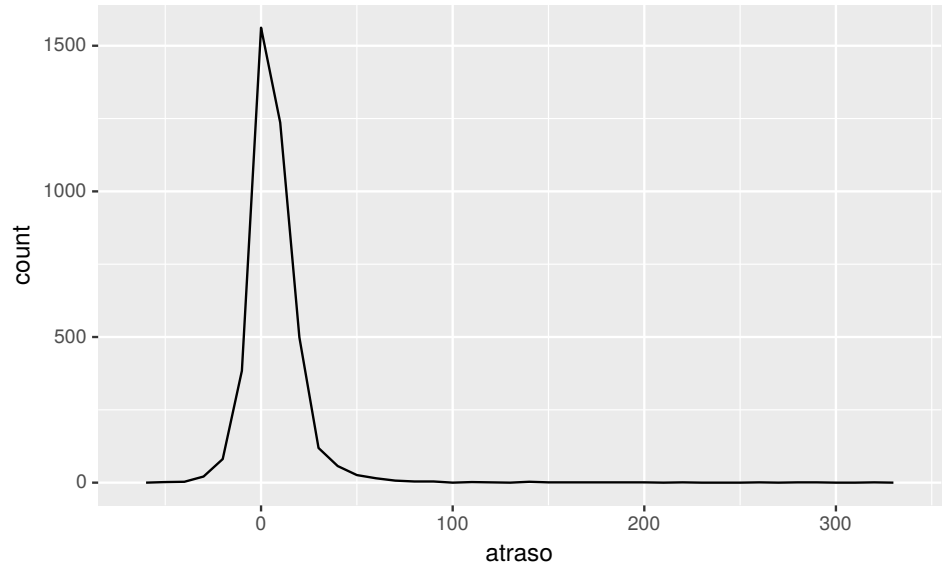
```
no_cancelados <- vuelos %>%
  filter(!is.na(atraso_salida), !is.na(atraso_llegada))

no_cancelados %>%
  group_by(anio, mes, dia) %>%
  summarise(mean = mean(atraso_salida))
#> # 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
#> # ... with 359 more rows
```

5.6.3. Conteos

Siempre que realices una agregación, es una buena idea incluir un conteo (`n()`) o un recuento de valores no faltantes (`sum(!is.na(x))`). De esta forma, puedes verificar que no estás sacando conclusiones basadas en cantidades muy pequeñas de datos. Por ejemplo, veamos los aviones (identificados por su número de cola) que tienen las demoras promedio más altas:

```
atrasos <- no_cancelados %>%
  group_by(codigo_cola) %>%
```

