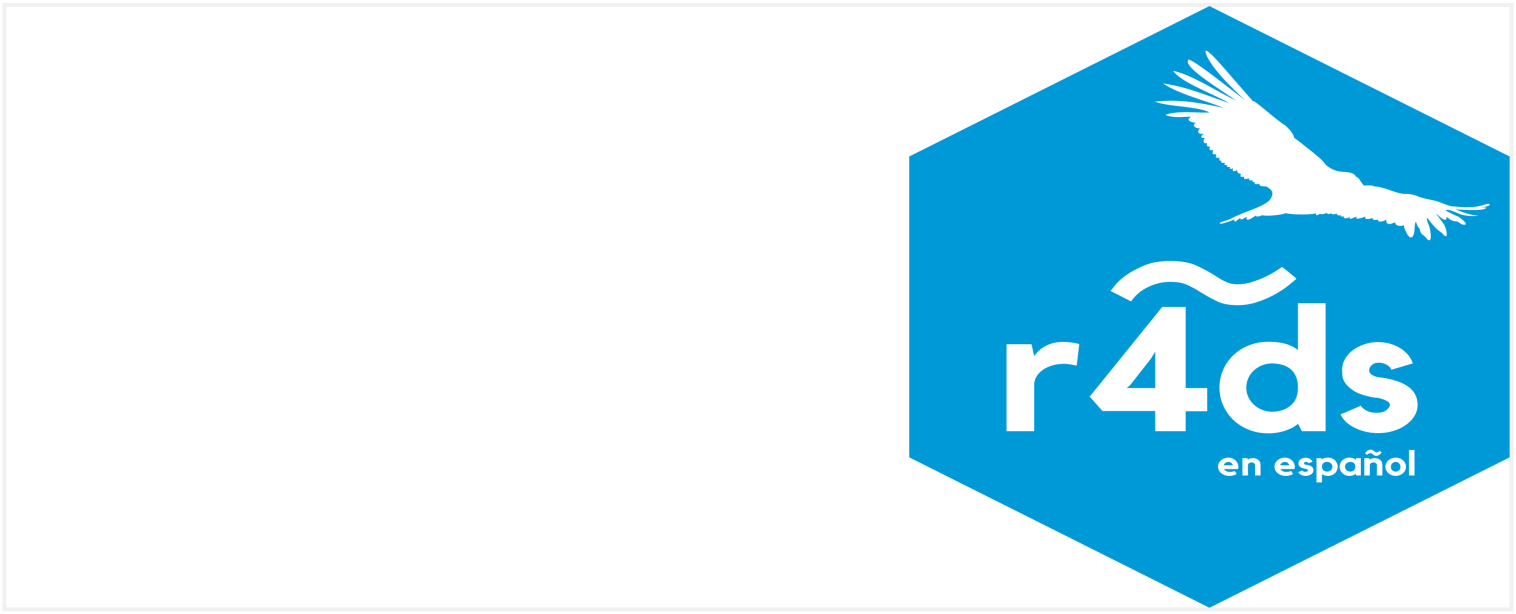
View source

Edit this page

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.

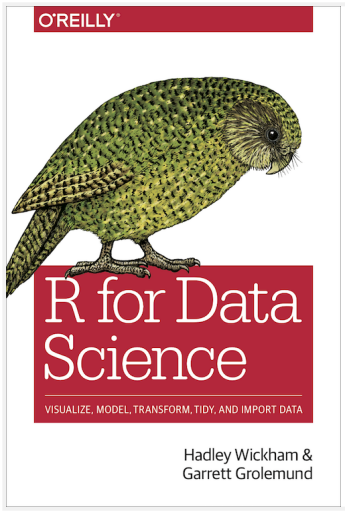
Agradecemos a todas las personas que han ayudado revisando las traducciones y haciendo sugerencias de mejora. Puedes revisar la documentación del proyecto para ver los créditos de participación. Gracias también a Marcela Alfaro por el tuit que hizo visible la necesidad de la versión en español, y a Laura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

1 Introducción »

"" was written by .

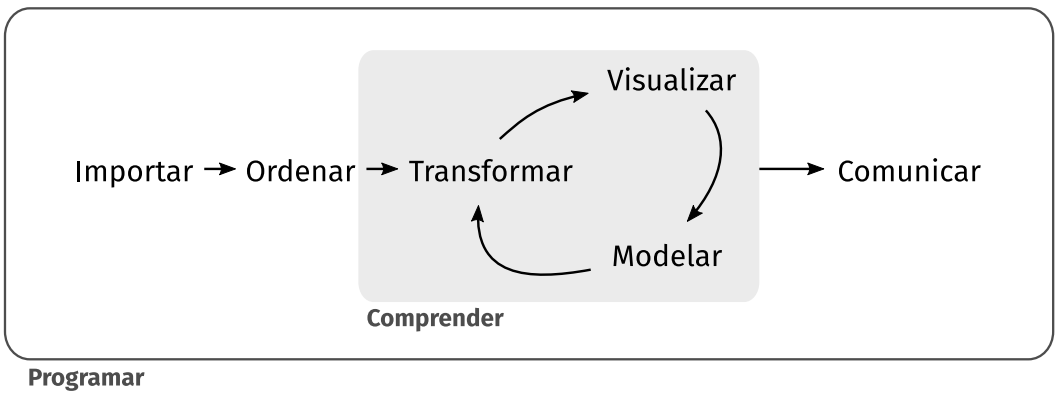
This book was built by the bookdown R package.

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 guardarlos 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,

On this page

View source
Edit this page

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

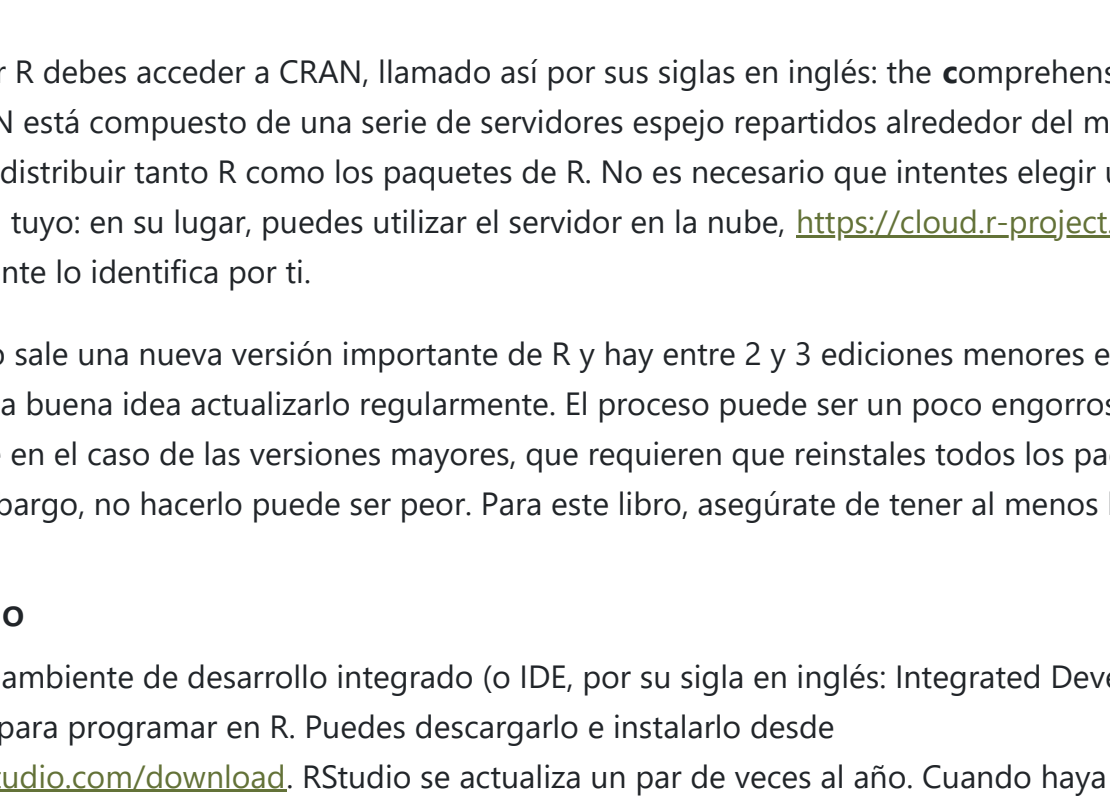
Para descargar R debes acceder a CRAN, llamado así por sus siglas en inglés: the **c**omprehensive **R a**rchive **n**etwork. CRAN está compuesto de una serie de servidores espejo repartidos alrededor del mundo y es utilizado para distribuir tanto R como los paquetes de R. No es necesario que intentes 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")
```

Copy

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)
#> ── Attaching packages ─────────────────────────────────────── tidyverse 1.3.0 ──
#> ✓ ggplot2 3.3.3     ✓ purrr   0.3.4
#> ✓ tibble  3.0.5     ✓ dplyr   1.0.3
#> ✓ tidyr   1.1.2     ✓ stringr 1.4.0
#> ✓ readr   1.4.0     ✓ forcats 0.5.0
#> ── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
#> ✖ dplyr::filter() masks stats::filter()
#> ✖ dplyr::lag()    masks stats::lag()
```

Copy

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 CRAN y puedes instalarlo ejecutando el siguiente código:

```
install.packages("datos")
```

Copy

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
```

Copy

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

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

Copy

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 Morris, 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 4.0.3 (2020-10-10)
#>  os       macOS Catalina 10.15.7
#>  system   x86_64, darwin17.0
#>  ui       X11
#>  language (EN)
#>  collate  en_US.UTF-8
#>  ctype    en_US.UTF-8
#>  tz       UTC
#>  date     2021-01-18
#>
#> ─ Packages ─────────────────────────────────────────────────────────────────
#>  ! package      * version    date       lib source
#>    askpass        1.1        2019-01-13 [1] standard (@1.1)
#>    assertthat     0.2.1      2019-03-21 [1] standard (@0.2.1)
#>    backports      1.2.1      2020-12-09 [1] standard (@1.2.1)
#>    base64enc      0.1-3      2015-07-28 [1] standard (@0.1-3)
#>  R BH             <NA>       <NA>       [?] <NA>
#>    blob           1.2.1      2020-01-20 [1] standard (@1.2.1)
#>    brio           1.1.0      2020-08-31 [1] standard (@1.1.0)
#>    broom          0.7.3      2020-12-16 [1] standard (@0.7.3)
#>    callr          3.5.1      2020-10-13 [1] standard (@3.5.1)
#>    cellranger     1.1.0      2016-07-27 [1] standard (@1.1.0)
#>    cli            2.2.0      2020-11-20 [1] standard (@2.2.0)
#>    clipr          0.7.1      2020-10-08 [1] standard (@0.7.1)
#>    colorspace     2.0-0      2020-11-11 [1] standard (@2.0-0)
#>  R cpp11          <NA>       <NA>       [?] <NA>
#>    crayon         1.3.4      2017-09-16 [1] standard (@1.3.4)
#>    curl           4.3        2019-12-02 [1] standard (@4.3)
#>    DBI            1.1.1      2021-01-15 [1] standard (@1.1.1)
#>    dbplyr         2.0.0      2020-11-03 [1] standard (@2.0.0)
#>    desc           1.2.0      2018-05-01 [1] standard (@1.2.0)
#>    diffobj        0.3.3      2021-01-07 [1] standard (@0.3.3)
#>    digest         0.6.27     2020-10-24 [1] standard (@0.6.27)
#>    dplyr        * 1.0.3      2021-01-15 [1] standard (@1.0.3)
#>    ellipsis       0.3.1      2020-05-15 [1] standard (@0.3.1)
#>    evaluate       0.14       2019-05-28 [1] standard (@0.14)
#>    fansi          0.4.2      2021-01-15 [1] standard (@0.4.2)
#>    farver         2.0.3      2020-01-16 [1] standard (@2.0.3)
#>    forcats      * 0.5.0      2020-03-01 [1] standard (@0.5.0)
#>    fs             1.5.0      2020-07-31 [1] standard (@1.5.0)
#>    generics       0.1.0      2020-10-31 [1] standard (@0.1.0)
#>    ggplot2      * 3.3.3      2020-12-30 [1] standard (@3.3.3)
#>    glue           1.4.2      2020-08-27 [1] standard (@1.4.2)
#>    gtable         0.3.0      2019-03-25 [1] standard (@0.3.0)
#>    haven          2.3.1      2020-06-01 [1] standard (@2.3.1)
#>    highr          0.8        2019-03-20 [1] standard (@0.8)
#>    hms            1.0.0      2021-01-13 [1] standard (@1.0.0)
#>    htmltools      0.5.1.9000 2021-01-18 [1] Github (rstudio/htmltools@11cfbf3)
#>    httr           1.4.2      2020-07-20 [1] standard (@1.4.2)
#>    isoband        0.2.3      2020-12-01 [1] standard (@0.2.3)
#>    jsonlite       1.7.2      2020-12-09 [1] standard (@1.7.2)
#>    knitr          1.30       2020-09-22 [1] standard (@1.30)
#>    labeling       0.4.2      2020-10-20 [1] standard (@0.4.2)
#>    lattice        0.20-41    2020-04-02 [2] CRAN (R 4.0.3)
#>    lifecycle      0.2.0      2020-03-06 [1] standard (@0.2.0)
#>    lubridate      1.7.9.2    2020-11-13 [1] standard (@1.7.9.2)
#>    magrittr       2.0.1      2020-11-17 [1] standard (@2.0.1)
#>    markdown       1.1        2019-08-07 [1] standard (@1.1)
#>    MASS           7.3-53     2020-09-09 [2] CRAN (R 4.0.3)
#>    Matrix         1.3-2      2021-01-06 [1] standard (@1.3-2)
#>    mgcv           1.8-33     2020-08-27 [2] CRAN (R 4.0.3)
#>    mime           0.9        2020-02-04 [1] standard (@0.9)
#>    modelr         0.1.8      2020-05-19 [1] standard (@0.1.8)
#>    munsell        0.5.0      2018-06-12 [1] standard (@0.5.0)
#>    nlme           3.1-151    2020-12-10 [1] standard (@3.1-151)
```

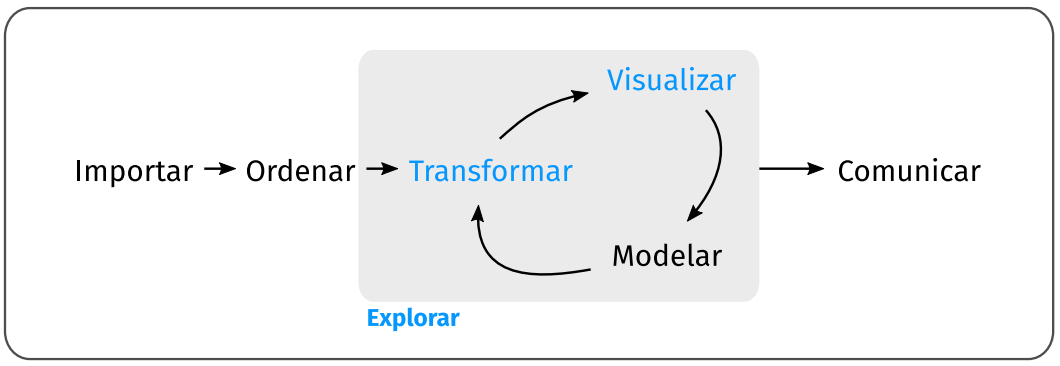
```
#>    openssl        1.4.3      2020-09-18 [1] standard (@1.4.3)
#>    pillar         1.4.7      2020-11-20 [1] standard (@1.4.7)
#>    pkgbuild       1.2.0      2020-12-15 [1] standard (@1.2.0)
#>    pkgconfig      2.0.3      2019-09-22 [1] standard (@2.0.3)
#>    pkgload        1.1.0      2020-05-29 [1] standard (@1.1.0)
#>    praise         1.0.0      2015-08-11 [1] standard (@1.0.0)
#>    prettyunits    1.1.1      2020-01-24 [1] standard (@1.1.1)
#>    processx       3.4.5      2020-11-30 [1] standard (@3.4.5)
#>  R progress       <NA>       <NA>       [?] <NA>
#>    ps             1.5.0      2020-12-05 [1] standard (@1.5.0)
#>    purrr        * 0.3.4      2020-04-17 [1] standard (@0.3.4)
#>    R6             2.5.0      2020-10-28 [1] standard (@2.5.0)
#>    RColorBrewer   1.1-2      2014-12-07 [1] standard (@1.1-2)
#>    Rcpp           1.0.6      2021-01-15 [1] standard (@1.0.6)
#>    readr        * 1.4.0      2020-10-05 [1] standard (@1.4.0)
#>    readxl         1.3.1      2019-03-13 [1] standard (@1.3.1)
#>    rematch        1.0.1      2016-04-21 [1] standard (@1.0.1)
#>    rematch2       2.1.2      2020-05-01 [1] standard (@2.1.2)
#>    reprex         0.3.0      2019-05-16 [1] standard (@0.3.0)
#>    rlang          0.4.10     2020-12-30 [1] standard (@0.4.10)
#>    rmarkdown      2.6.4      2021-01-18 [1] Github (rstudio/rmarkdown@2e8572e)
#>    rprojroot      2.0.2      2020-11-15 [1] standard (@2.0.2)
#>    rstudioapi     0.13       2020-11-12 [1] standard (@0.13)
#>    rvest          0.3.6      2020-07-25 [1] standard (@0.3.6)
#>    scales         1.1.1      2020-05-11 [1] standard (@1.1.1)
#>    selectr        0.4-2      2019-11-20 [1] standard (@0.4-2)
#>    stringi        1.5.3      2020-09-09 [1] standard (@1.5.3)
#>    stringr      * 1.4.0      2019-02-10 [1] standard (@1.4.0)
#>    sys            3.4        2020-07-23 [1] standard (@3.4)
#>    testthat       3.0.1      2020-12-17 [1] standard (@3.0.1)
#>    tibble       * 3.0.5      2021-01-15 [1] standard (@3.0.5)
#>    tidyr        * 1.1.2      2020-08-27 [1] standard (@1.1.2)
#>    tidyselect     1.1.0      2020-05-11 [1] standard (@1.1.0)
#>    tidyverse    * 1.3.0      2019-11-21 [1] standard (@1.3.0)
#>    tinytex        0.28       2020-12-14 [1] standard (@0.28)
#>    utf8           1.1.4      2018-05-24 [1] standard (@1.1.4)
#>    vctrs          0.3.6      2020-12-17 [1] standard (@0.3.6)
#>    viridisLite    0.3.0      2018-02-01 [1] standard (@0.3.0)
#>    waldo          0.2.3      2020-11-09 [1] standard (@0.2.3)
#>    whisker        0.4        2019-08-28 [1] standard (@0.4)
#>    withr          2.4.0      2021-01-16 [1] standard (@2.4.0)
#>    xfun           0.20       2021-01-06 [1] standard (@0.20)
#>    xml2           1.3.2      2020-04-23 [1] standard (@1.3.2)
#>    yaml           2.2.1      2020-02-01 [1] standard (@2.2.1)
#>
#> [1] /Users/runner/work/_temp/Library
#> [2] /Library/Frameworks/R.framework/Versions/4.0/Resources/library
#>
#>  R ── Package was removed from disk.
```

« Bienvenida

2 Introducción »

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 el modelamiento 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 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 largo plazo, en términos de que te entregan las herramientas necesarias para organizarte cuando abordes proyectos reales.

« 1 Introducción

3 Visualización de datos »

On this page

View source

Edit this page

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)
#> ── Attaching packages ─────────────────────────────────────── tidyverse 1.3.0 ──
#> ✓ ggplot2 3.3.3     ✓ purrr   0.3.4
#> ✓ tibble  3.0.5     ✓ dplyr   1.0.3
#> ✓ tidyr   1.1.2     ✓ stringr 1.4.0
#> ✓ readr   1.4.0     ✓ forcats 0.5.0
#> ── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
#> ✖ dplyr::filter() masks stats::filter()
#> ✖ dplyr::lag()    masks stats::lag()
```

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 "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 cargues el paquete `datos`, ya que en él están contenidas las versiones en español de los datos que utilizaremos en este capítulo:

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

On this page

View source
Edit this page

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, lo 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>))
```

Copy

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

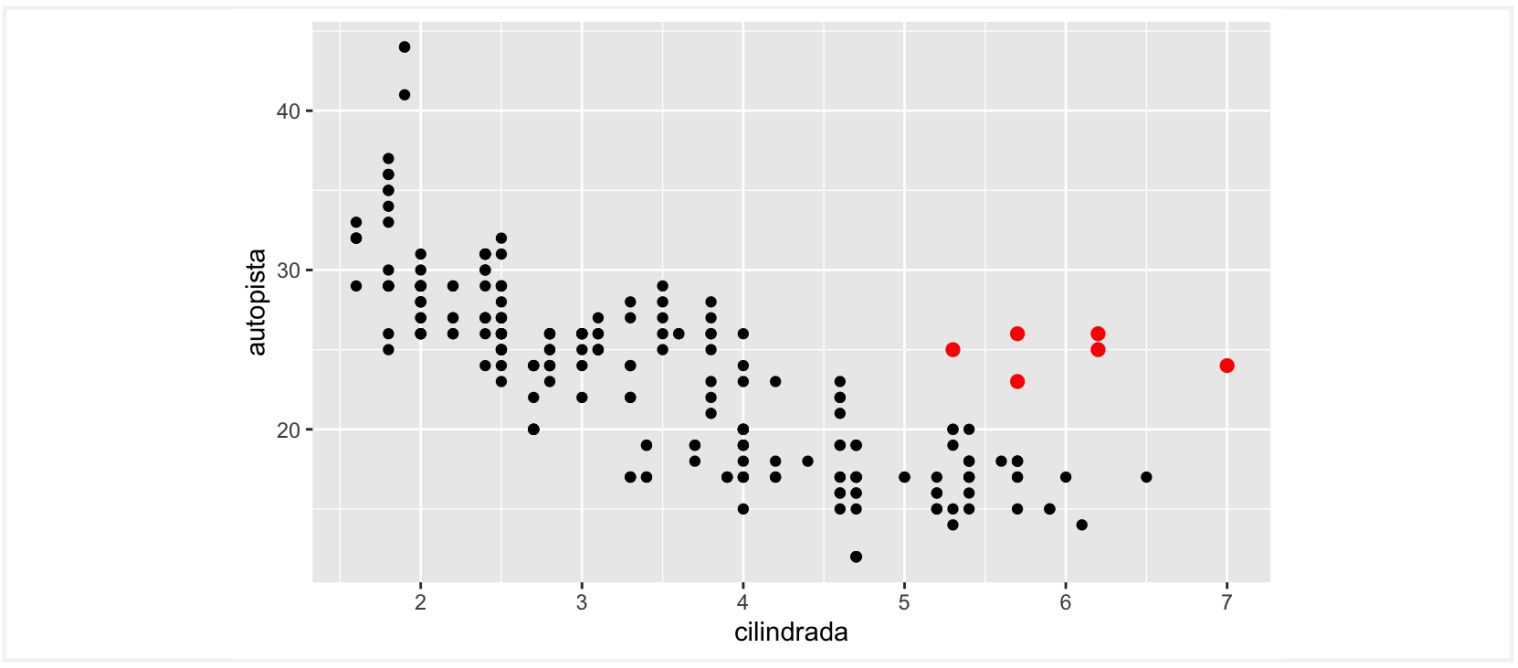
3.2.4 Ejercicios

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

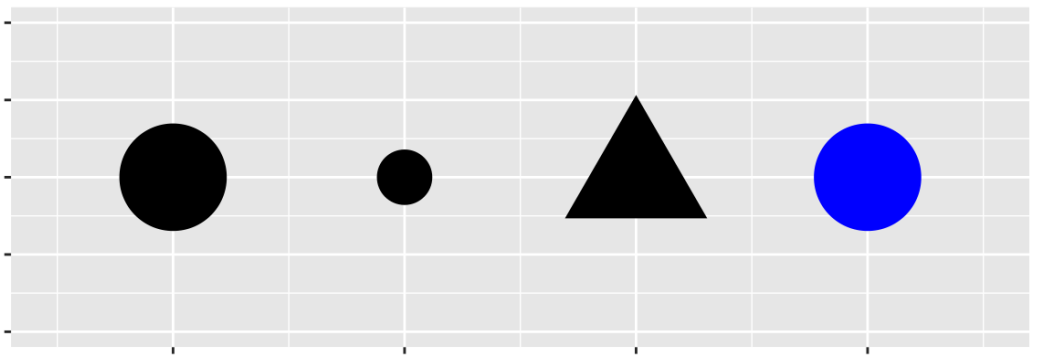
> "El mayor valor de una imagen es cuando nos obliga a observar lo que no esperábamos ver". — John Tukey

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).

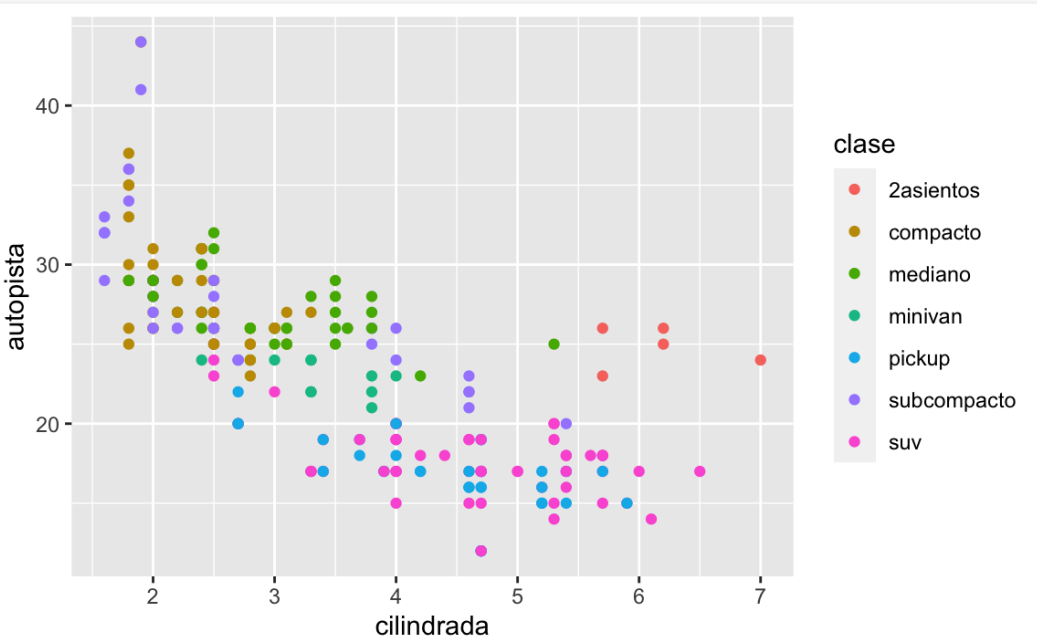
Puedes agregar una tercera variable, como `clase`, a un diagrama de dispersión bidimensional asignándolo a un parámetro **estético**. Un parámetro estético (o *estética*) es una propiedad visual de los objetos de un gráfico. Las estéticas incluye cosas como el tamaño, la forma o el color de tus puntos. 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))
```

Copy



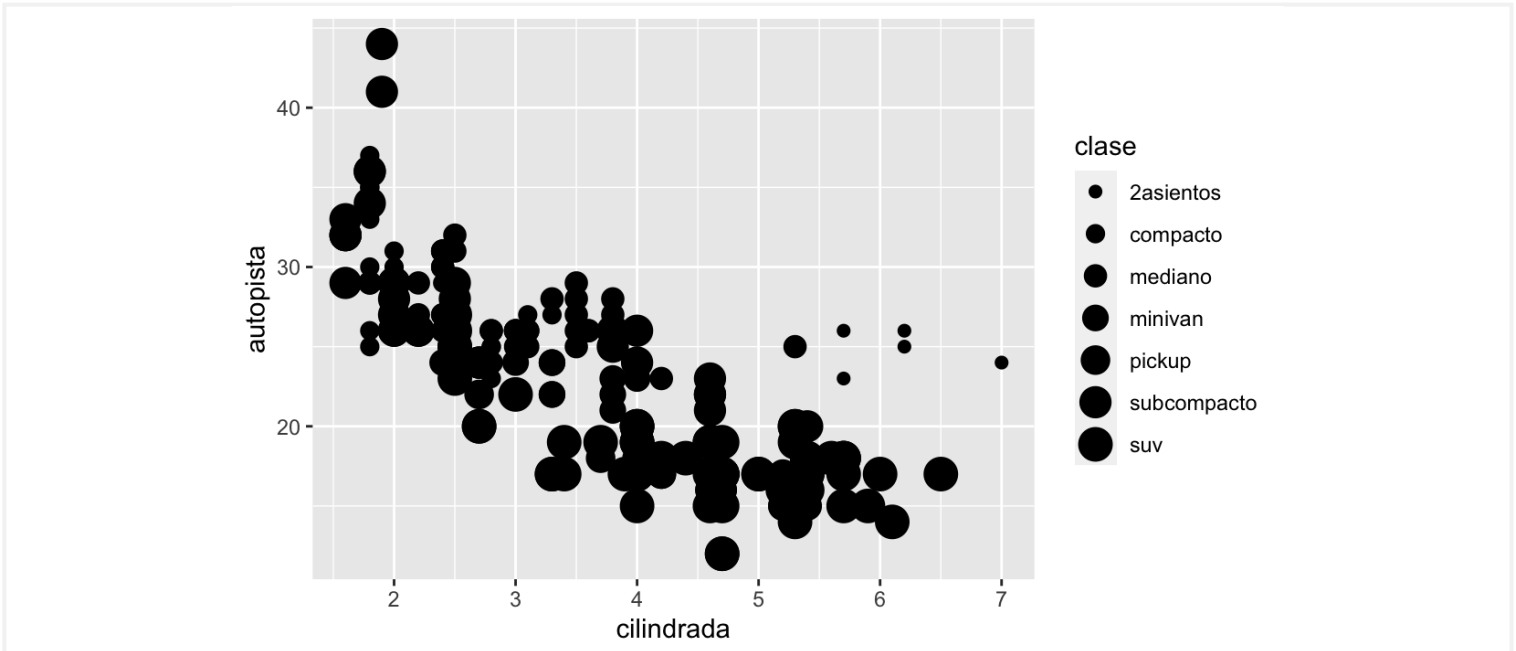
(Si prefieres el inglés británico, como Hadley, puedes usar `colour` en lugar de `color`).

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.

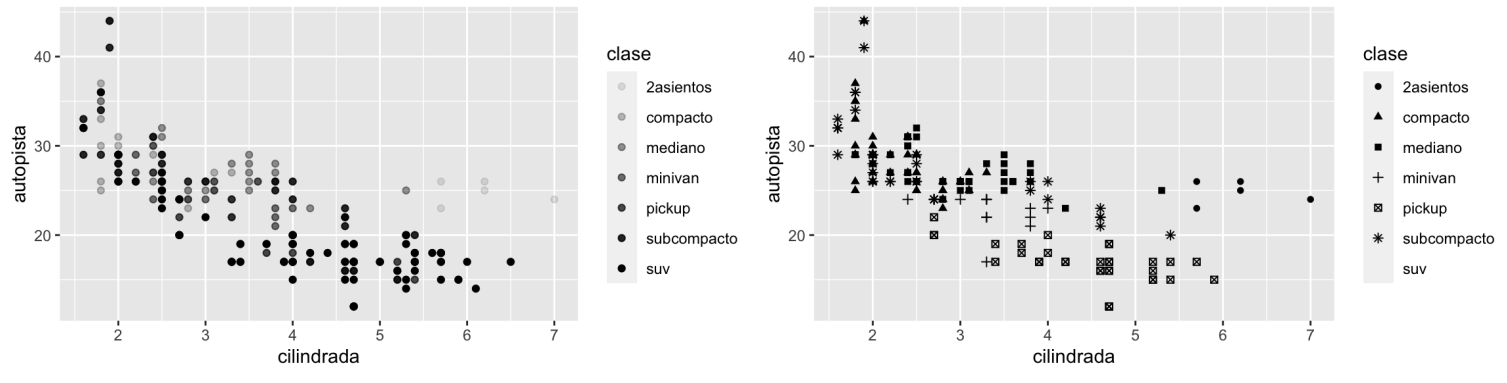
En el ejemplo anterior asignamos la variable `clase` a la estética de color, pero podríamos haberla asignado a la estética del tamaño del mismo modo. En este caso, el tamaño exacto de cada punto revelaría a qué clase pertenece. Recibimos aquí una **advertencia** (*warning*), porque mapear una variable 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))
#> Warning: Using size for a discrete variable is not advised.
```



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))
```



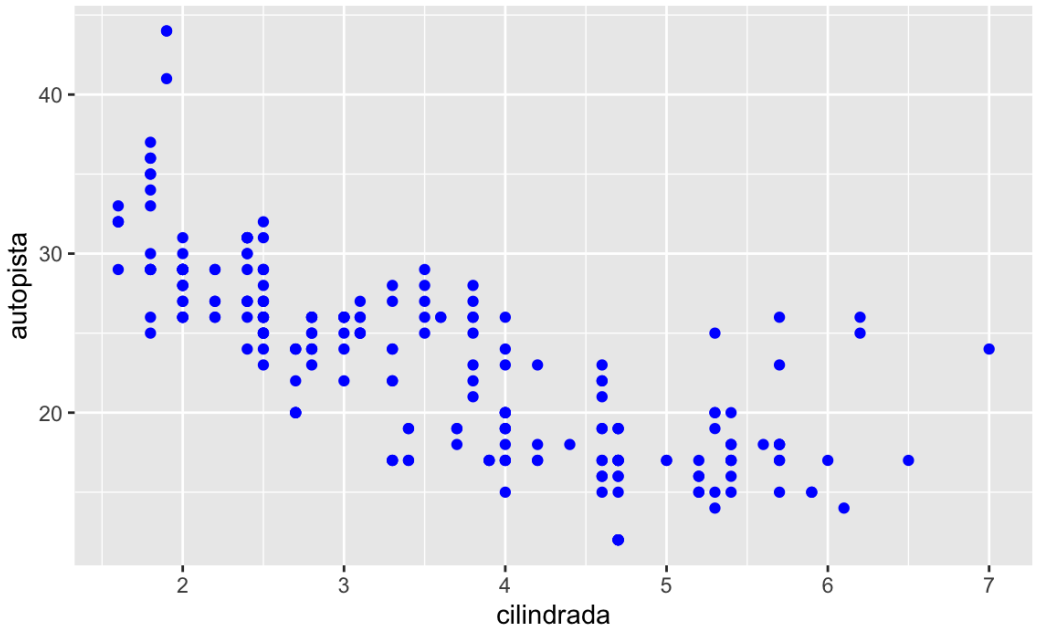
¿Qué pasó con los SUV? **ggplot2** solo puede usar seis formas a la vez. De forma predeterminada, los grupos adicionales no se grafican cuando se emplea la estética de la forma (*shape*).

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 3.1.

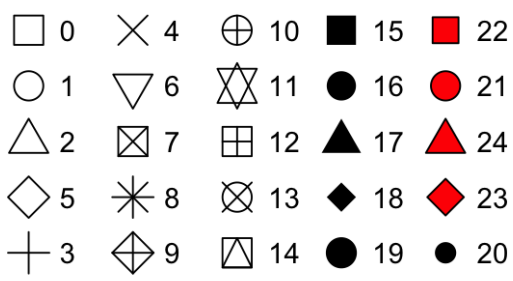


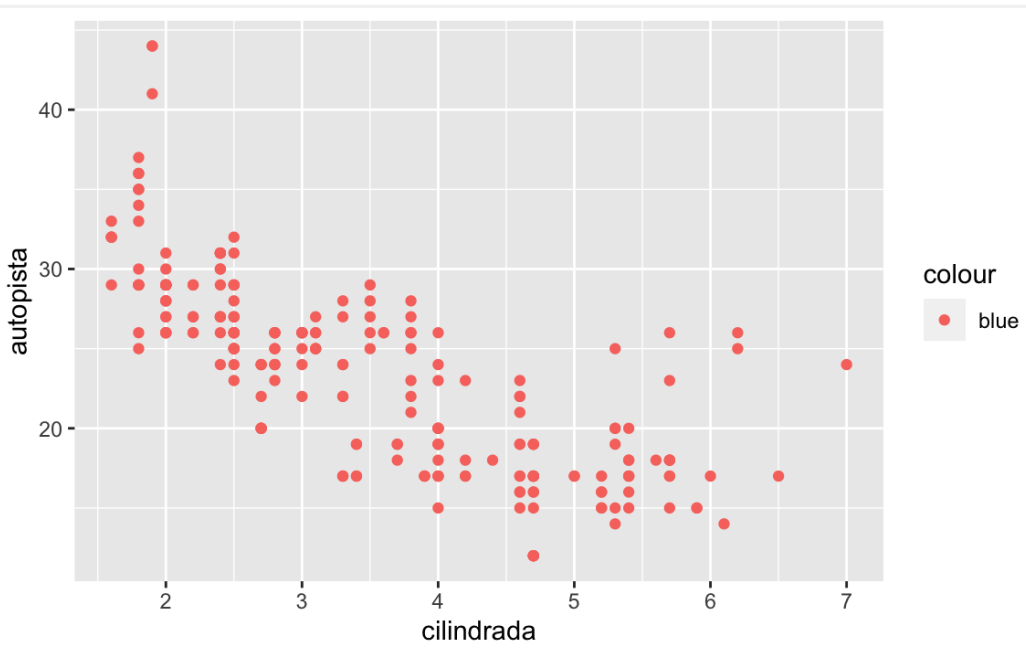
Figure 3.1: R tiene 25 formas predefinidas que están identificadas por números. Hay algunas que parecen duplicados: por ejemplo 0, 15 y 22 son todos cuadrados. La diferencia viene de la interacción entre las estéticas `color` y `fill` (*relleno*). Las formas vacías (0–14) tienen un borde determinado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

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"))
```