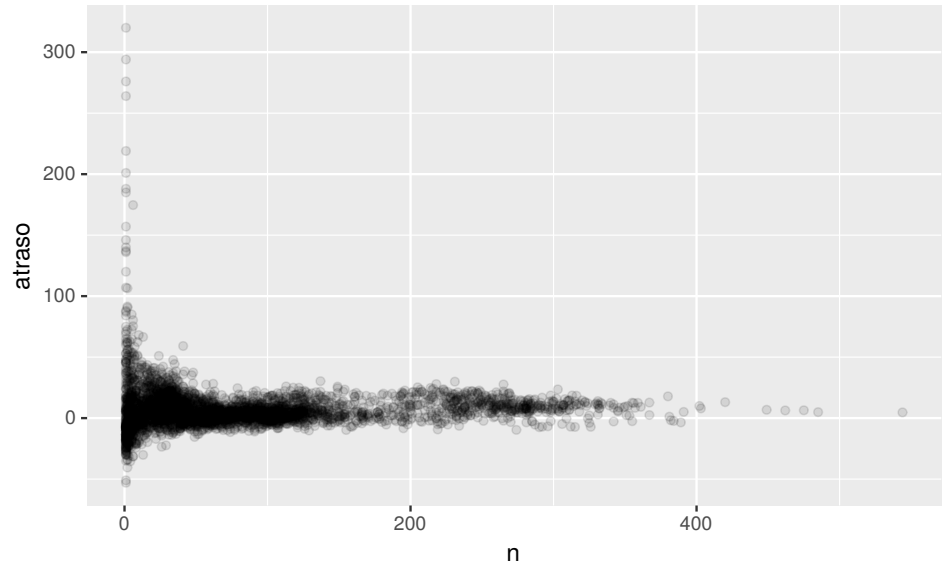
```
  summarise(
   atraso = mean(atraso_llegada)
  )

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:

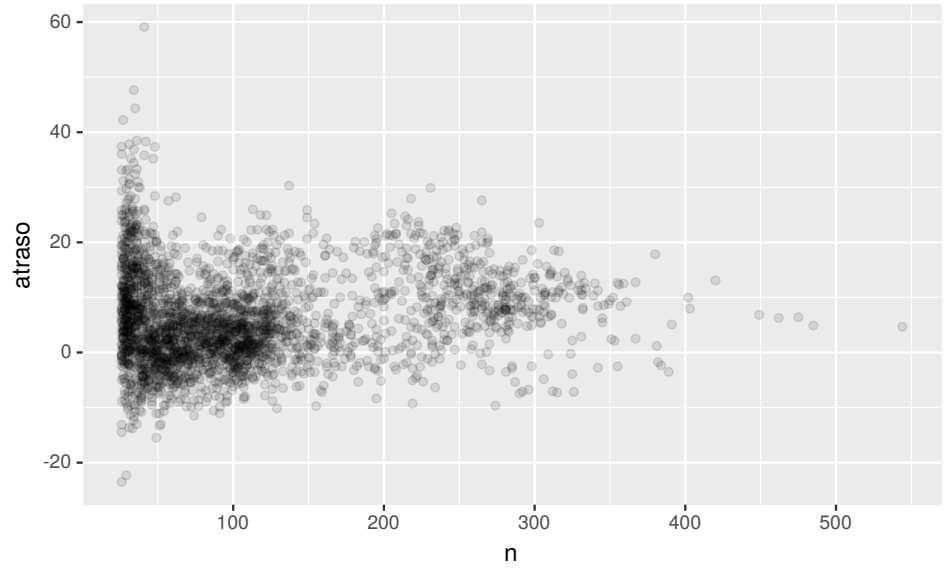
```
atrasos <- no_cancelados %>%
  group_by(codigo_cola) %>%
  summarise(
    atraso = mean(atraso_llegada, na.rm = TRUE),
    n = n()
  )

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



No es sorprendente que haya una mayor variación en el promedio de retraso cuando hay pocos vuelos. La forma de este gráfico es muy característica: cuando trazas un promedio (o cualquier otra medida de resumen) contra el tamaño del grupo, verás que la variación decrece a medida que el tamaño de muestra aumenta.

Cuando se observa este tipo de gráficos, resulta útil eliminar los grupos con menor número de observaciones, ya que puedes ver más del patrón y menos de la variación extrema de los grupos pequeños. Esto es lo que hace el siguiente bloque de código. También te ofrece una manera muy útil para integrar **ggplot2** en el flujo de trabajo de **dplyr**. Es un poco incómodo tener que cambiar de %>% a +, pero una vez que entiendas el código, verás que es bastante conveniente.

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



RStudio tip: un atajo en tu teclado que puede ser muy útil es Cmd/Ctrl + Shift + P. Este reenvía el fragmento enviado previamente del editor a la consola. Esto es muy útil cuando por ejemplo estás explorando el valor de n en el ejemplo anterior. Envías todo el bloque a la consola una vez con Cmd / Ctrl + Enter, y luego modificas el valor de n y presionas Cmd / Ctrl + Shift + P para reenviar el bloque completo.