Copy



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))
```

Copy

Si esto no resuelve el problema, prueba con la ayuda. Puedes obtener ayuda sobre cualquier función de 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.

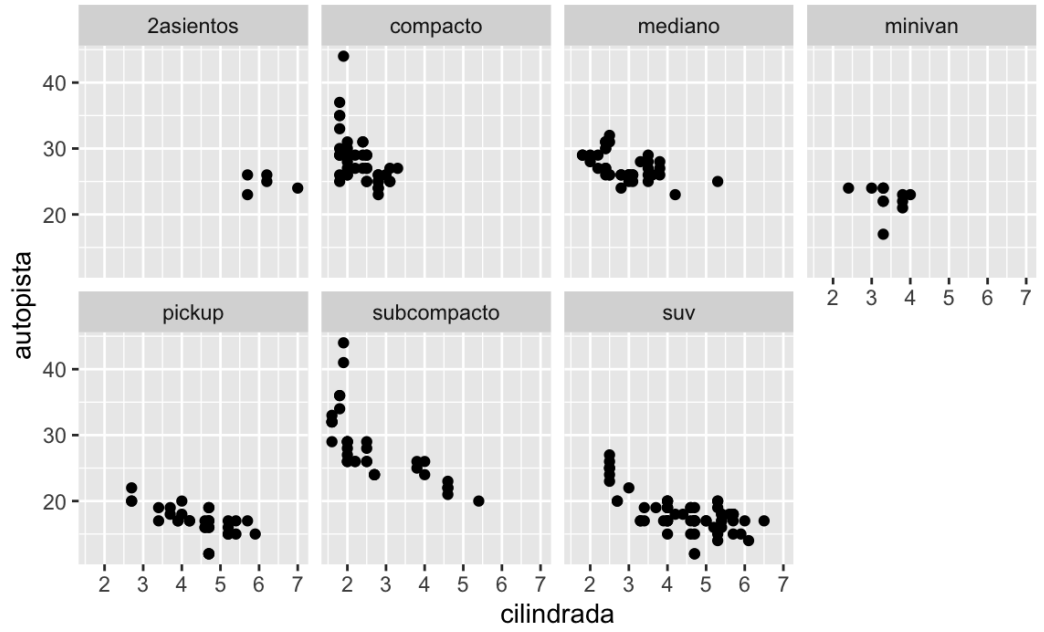
3.5 **Separar en facetas**

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 `~` seguida 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.

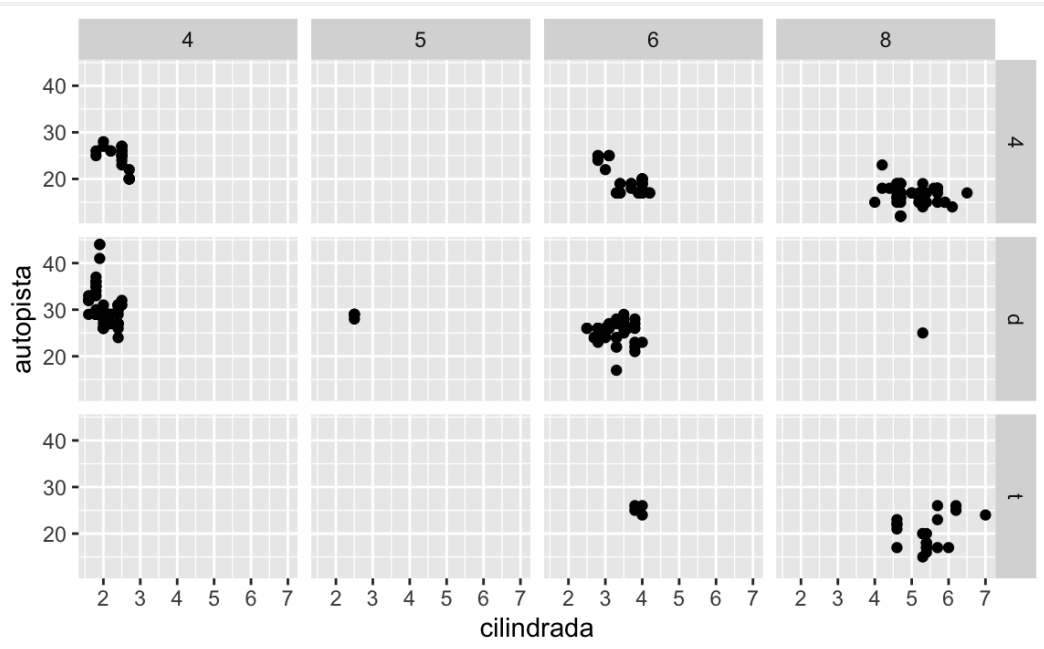
```
ggplot(data = millas) +
  geom_point(mapping = aes(x = cilindrada, y = autopista)) +
  facet_wrap(~ clase, nrow = 2)
```

Copy



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)
```

Copy

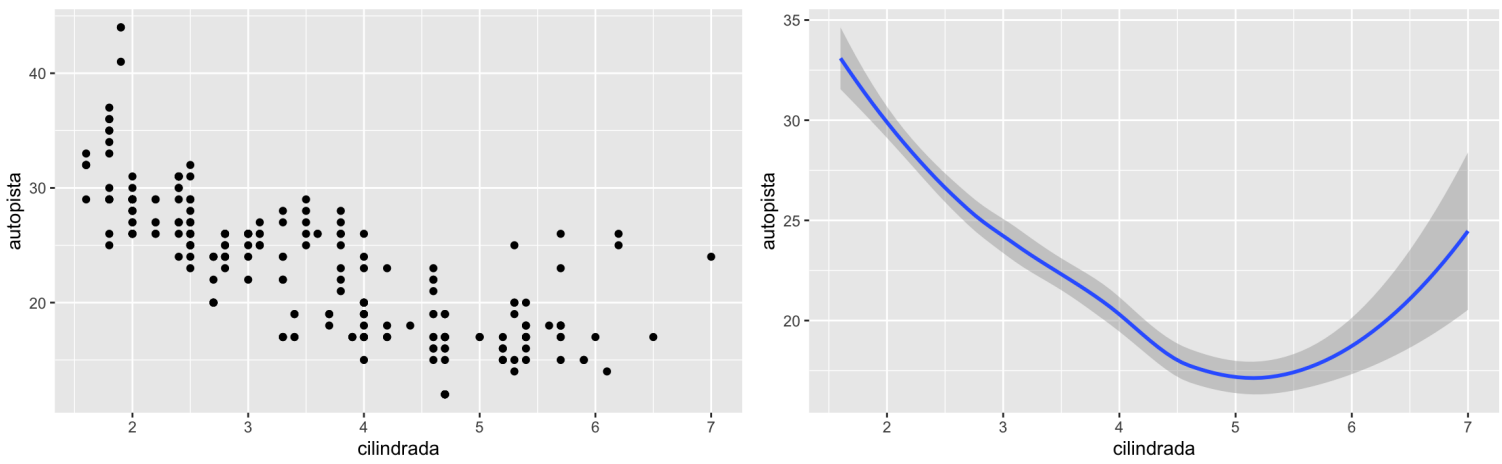
¿Cuáles son las ventajas de separar en facetas en lugar de aplicar una estética de color? ¿Cuáles son las desventajas? ¿Cómo cambiaría este balance si tuvieras un conjunto de datos más grande?

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:

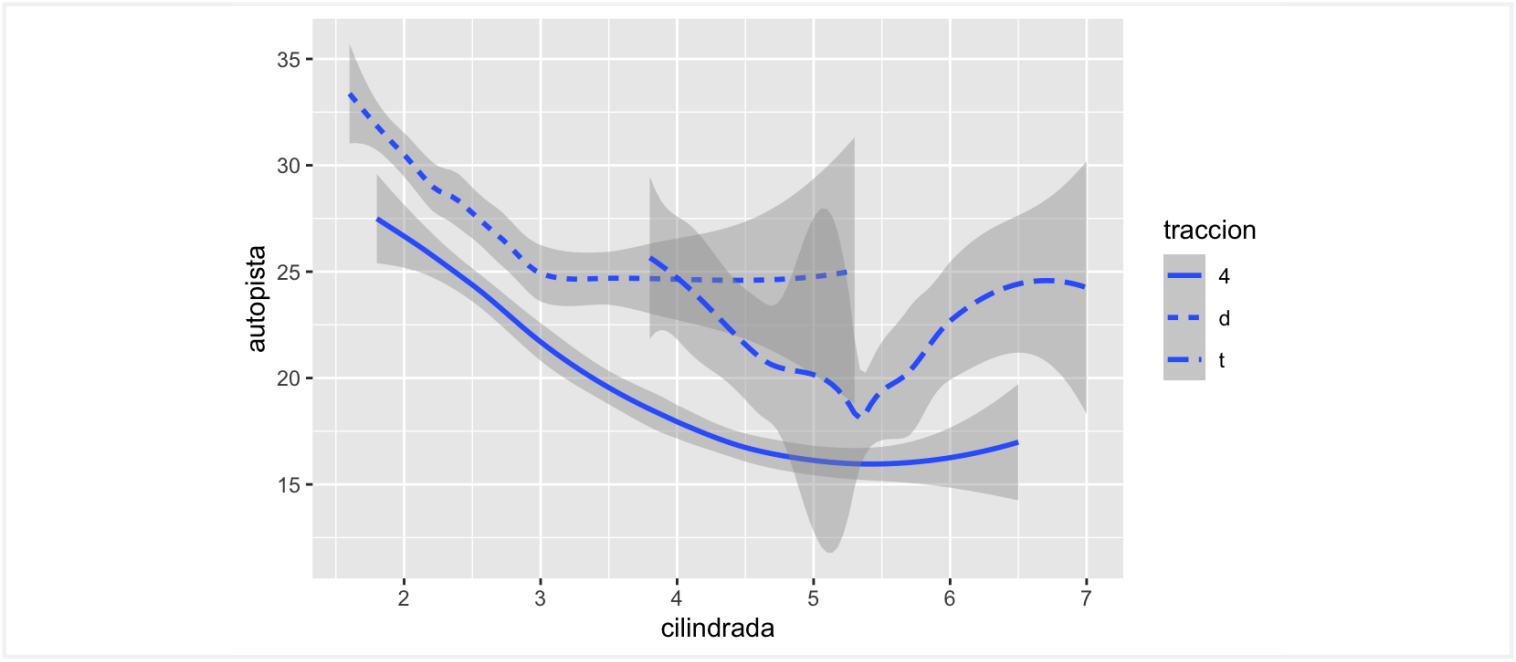
```
# izquierda
ggplot(data = millas) +
  geom_point(mapping = aes(x = cilindrada, y = autopista))

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

Copy

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*).

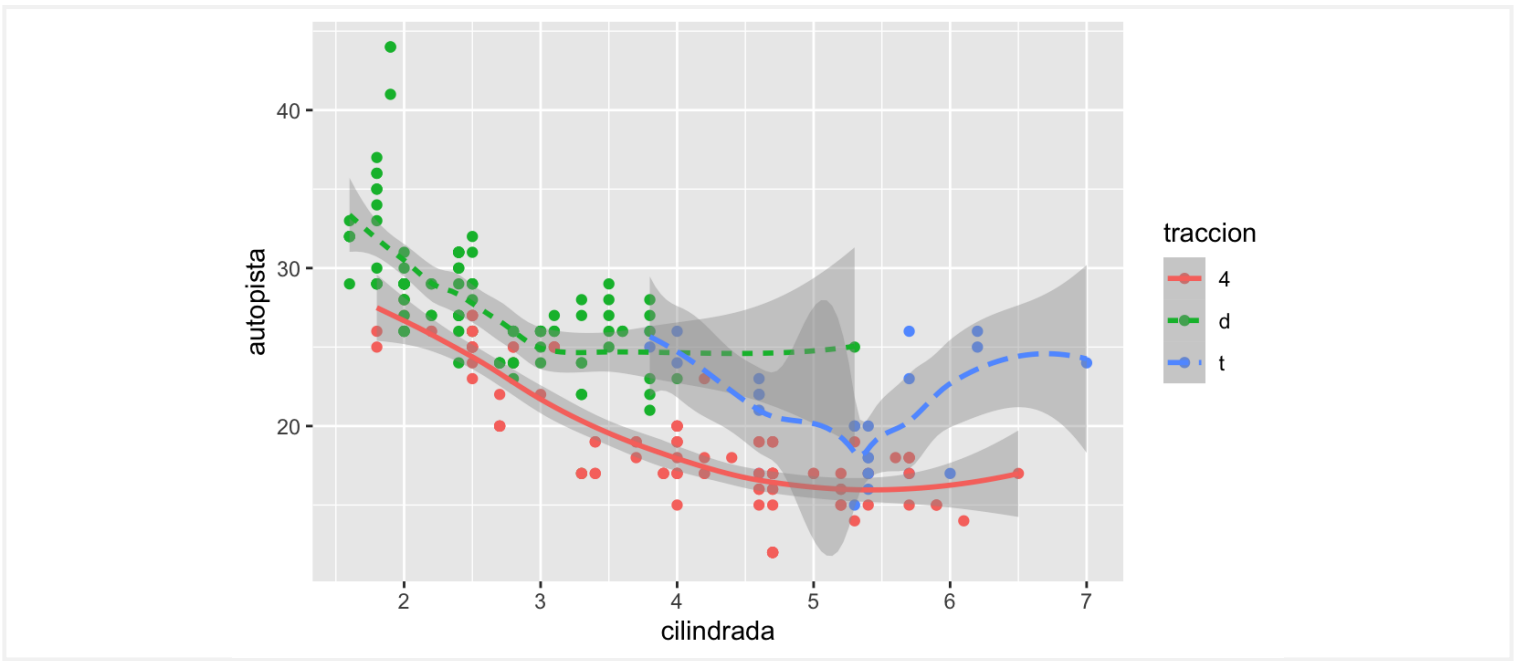
```
ggplot(data = millas) +
  geom_smooth(mapping = aes(x = cilindrada, y = autopista, linetype = traccion))
```

Copy



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`.

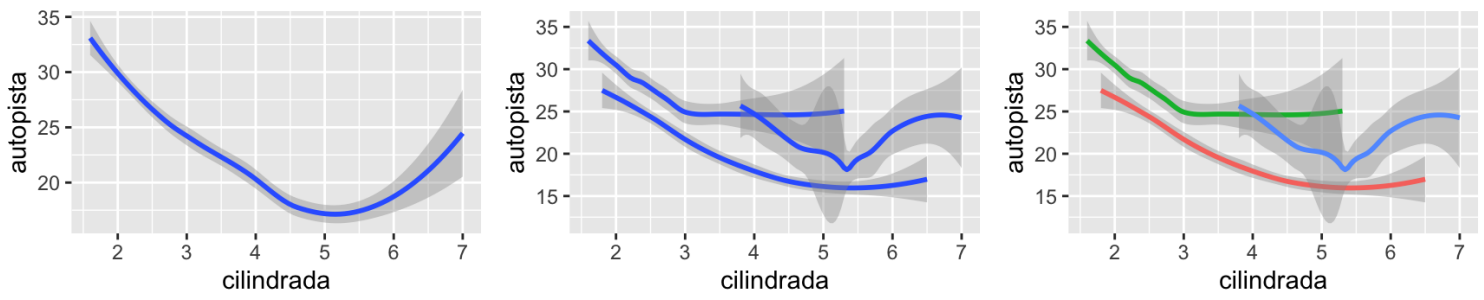


¡Observa que generamos un gráfico que contiene dos geoms! Si esto te emociona, abróchate el cinturón. En la siguiente sección aprenderemos cómo colocar múltiples geoms en el mismo gráfico.

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/ (en la parte baja de la página encontrarás la versión en español). Para obtener más información sobre un tipo dado de geoms, usa la ayuda: `?geom_smooth`.

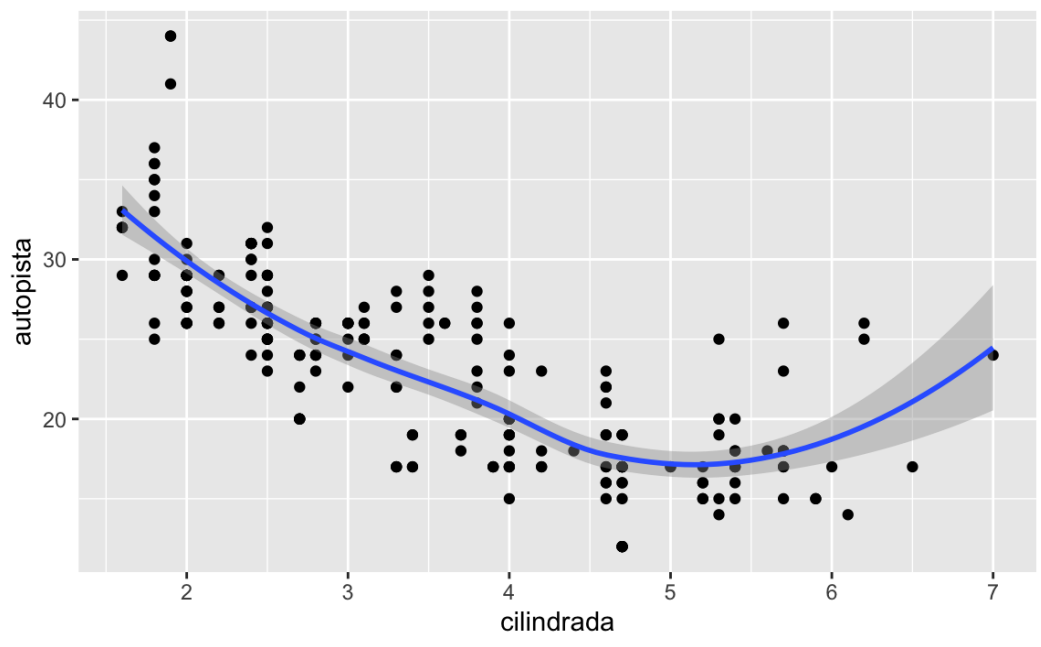
Muchos geoms, como `geom_smooth()`, usan un único objeto geométrico para mostrar múltiples filas de datos. Con estos geoms, puedes asignar la estética de `group` (**grupo**) a una variable categórica para graficar múltiples objetos. **ggplot2** representará un objeto distinto por cada valor único de 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.

```r
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()`:

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

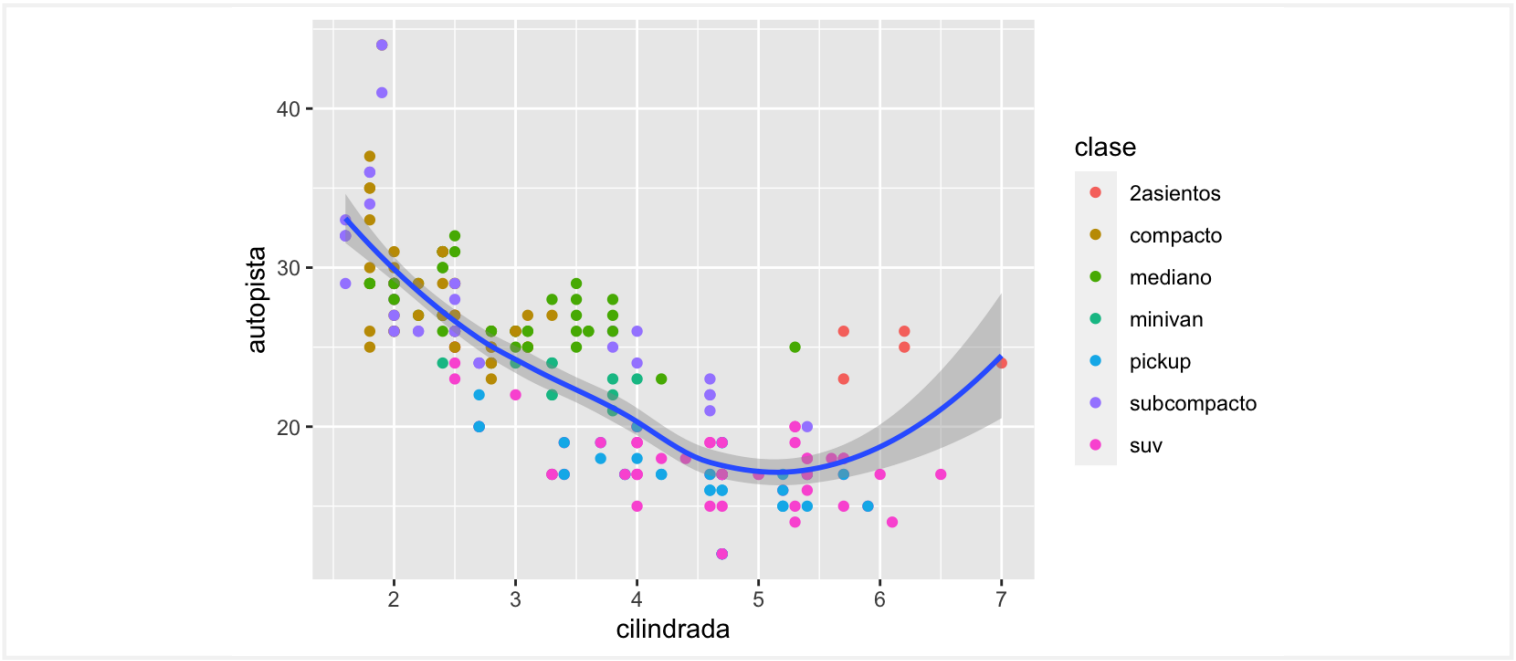


Esto introduce, sin embargo, cierta duplicación en nuestro código. Imagina que deseas cambiar el eje y para mostrar `cilindrada` en lugar de `autopista`. Necesitarías cambiar la variable en dos lugares y podrías olvidarte de actualizar uno. Puedes evitar este tipo de repetición pasando un conjunto de 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:

```r
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.

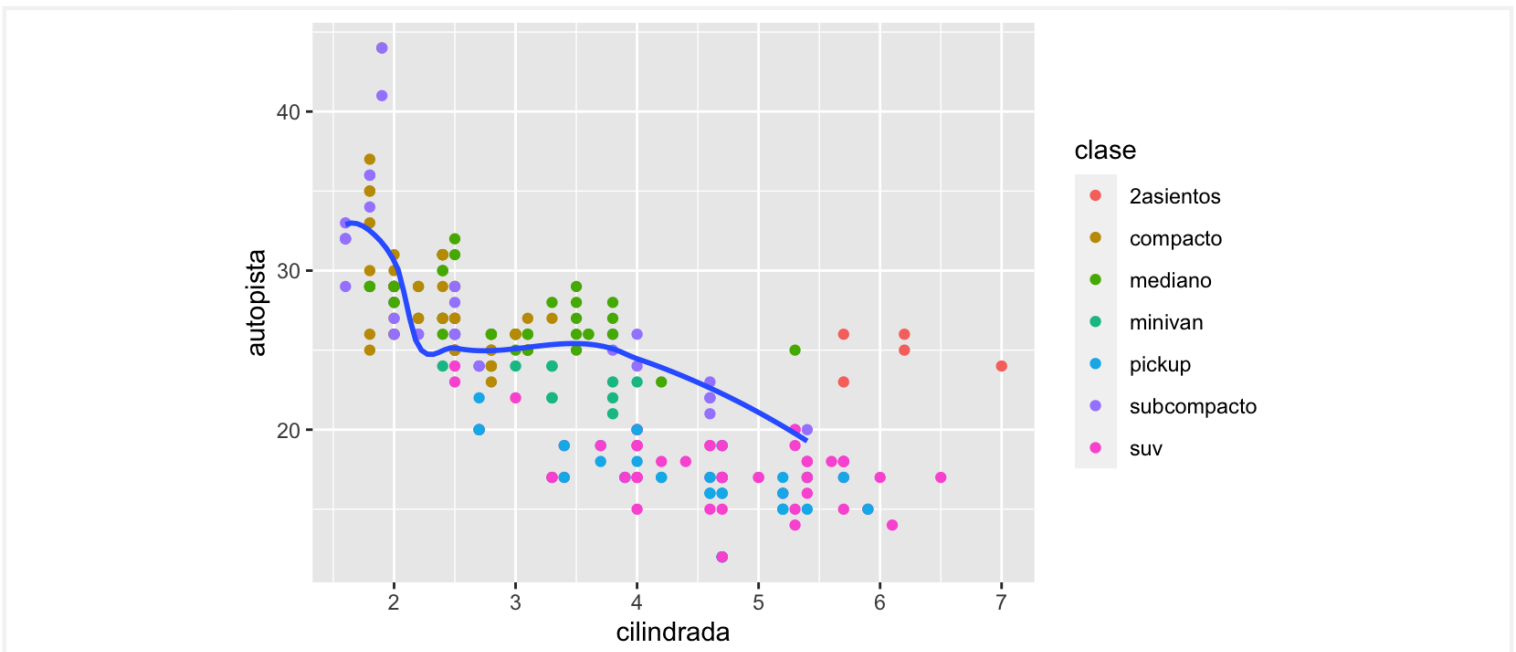
```r
ggplot(data = millas, mapping = aes(x = cilindrada, y = autopista)) +
  geom_point(mapping = aes(color = clase)) +
  geom_smooth()
```



La misma idea se puede emplear para especificar distintos conjuntos de datos (`data`) para cada capa. En el siguiente caso, 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)
```

Copy



(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 = traccion)) +
  geom_point() +
  geom_smooth(se = FALSE)
```

Copy

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

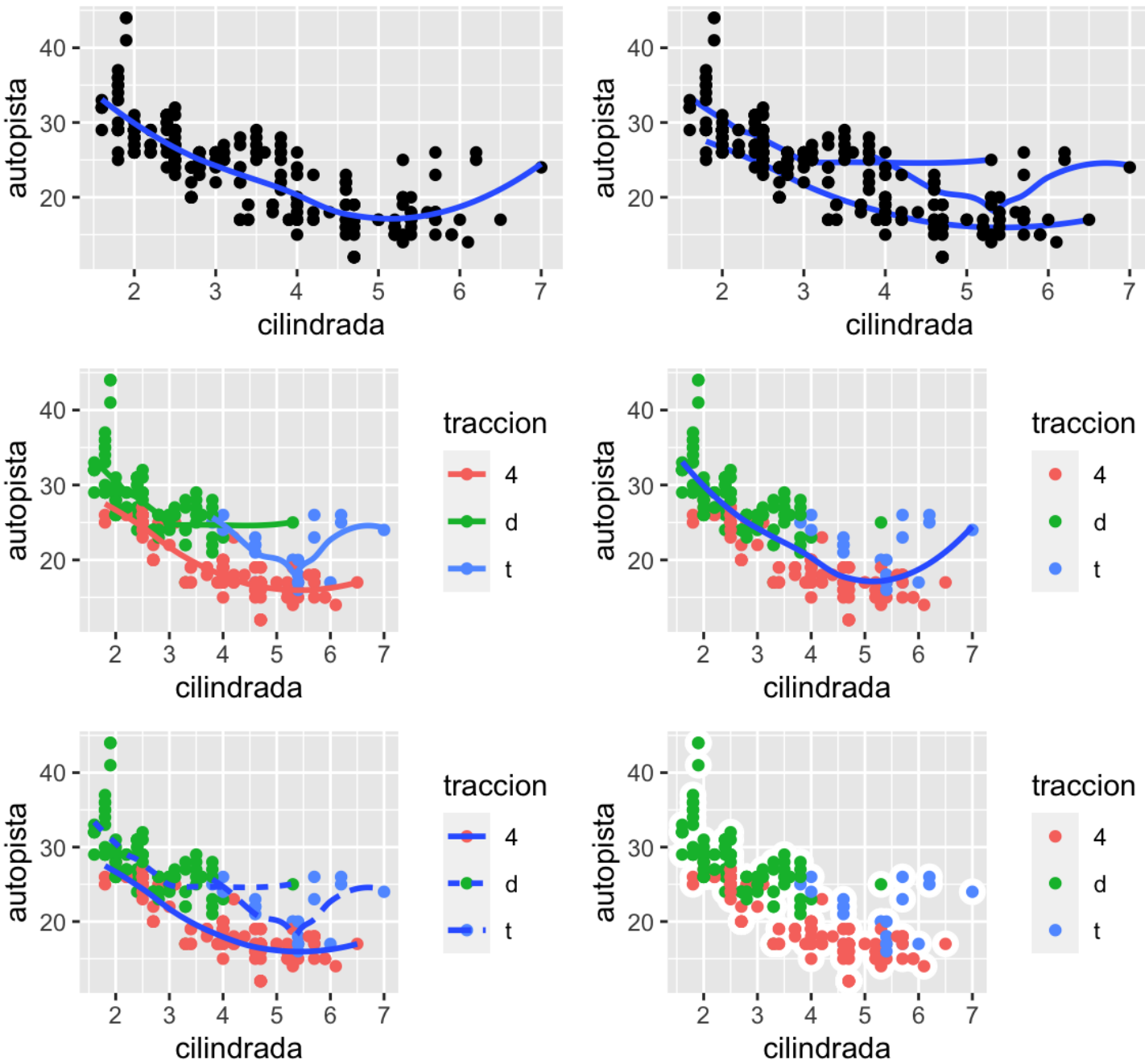
4. ¿Qué hace el argumento `se` en `geom_smooth()` ?

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))
```

Copy

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

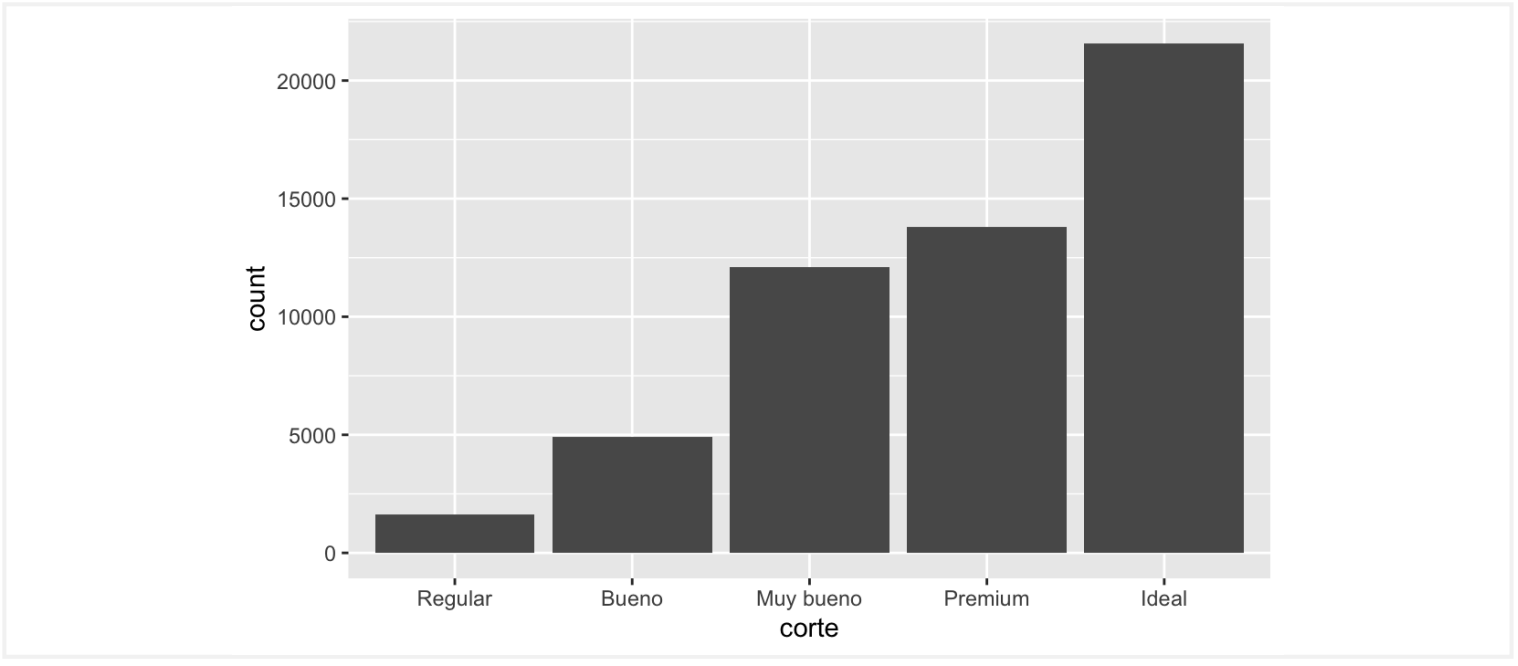


3.7 Transformaciones estadísticas

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))
```

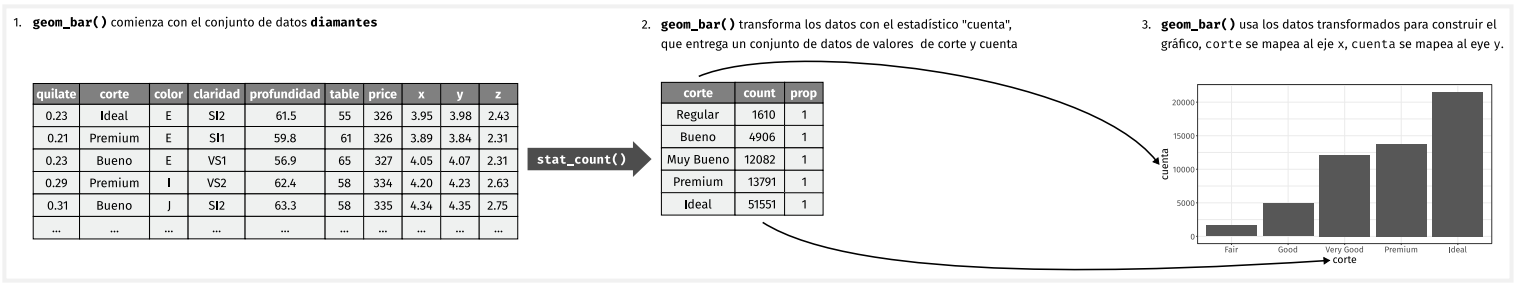
Copy



En el eje x, el gráfico muestra `corte`, una variable de `diamantes`. En el eje y muestra el 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 (*bins*), 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()`.

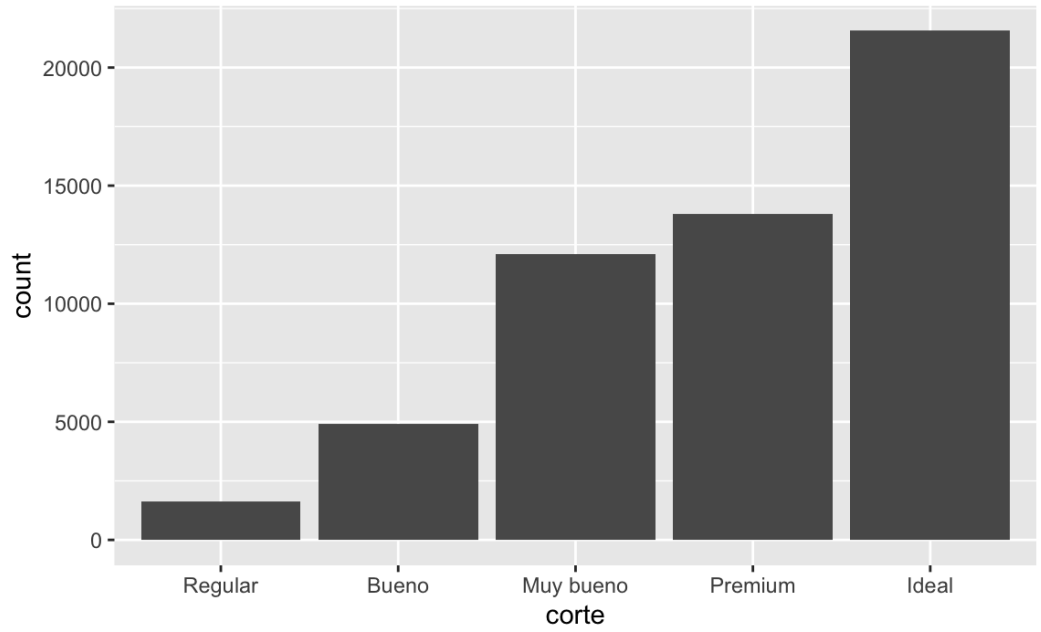


Puedes aprender acerca de qué *stat* usa cada geom inspeccionando el valor predeterminado para el argumento `stat`. Por ejemplo, `?geom_bar` muestra que el valor predeterminado para `stat` es "count", lo que significa que `geom_bar()` usa `stat_count()`. `stat_count()` está documentado en la misma página 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))
```