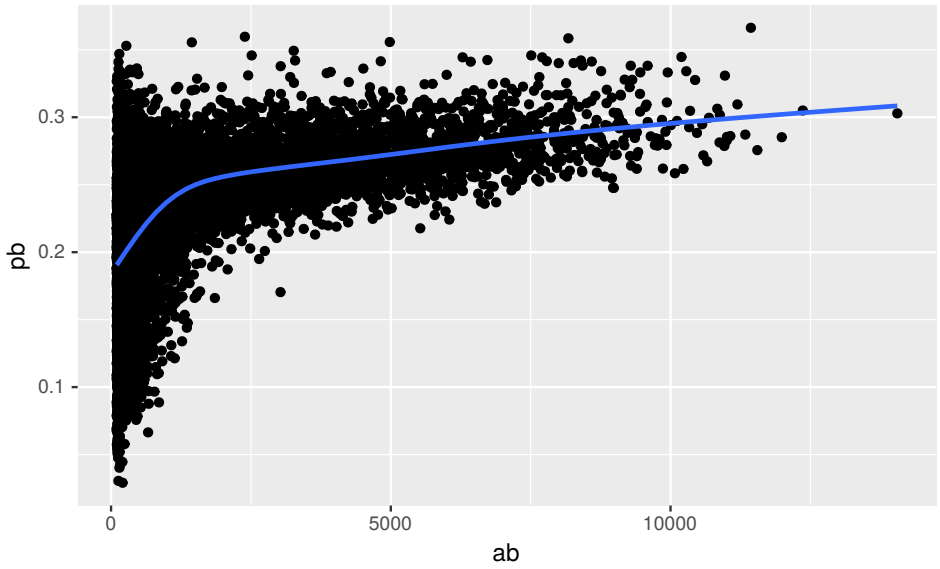
Hay otra variación común de este tipo de patrón. Veamos cómo el rendimiento promedio de los bateadores en el béisbol está relacionado con el número de veces que les toca batear Aquí utilizaremos el conjunto de datos de `bateadores` para calcular el promedio de bateo (número de bateos / número de intentos) de cada jugador de béisbol de Grandes Ligas.

Cuando graficamos la habilidad del bateador (medido por el promedio de bateo, pb) contra el número de oportunidades para golpear la pelota (medido por al bate,ab), verás dos patrones:

1. Como en el ejemplo anterior, la variación en nuestro estadístico de resumen disminuye a medida que obtenemos más observaciones.

2. Existe una correlación positiva entre la habilidad (pb) y las oportunidades para golpear la pelota (ab). Esto se debe a que los equipos controlan quién puede jugar, y obviamente elegirán 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 también tiene implicaciones importantes para la clasificación. Si ingenuamente ordenas `desc(pb)`, verás que las personas con los mejores promedios de bateo tienen claramente mucha suerte, pero no son necesariamente hábiles:

```
rendimiento_bateadores %>%
  arrange(desc(pb))
#> # A tibble: 19,689 x 3
#>   id_jugador    pb    ab
#>   <fct>      <dbl> <int>
#> 1 abramge01      1     1
#> 2 alanirj01      1     1
#> 3 alberan01      1     1
#> 4 banisje01      1     1
#> 5 bartocl01      1     1
#> 6 bassdo01       1     1
#> # ... with 19,683 more rows
```

Puedes encontrar una buena explicación de este problema en http://varianceexplained.org/r/empirical_bayes_baseball/ y http://www.evanmiller.org/how-not-to-sort-by-average-rating.html.

5.6.4. Funciones de resumen útiles

Solo el uso de medias, conteos y sumas puede llevarte muy lejos, pero R proporciona muchas otras funciones de resumen útiles:

- Medidas de centralidad: hemos usado `mean(x)`, pero `median(x)` también resulta muy útil. La media es la suma dividida por el número de observaciones; la mediana es un valor donde el 50 % de x está por encima de él y el 50 % está por debajo. A veces es útil combinar agregación con un subconjunto lógico. Todavía no hemos hablado sobre este tipo de subconjuntos, pero aprenderás más al respecto en [subsetting].

```
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
  )
#> # 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
#> # ... with 359 more rows
```

- Medidas de dispersión: `sd(x)`, `IQR(x)`, `mad(x)`. La raíz de la desviación media al cuadrado o desviación estándar `sd(x)` es una medida estándar de dispersión. El rango intercuartil `IQR()` y la desviación media absoluta `mad(x)` son medidas robustas equivalentes que pueden ser más útiles si tienes valores atípicos.

```
# ¿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
#> # ... with 98 more rows
```