Copy

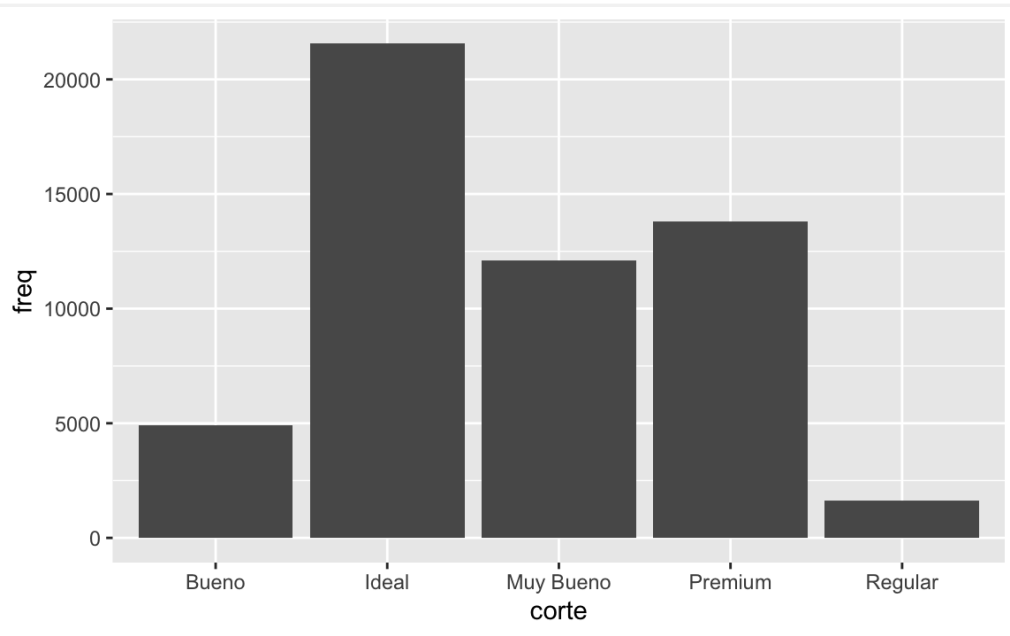


Esto funciona porque cada geom tiene una estadística predeterminada y cada estadística tiene un geom predeterminado. Esto significa que generalmente puedes usar geoms sin preocuparte por la transformación estadística subyacente.

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.

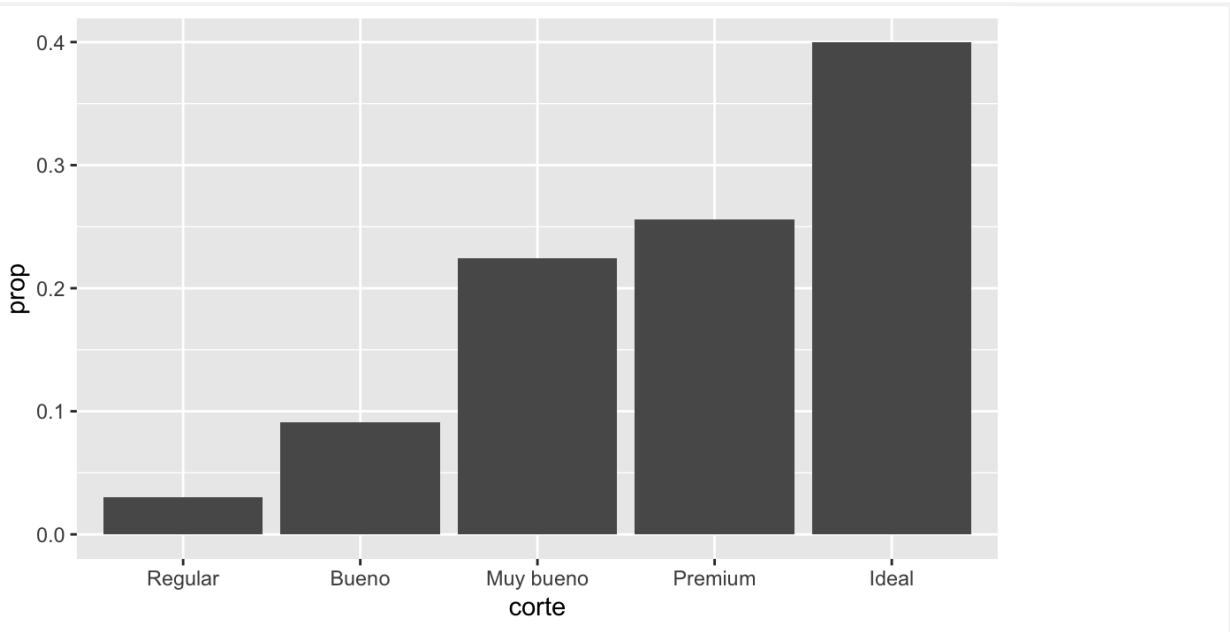
```
demo <- tribble(
  ~corte,      ~freq,
  "Regular",    1610,
  "Bueno",      4906,
  "Muy Bueno", 12082,
  "Premium",    13791,
  "Ideal",      21551
)

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:

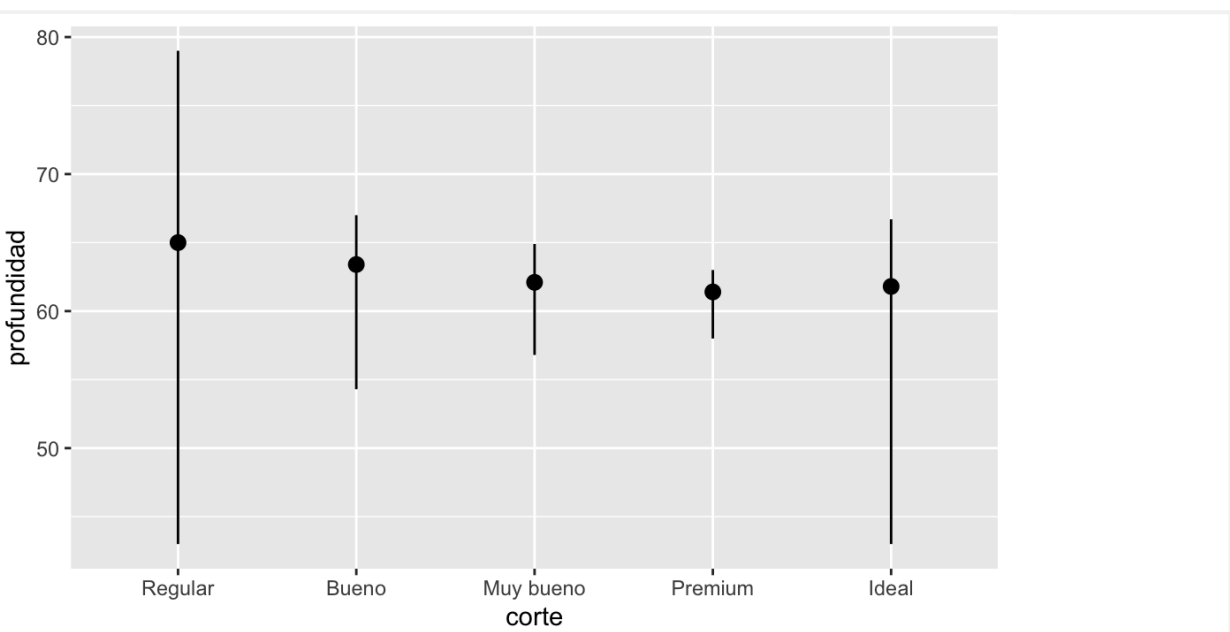
```
ggplot(data = diamantes) +
  geom_bar(mapping = aes(x = corte, y = stat(prop), group = 1))
```



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..))
```

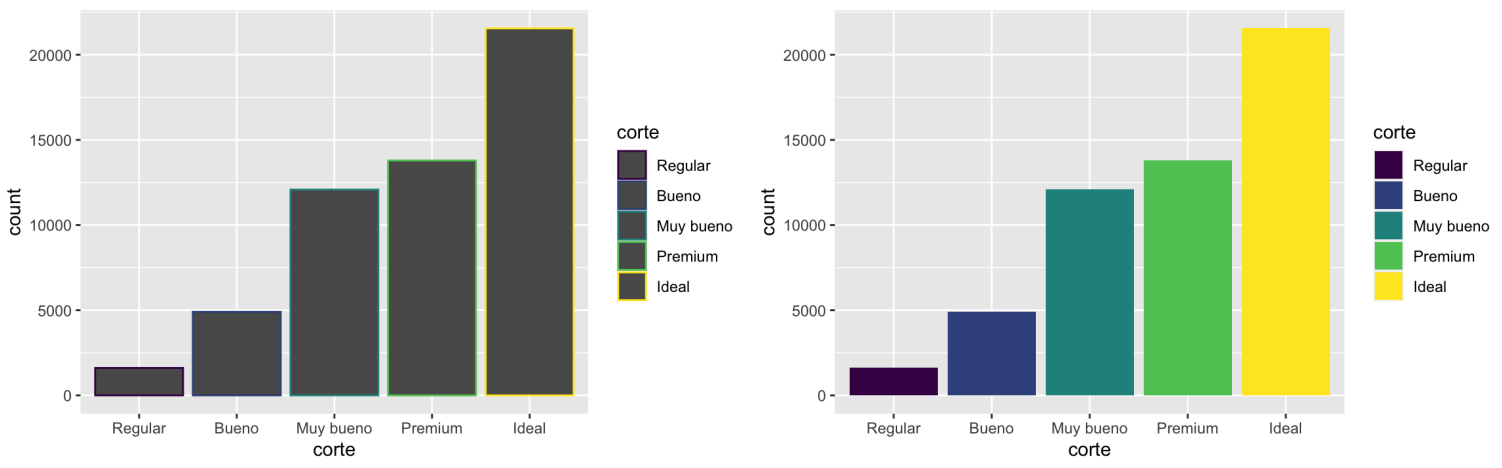
Copy

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))
```

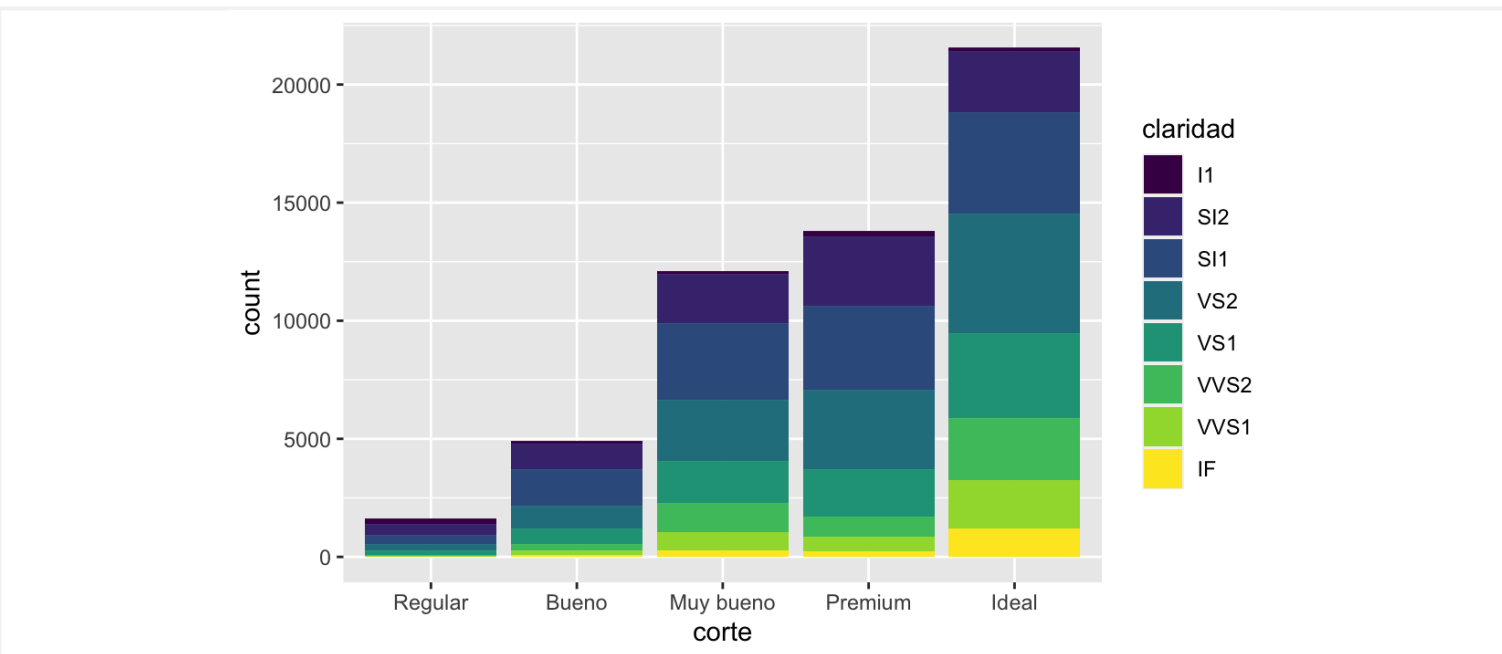
Copy



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))
```

Copy

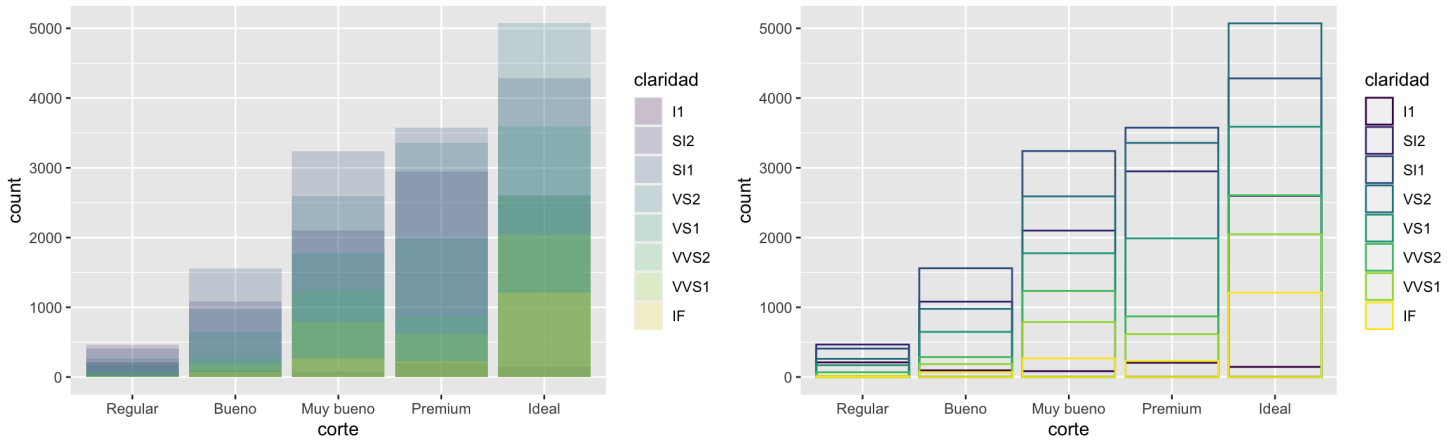


El apilamiento se realiza automáticamente mediante el ajuste de posición especificado por el argumento `position`. Si no deseas un gráfico de barras apiladas (`"stack"`), puedes usar una de las otras tres opciones: `"identity"`, `"dodge"` o `"fill"`.

- `position = "identity"` 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")
```

Copy

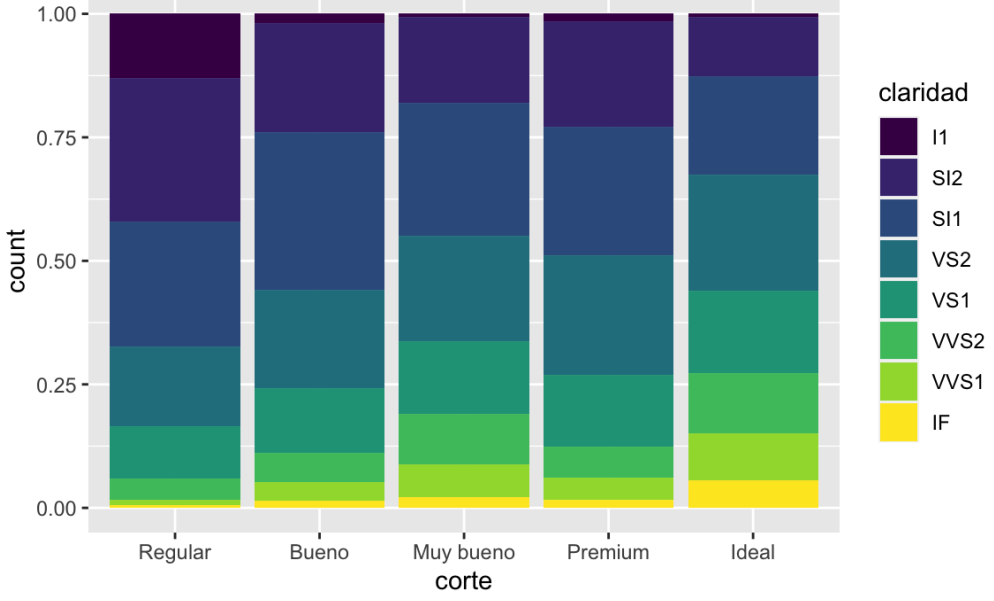


El ajuste de `position = identity` es más útil para geoms 2D, como puntos, donde es la opción predeterminada.

- `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")
```

Copy

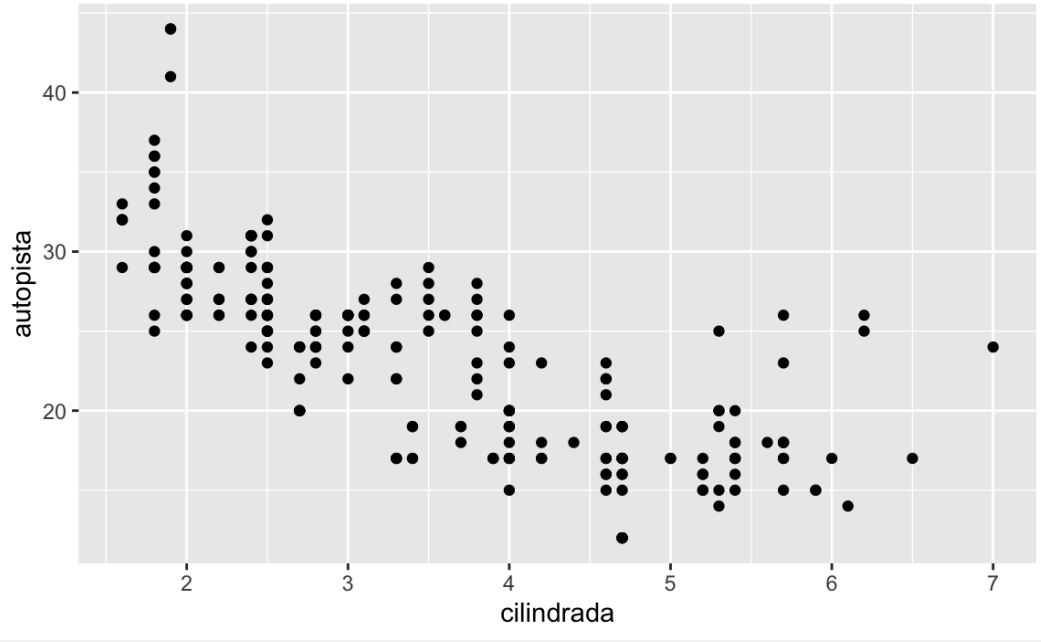


- `position = "dodge"` coloca los objetos superpuestos uno al lado del otro. Esto hace que sea más fácil comparar valores individuales.

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

Copy

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?

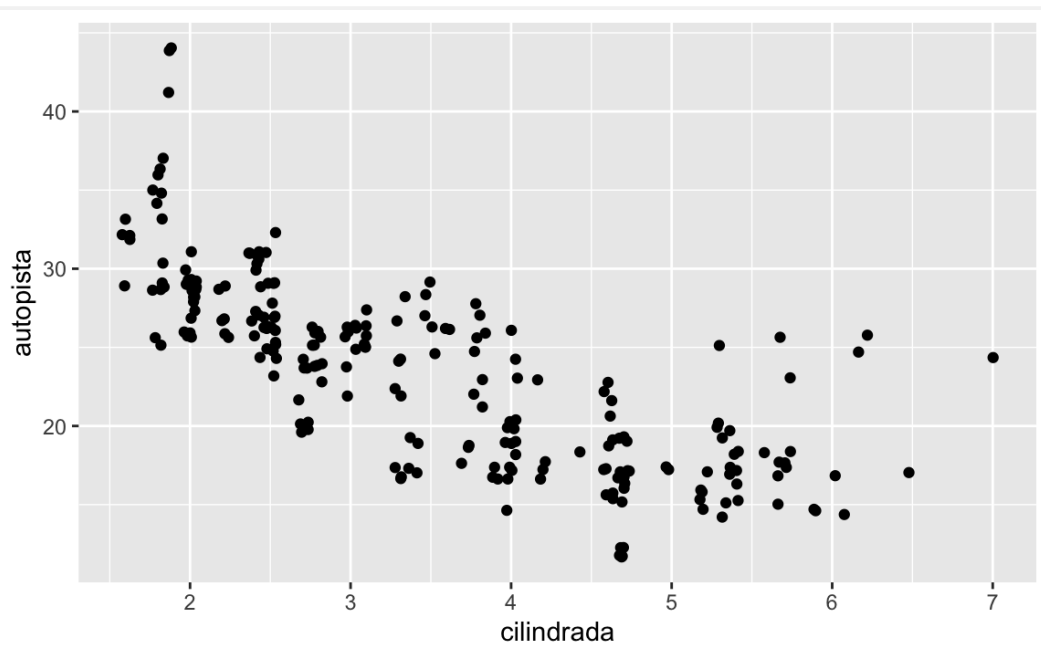


Los valores de las variables `autopista` y `cilindrada` se redondean de modo que los puntos aparecen en una cuadrícula y muchos se superponen entre sí. Este problema se conoce como **solapamiento** (*overplotting*). Esta disposición hace que sea difícil ver dónde está la masa de datos. ¿Los puntos de 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")
```

Copy



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()`.

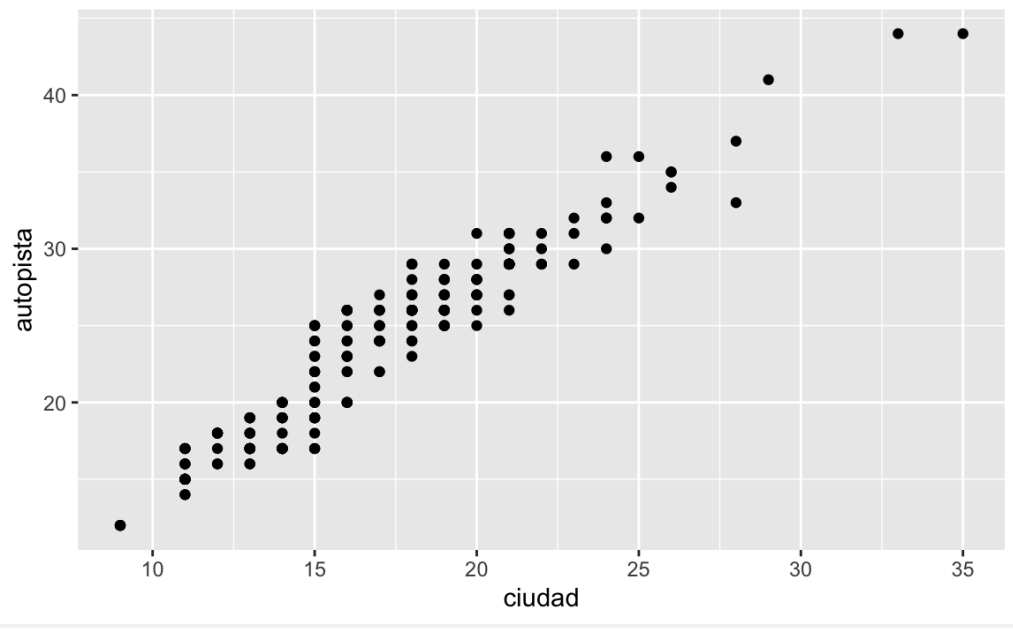
Para obtener más información sobre ajustes de posición, busca la página de ayuda asociada con cada ajuste: `?position_dodge`, `?position_fill`, `?position_identity`, `?position_jitter` y `?position_stack`.

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()
```

Copy



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.

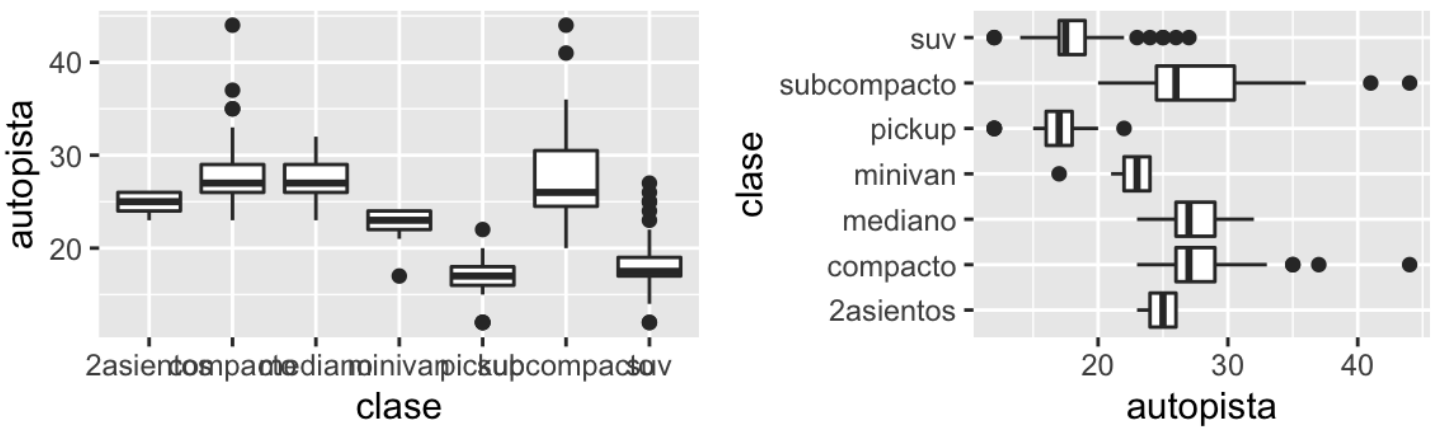
3.9 Sistemas de coordenadas

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()
```

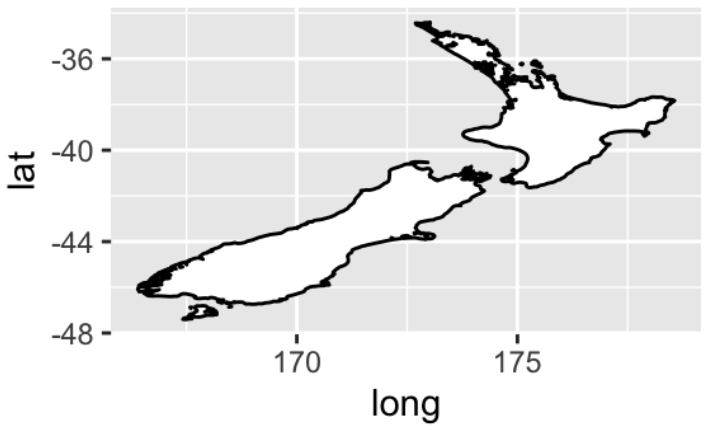
Copy

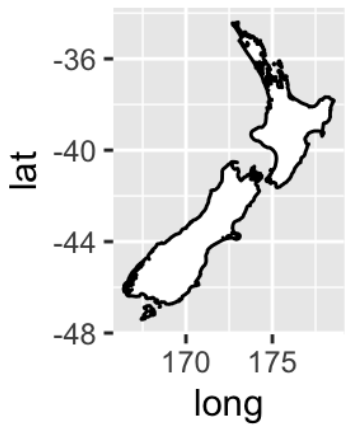


- `coord_quickmap()` establece correctamente la relación de aspecto para los mapas. Esto es muy importante si gráficas datos espaciales con ggplot2 (tema para el que, desafortunadamente, no contamos con espacio para desarrollar en este libro).

```
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()
```

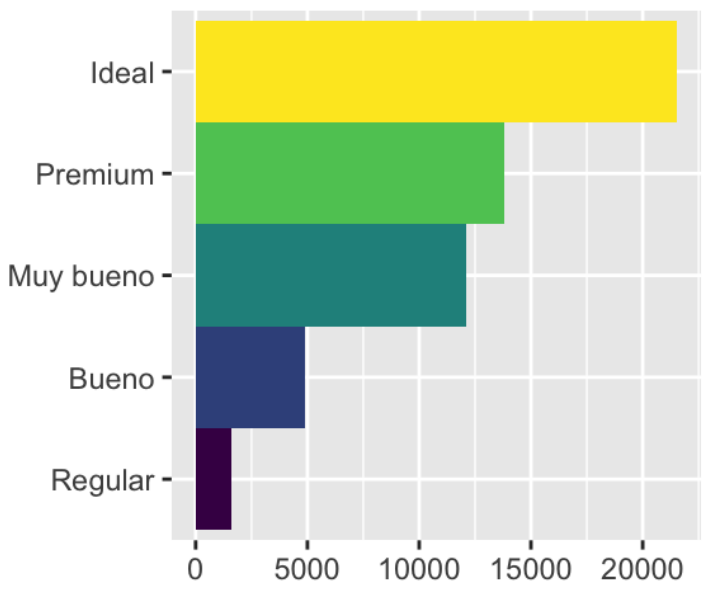
Copy

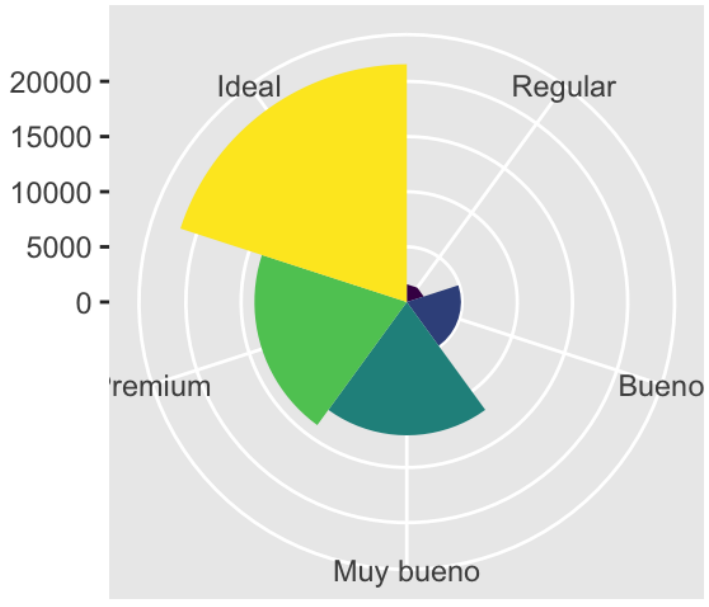




- `coord_polar()` usa coordenadas polares. Las coordenadas polares revelan una conexión interesante entre un gráfico de barras y un gráfico de Coxcomb.

```
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**

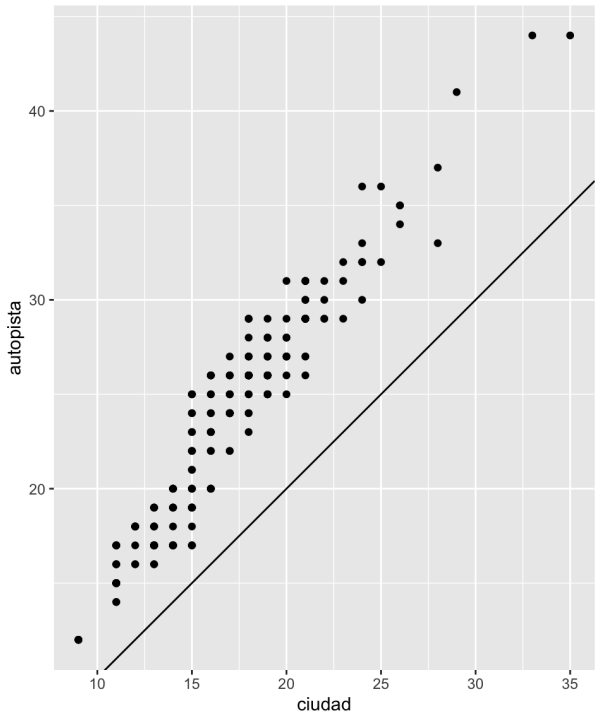
1. Convierte un gráfico de barras apiladas en un gráfico circular usando `coord_polar()`.