- Medidas de rango: `min(x)`, `quantile(x, 0.25)`, `max(x)`. Los cuantiles son una generalización de la mediana. Por ejemplo, `quantile(x, 0.25)` encontrará un valor de x que sea mayor a 25 % de los valores, y menor que el 75 % restante.

```
# ¿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)
  )
#> # 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
#> # ... with 359 more rows
```

- Medidas de posición: `first(x)`, `nth(x, 2)`, `last(x)`. Estas trabajan de forma similar a `x[1]`, `x[2]` y `x[length (x)]`, pero te permiten establecer un valor predeterminado en el caso de que esa posición no exista (es decir, si estás tratando de obtener el tercer elemento de un grupo que solo tiene dos elementos). Por ejemplo, podemos encontrar la primera (*first*) y última (*last*) salida para cada día:

```
no_cancelados %>%
  group_by(anio, mes, dia) %>%
  summarise(
    primera_salida = first(horario_salida),
    ultima_salida = last(horario_salida)
  )
#> # 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
#> # ... with 359 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_programa~ 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
#> # ... with 764 more rows, and 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: has visto `n()`, que no toma argumentos y que devuelve el tamaño del grupo actual. Para contar la cantidad de valores no faltantes, usa `sum(!is.na(x))`. Para contar la cantidad de valores distintos (únicos), usa `n_distinct(x)`.

```
# ¿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
#> # ... with 98 more rows
```

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

```
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
#> # ... with 98 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
#> # ... with 4,031 more rows
```

- Conteos y proporciones de valores lógicos: `sum(x > 10)`, `mean(y == 0)`. Cuando se usan con funciones numéricas, TRUE se convierte en 1 y FALSE en 0. Esto hace que `sum()` y `mean()` sean muy útiles: `sum(x)` te da la cantidad de TRUE en `x`, y `mean(x)` te da la proporción.

```
# ¿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))
#> # 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
#> # ... with 359 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))
#> # 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
#> # ... with 359 more rows
```

5.6.5. 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()))
#> # 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
#> # ... with 359 more rows
(por_mes <- summarise(por_dia, vuelos = sum(vuelos)))
#> # 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
#> # ... with 6 more rows
(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: está bien para las sumas y los recuentos, pero debes pensar en la ponderación de las medias y las varianzas, además de que no es posible hacerlo exactamente para estadísticas basadas en rangos como la mediana. En otras palabras, la suma de las sumas agrupadas es la suma total, pero la mediana de las medianas agrupadas no es la mediana general.

5.6.6. Desagrupar

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

```
diario %>%
  ungroup() %>%             # ya no está agrupado por fecha
  summarise(vuelos = n())   # todos los vuelos
#> # A tibble: 1 x 1
#>   vuelos
#>    <int>
#> 1 336776
```

5.6.7. 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())`)

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

5.7. Transformaciones agrupadas (y filtros)

La agrupación es más útil si se utiliza junto con `summarise()`, pero también puedes hacer operaciones convenientes 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
#> # ... with 3,300 more rows
```

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

```
popular_destinos <- vuelos %>%
  group_by(destino) %>%
  filter(n() > 365)
popular_destinos
#> # A tibble: 332,577 x 19
#> # Groups:   destino [77]
#>    anio   mes   dia horario_salida salida_programa~ 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
#> # ... with 332,571 more rows, and 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:

```
popular_destinos %>%
  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
#> # ... with 131,100 more rows
```

Un filtro agrupado es una transformación agrupada seguida de un filtro desagrupado. En general, preferimos evitarlos, excepto para las manipulaciones rápidas y sucias: de lo contrario, es difícil comprobar que has hecho la manipulación correctamente.

Las funciones que trabajan de forma más natural en transformaciones agrupadas y filtros se conocen como funciones de ventana o *window functions* (frente a las funciones de resumen utilizadas para los resúmenes). Puedes obtener más información sobre las funciones de ventana útiles en la viñeta correspondiente: `vignette("window-functions")`.

5.7.1. 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.