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>
```

Copy

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`**

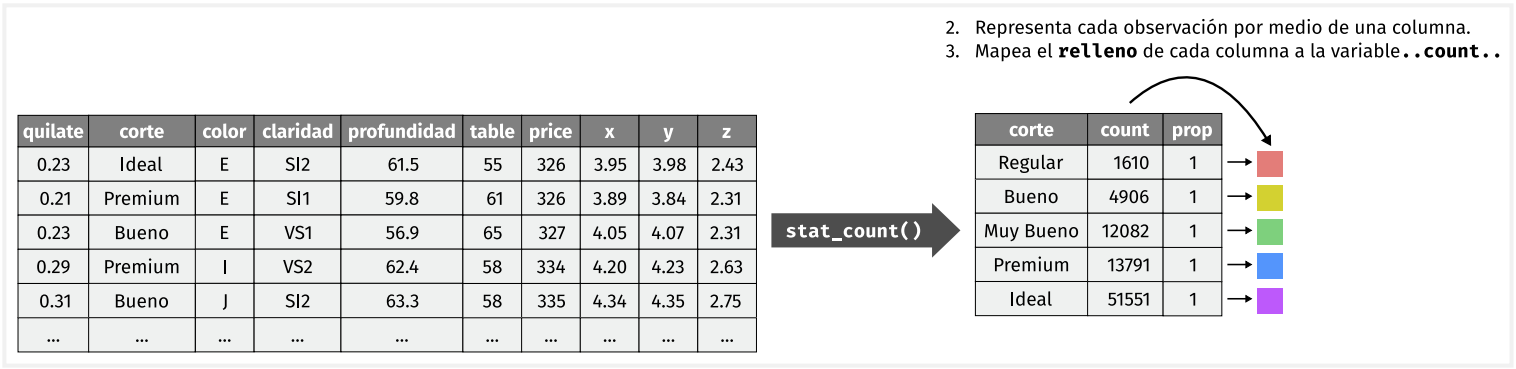
quilate	corte	color	claridad	profundidad	table	price	x	y	z
0.23	Ideal	E	SI2	61.5	55	326	3.95	3.98	2.43
0.21	Premium	E	SI1	59.8	61	326	3.89	3.84	2.31
0.23	Bueno	E	VS1	56.9	65	327	4.05	4.07	2.31
0.29	Premium	I	VS2	62.4	58	334	4.20	4.23	2.63
0.31	Bueno	J	SI2	63.3	58	335	4.34	4.35	2.75
...	...	...	...	...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

`stat_count()`

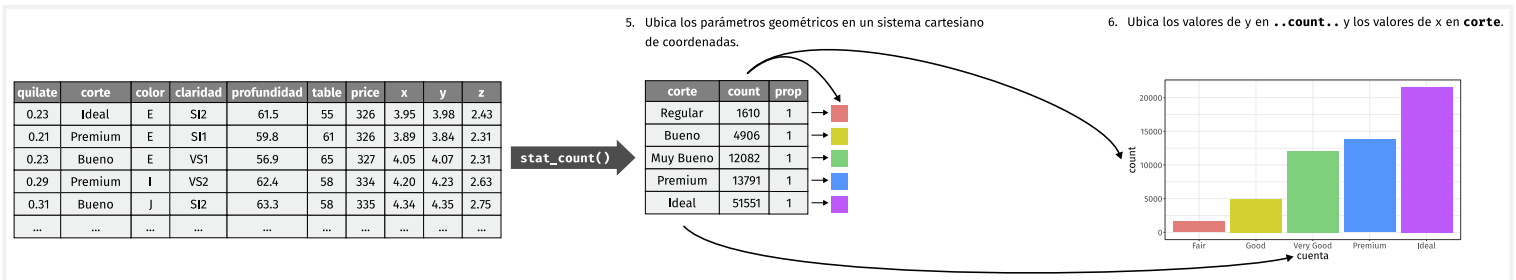
2. Calcula la cuenta para cada valor de corte con **`stat_count()`**.

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.



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.



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.

« 2 Introducción

4 Flujo de trabajo: conocimientos básicos »

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] 44.66667
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.

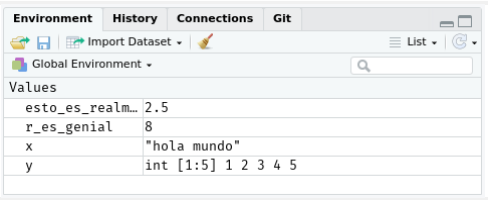
Cuando realizas una asignación no se imprime en la consola el valor asignado. Es una tentación confirmar inmediatamente el resultado:

```
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
```

Nota de la traducción: El código anterior falla (y es lo esperado) en entornos Linux/Mac, no así en Windows

¡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?

« 3 Visualización de datos

5 Transformación de datos »

5 Transformación de datos

5.1 Introducción

La visualización es una herramienta importante para la generación de conocimiento; sin embargo, es raro que obtengas los datos exactamente en la forma en 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 Prerrequisitos

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** como ayuda para comprender los datos.

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

Copy

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>
```

Copy

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:

On this page

View source
Edit this page

- `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. Sumerjámonos 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>
```

Copy

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)
```

Copy

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>
```

Copy

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`.
#> ✖ Input `..1` is named.
#> ℹ This usually means that you've used `=` instead of `==`.
#> ℹ Did you mean `mes == 1`?
```

Copy

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

```
sqrt(2)^2 == 2
#> [1] FALSE
1 / 49 * 49 == 1
#> [1] FALSE
```

Copy

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):

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

Copy

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.

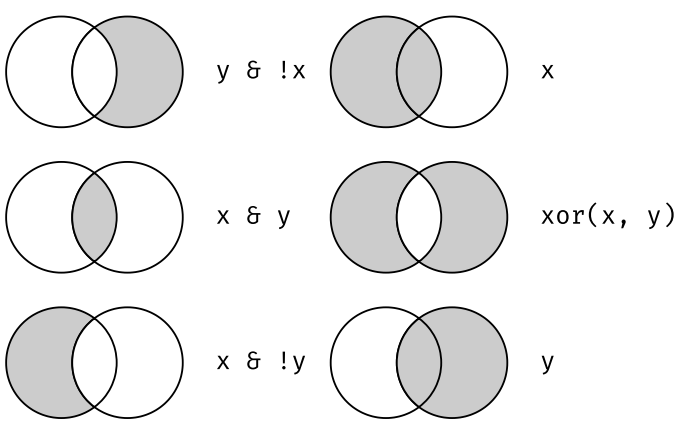


Figure 5.1: Set completo de operadores booleanos. `x` es el círculo de la derecha, `y` es el círculo de la izquierda y la región sombreada qué partes selecciona cada operador.

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

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

Copy

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))
```

Copy

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)
```

Copy

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 `NA`s (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:

    1. Tuvieron un retraso de llegada de dos o más horas

    2. Volaron a Houston (`IAH` o `HOU`)

    3. Fueron operados por United, American o Delta

    4. Partieron en invierno del hemisferio sur (julio, agosto y septiembre)

5. Llegaron más de dos horas tarde, pero no salieron tarde
6. Se retrasaron por lo menos una hora, pero repusieron más de 30 minutos en vuelo
7. Partieron entre la medianoche y las 6 a.m. (incluyente)

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

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>
```

Copy

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>
```

Copy

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
```

Copy

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>
```

Copy

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("ijk")`: 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>
```

Copy

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>
```

Copy

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")
```

Copy

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"))
```

Copy

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,
  ganancia_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>,
#> #   ganancia_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 partes. 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 las 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] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5
```

- 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:

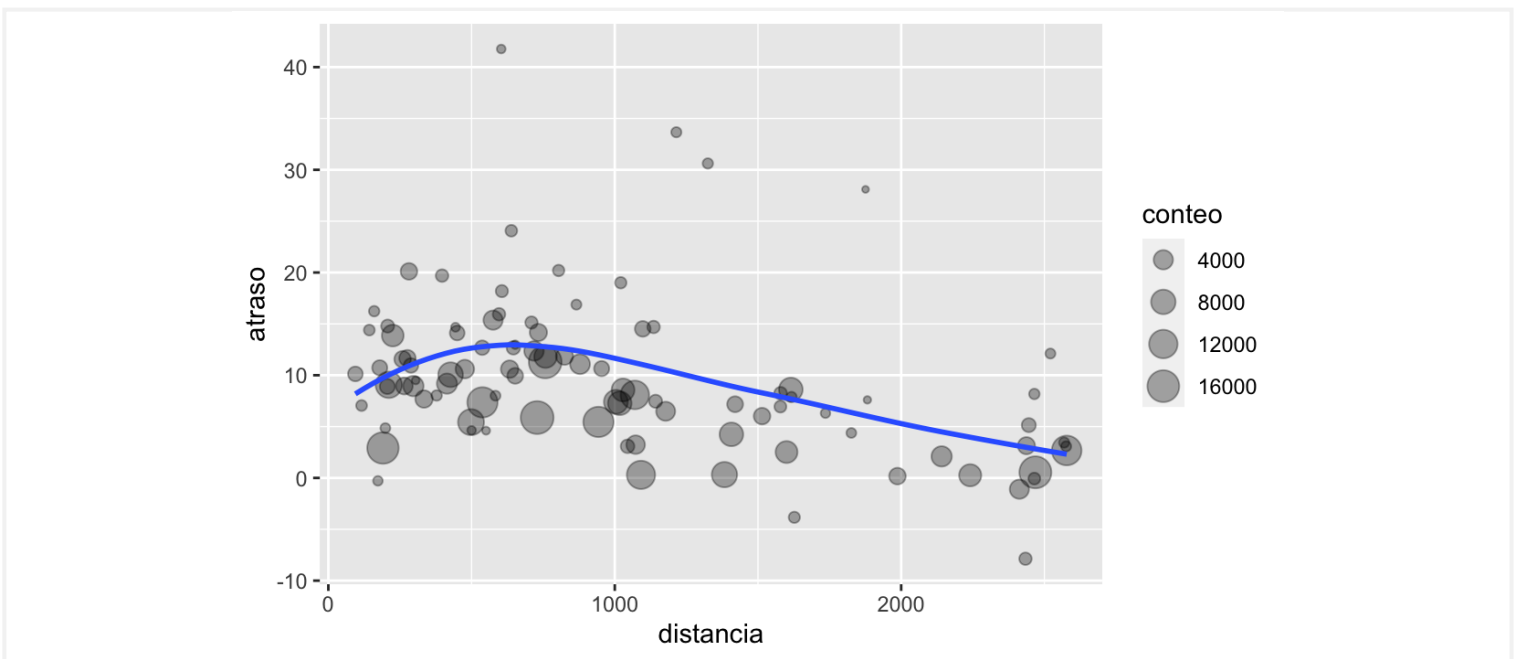
```
por_dia <- group_by(vuelos, anio, mes, dia)
summarise(por_dia, atraso = mean(atraso_salida, na.rm = TRUE))
#> `summarise()` has grouped output by 'anio', 'mes'. You can override using the `.groups`
argument.
#> # A tibble: 365 x 4
#> # Groups:   anio, mes [12]
#>    anio   mes   dia atraso
#>   <int> <int> <int>  <dbl>
#> 1  2013     1     1  11.5
#> 2  2013     1     2  13.9
#> 3  2013     1     3  11.0
#> 4  2013     1     4   8.95
#> 5  2013     1     5   5.73
#> 6  2013     1     6   7.15
#> # … 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:

```
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)
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



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")
```

Copy

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))
#> `summarise()` has grouped output by 'anio', 'mes'. You can override using the `.groups`
argument.
#> # A tibble: 365 x 4
#> # Groups:   anio, mes [12]
#>    anio   mes   dia  mean
#>   <int> <int> <int> <dbl>
#> 1  2013     1     1    NA
#> 2  2013     1     2    NA
#> 3  2013     1     3    NA
#> 4  2013     1     4    NA
#> 5  2013     1     5    NA
#> 6  2013     1     6    NA
#> # … with 359 more rows
```

Copy

¡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 valores faltantes antes del cálculo:

```
vuelos %>%
  group_by(anio, mes, dia) %>%
  summarise(mean = mean(atraso_salida, na.rm = TRUE))
#> `summarise()` has grouped output by 'anio', 'mes'. You can override using the `.groups`
argument.
#> # A tibble: 365 x 4
#> # Groups:   anio, mes [12]
#>    anio   mes   dia  mean
#>   <int> <int> <int> <dbl>
#> 1  2013     1     1 11.5
#> 2  2013     1     2 13.9
#> 3  2013     1     3 11.0
#> 4  2013     1     4  8.95
#> 5  2013     1     5  5.73
#> 6  2013     1     6  7.15
#> # … with 359 more rows
```

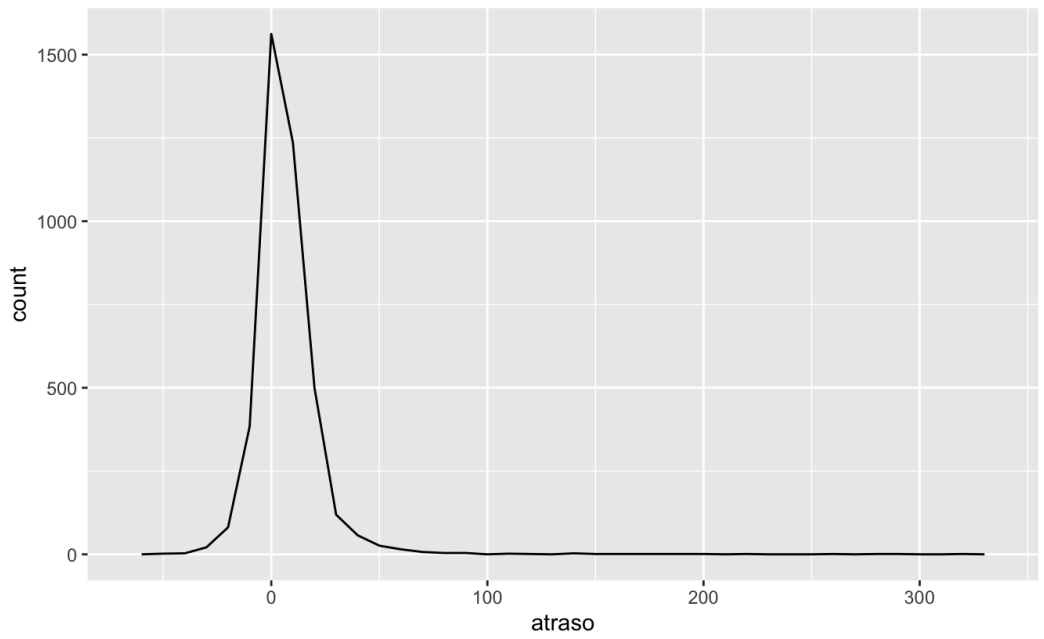
En este caso, en el que los valores faltantes representan vuelos cancelados, también podríamos abordar el problema eliminando primero este tipo de vuelos. Guardaremos este conjunto de datos para poder reutilizarlo en los siguientes ejemplos.

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

no_cancelados %>%
  group_by(anio, mes, dia) %>%
  summarise(mean = mean(atraso_salida))
#> `summarise()` has grouped output by 'anio', 'mes'. You can override using the `.groups`
argument.
#> # A tibble: 365 x 4
#> # Groups:   anio, mes [12]
#>    anio   mes   dia  mean
#>   <int> <int> <int> <dbl>
#> 1  2013     1     1 11.4
#> 2  2013     1     2 13.7
#> 3  2013     1     3 10.9
#> 4  2013     1     4  8.97
#> 5  2013     1     5  5.73
#> 6  2013     1     6  7.15
#> # … 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)
```

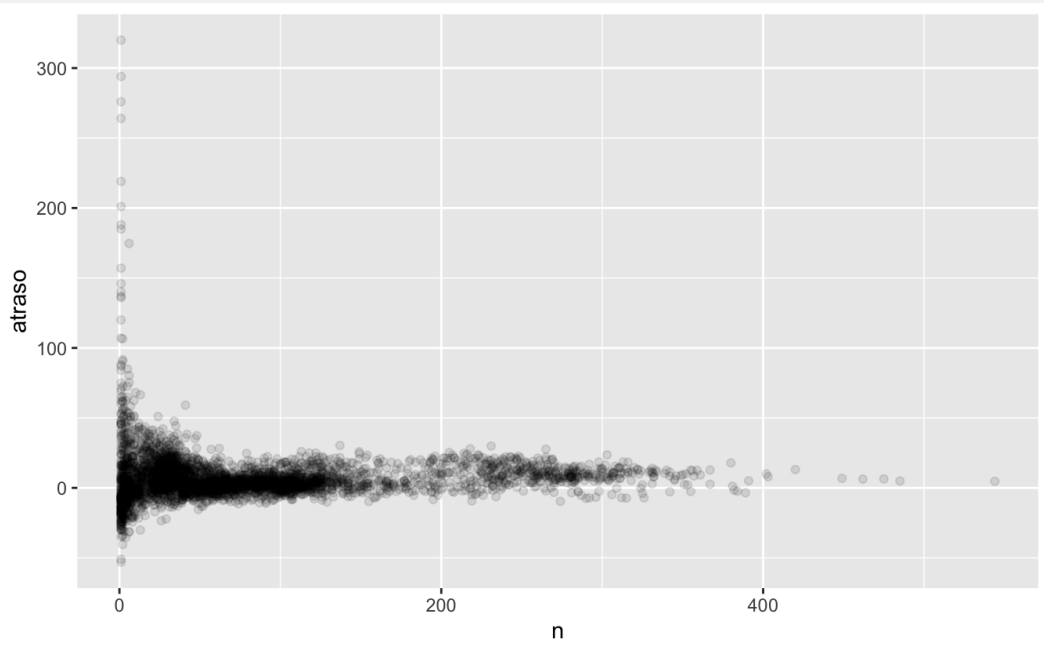


¡Hay algunos aviones que tienen una demora *promedio* de 5 horas (300 minutos)!

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)
```

Copy

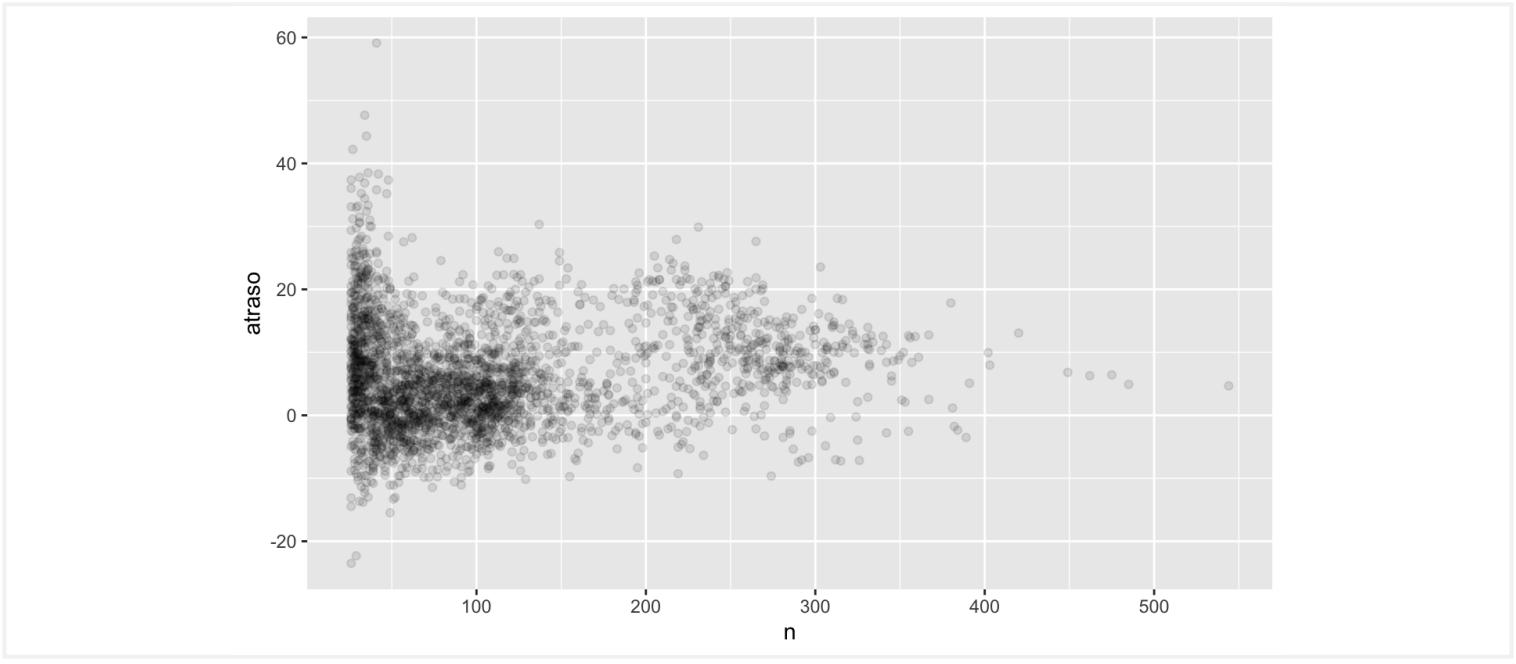


No es sorprendente que haya una mayor variación en el promedio de retraso cuando hay pocos vuelos. La forma de este gráfico es muy característica: cuando trazas un promedio (o cualquier otra medida de resumen) contra el tamaño del grupo, verás que la variación decrece a medida que el tamaño de muestra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

Copy



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 las Grandes Ligas.

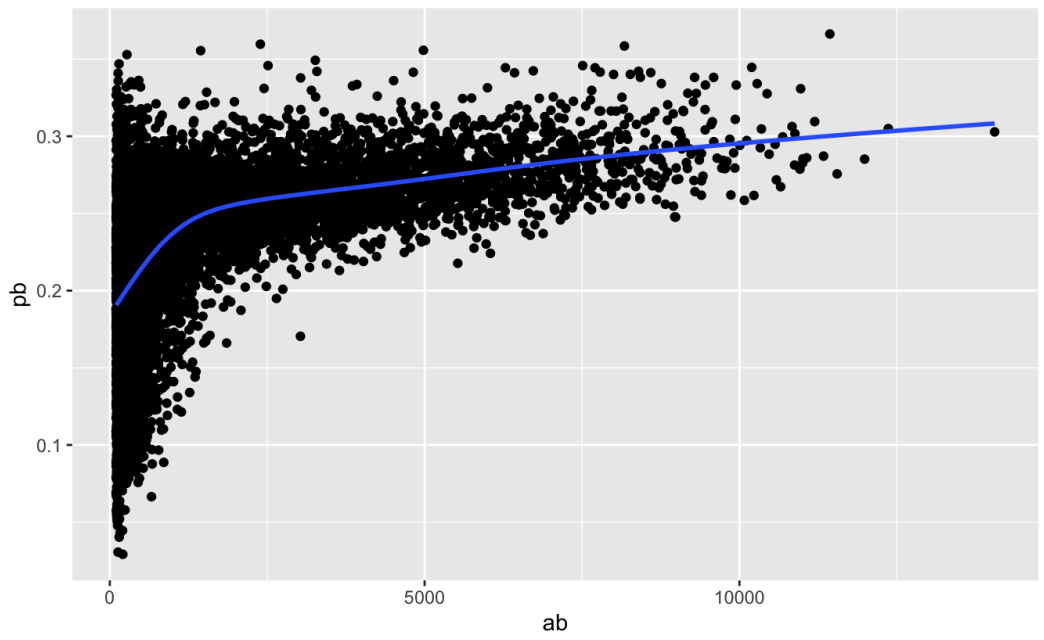
Cuando graficamos la habilidad del bateador (medido por el promedio de bateo, `pb`) contra el número de oportunidades para golpear la pelota (medido por el tiempo al bate, `ab`), verás dos patrones:

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)
```

Copy



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
#>   <chr>      <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 selección de subconjuntos.

```
no_cancelados %>%
  group_by(anio, mes, dia) %>%
  summarise(
    prom_atraso1 = mean(atraso_llegada),
    prom_atraso2 = mean(atraso_llegada[atraso_llegada > 0]) # el promedio de atrasos positivos
  )
#> `summarise()` has grouped output by 'anio', 'mes'. You can override using the `.groups`
argument.
#> # A tibble: 365 x 5
#> # Groups:   anio, mes [12]
#>    anio   mes   dia prom_atraso1 prom_atraso2
#>   <int> <int> <int>        <dbl>        <dbl>
#> 1  2013     1     1        12.7          32.5
#> 2  2013     1     2        12.7          32.0
#> 3  2013     1     3         5.73         27.7
#> 4  2013     1     4        -1.93         28.3
#> 5  2013     1     5        -1.53         22.6
#> 6  2013     1     6         4.24         24.4
#> # … 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)
  )
#> `summarise()` has grouped output by 'anio', 'mes'. You can override using the `.groups`
argument.
#> # A tibble: 365 x 5
#> # Groups:   anio, mes [12]
#>    anio   mes   dia primero ultimo
#>   <int> <int> <int>   <int>  <int>
#> 1  2013     1     1     517   2356
#> 2  2013     1     2      42   2354
#> 3  2013     1     3      32   2349
#> 4  2013     1     4      25   2358
#> 5  2013     1     5      14   2357
#> 6  2013     1     6      16   2355
#> # … 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)
  )
#> `summarise()` has grouped output by 'anio', 'mes'. You can override using the `.groups`
argument.
#> # A tibble: 365 x 5
#> # Groups:   anio, mes [12]
#>    anio   mes   dia primera_salida ultima_salida
#>   <int> <int> <int>          <int>         <int>
#> 1  2013     1     1            517          2356
#> 2  2013     1     2             42          2354
#> 3  2013     1     3             32          2349
#> 4  2013     1     4             25          2358
#> 5  2013     1     5             14          2357
#> 6  2013     1     6             16          2355
#> # … 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
```

Copy

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
```

Copy

- 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))
#> `summarise()` has grouped output by 'anio', 'mes'. You can override using the `.groups`
argument.
#> # A tibble: 365 x 4
#> # Groups:   anio, mes [12]
#>    anio   mes   dia n_temprano
#>   <int> <int> <int>      <int>
#> 1  2013     1     1          0
#> 2  2013     1     2          3
#> 3  2013     1     3          4
#> 4  2013     1     4          3
#> 5  2013     1     5          3
#> 6  2013     1     6          2
#> # … 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))
#> `summarise()` has grouped output by 'anio', 'mes'. You can override using the `.groups`
argument.
#> # A tibble: 365 x 4
#> # Groups:   anio, mes [12]
#>    anio   mes   dia hora_prop
#>   <int> <int> <int>     <dbl>
#> 1  2013     1     1    0.0722
#> 2  2013     1     2    0.0851
#> 3  2013     1     3    0.0567
#> 4  2013     1     4    0.0396
#> 5  2013     1     5    0.0349
#> 6  2013     1     6    0.0470
#> # … with 359 more rows
```

Copy

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()))
#> `summarise()` has grouped output by 'anio', 'mes'. You can override using the `.groups`
argument.
#> # A tibble: 365 x 4
#> # Groups:   anio, mes [12]
#>    anio   mes   dia vuelos
#>   <int> <int> <int>  <int>
#> 1  2013     1     1    842
#> 2  2013     1     2    943
#> 3  2013     1     3    914
#> 4  2013     1     4    915
#> 5  2013     1     5    720
#> 6  2013     1     6    832
#> # … with 359 more rows
(por_mes <- summarise(por_dia, vuelos = sum(vuelos)))
#> `summarise()` has grouped output by 'anio'. You can override using the `.groups` argument.
#> # A tibble: 12 x 3
#> # Groups:   anio [1]
#>    anio   mes vuelos
#>   <int> <int>  <int>
#> 1  2013     1  27004
#> 2  2013     2  24951
#> 3  2013     3  28834
#> 4  2013     4  28330
#> 5  2013     5  28796
#> 6  2013     6  28243
#> # … 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:

```
destinos_populares <- vuelos %>%
  group_by(destino) %>%
  filter(n() > 365)
destinos_populares
#> # A tibble: 332,577 x 19
#> # Groups:   destino [77]
#>    anio   mes   dia horario_salida salida_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:

```
destinos_populares %>%
  filter(atraso_llegada > 0) %>%
  mutate(prop_atraso = atraso_llegada / sum(atraso_llegada)) %>%
  select(anio:dia, destino, atraso_llegada, prop_atraso)
#> # A tibble: 131,106 x 6
#> # Groups:   destino [77]
#>    anio   mes   dia destino atraso_llegada prop_atraso
#>   <int> <int> <int> <chr>            <dbl>       <dbl>
#> 1  2013     1     1 IAH                 11   0.000111
#> 2  2013     1     1 IAH                 20   0.000201
#> 3  2013     1     1 MIA                 33   0.000235
#> 4  2013     1     1 ORD                 12   0.0000424
#> 5  2013     1     1 FLL                 19   0.0000938
#> 6  2013     1     1 ORD                  8   0.0000283
#> # … 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.

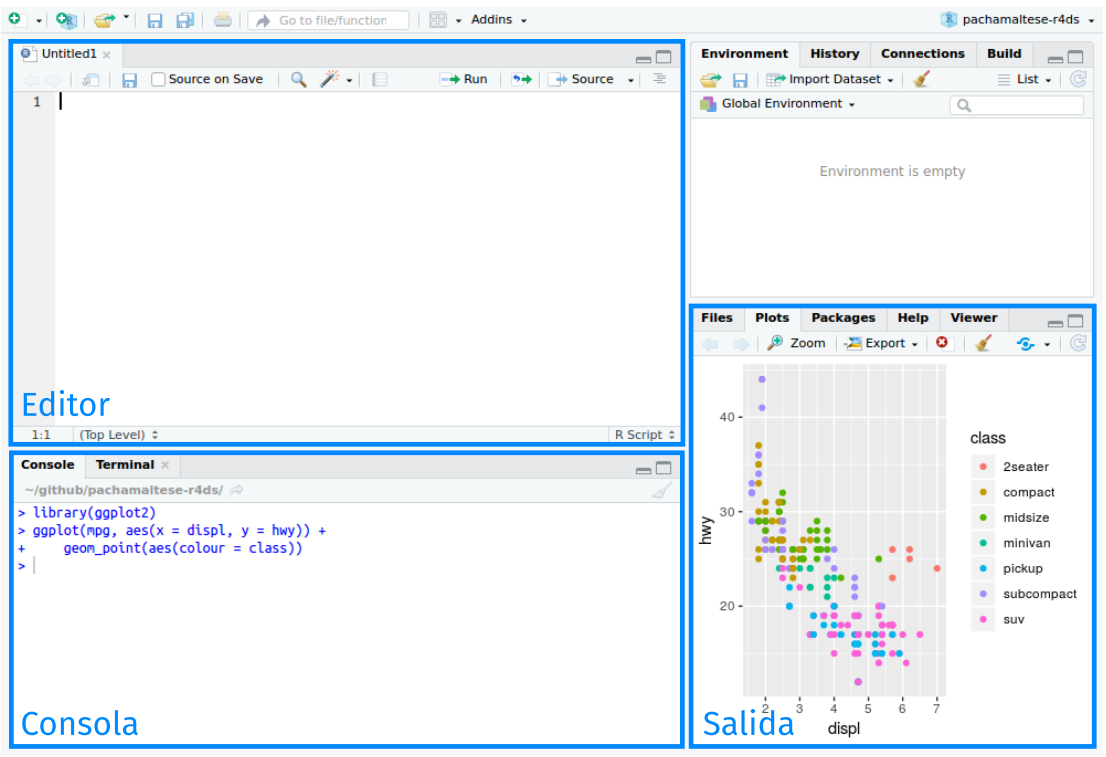
8. Para cada avión, cuenta el número de vuelos antes del primer retraso de más de 1 hora.

« 4 Flujo de trabajo: conocimientos básicos

6 Flujo de trabajo: Scripts »

6 Flujo de trabajo: *Scripts*

Hasta ahora estuviste utilizando la consola para ejecutar código. Ese es un buen punto de partida, pero verás que se vuelve incómodo rápidamente a medida que creas gráficos de **ggplot2** y *pipes* en **dplyr** más complejos. Para tener más espacio de trabajo, una buena idea es usar el editor de *script*. Ábrelo ya sea haciendo clic en el menú de Archivo (*File*), seleccionando Nuevo Archivo (*New File*), y luego Script de R (*R Script*), o bien, utilizando el atajo del teclado `Cmd/Ctrl + Shift + N`. Ahora verás cuatro paneles:



El editor de *script* es un excelente espacio para colocar el código que te importa. Continúa experimentando en la consola, pero una vez que has escrito un código que funciona y hace lo que quieres, colócalo en el editor de *script*. RsStudio guardará automáticamente los contenidos del editor cuando salgas del programa, y los cargará automáticamente cuando vuelvas a abrirlo. De todas formas, es una buena idea que guardes los *scripts* regularmente y que los respaldes.

6.1 Ejecutando código

El editor de *script* es también un excelente espacio para desarrollar gráficos de **ggplot2** complejos o largas secuencias de manipulación con **dplyr**. La clave para utilizar el editor de *script* efectivamente es memorizar uno de los atajos del teclado más importantes: Cmd/Ctrl + Enter. Esto ejecuta la expresión actual de R en la consola. Por ejemplo, toma el código de abajo. Si tu cursor está sobre `no_cancelado <- vuelos %>%`, presionar `Cmd/Ctrl + Enter` ejecutará el comando completo que genera `no_cancelado`. También moverá el cursor al siguiente enunciado (que comienza con `no_cancelado %>%`). Esto facilita ejecutar todo tu *script* presionando repetidamente Cmd/Ctrl + Enter.

```
library(dplyr)
library(datos)
no_cancelado <- vuelos %>%
  filter(!is.na(atraso_salida), !is.na(atraso_llegada))
no_cancelado %>%
  group_by(anio, mes, dia) %>%
  summarise(media = mean(atraso_salida))
```

Copy

En lugar de correr expresión por expresión, también puedes ejecutar el *script* completo en un paso: `Cmd/Ctrl + Shift + S`. Hacer esto regularmente es una buena forma de verificar que has capturado todas las partes importantes de tu código en el *script*. Te recomendamos que siempre comiences tu *script* cargando los paquetes que necesitas. De este modo, si compartes tu código con otras personas, quienes lo utilicen pueden fácilmente ver qué paquetes necesitan instalar. Ten en cuenta, sin embargo, que nunca deberías incluir `install.packages()` (del inglés, *instalar paquetes*) o `setwd()` (del inglés *set working directory*, establecer directorio de trabajo) en un *script* que compartes. ¡Es muy antisocial cambiar la configuración en la computadora de otra persona!

On this page

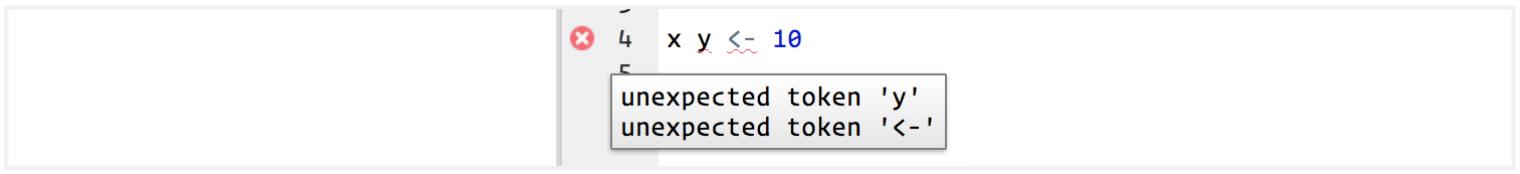
View source
Edit this page

Cuando trabajes en los capítulos que siguen, te sugerimos comenzar en el editor y practicar los atajos de tu teclado. Con el tiempo, enviar código a la consola de esta forma se volverá tan natural que ni siquiera tendrás que pensarlo.

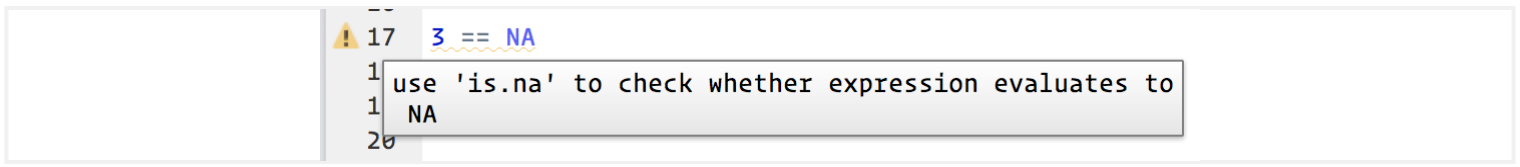
6.2 Diagnósticos de RStudio

El editor de *script* resaltará errores de sintaxis con una línea roja serpenteante bajo el código y una cruz en la barra lateral:

Sitúate sobre la cruz para ver cuál es el problema:



RStudio te informará también sobre posibles problemas:



6.3 Ejercicios

1. Visita la cuenta de Twitter de RStudio Tips, https://twitter.com/rstudiotips, y encuentra algún *tip* que parezca interesante. Practica utilizándolo.

2. ¿Qué otros errores comunes reportarán los diagnósticos de RStudio? Lee https://support.rstudio.com/hc/en-us/articles/205753617-Code-Diagnostics para descubrirlo.

« 5 Transformación de datos

7 Análisis exploratorio de datos (EDA) »

7 Análisis exploratorio de datos (*EDA*)

7.1 Introducción

Este capítulo te mostrará cómo usar la visualización y la transformación para explorar tus datos de manera sistemática, una tarea que las personas de Estadística suelen llamar análisis exploratorio de datos, o *EDA* (por sus siglas en inglés ***e**xploratory **d**ata **a**nalysis*). El *EDA* es un ciclo iterativo en el que:

1. Generas preguntas acerca de tus datos.

2. Buscas respuestas visualizando, transformando y modelando tus datos.

3. Usas lo que has aprendido para refinar tus preguntas y/o generar nuevas interrogantes.

El análisis exploratorio de datos no es un proceso formal regido por un conjunto estricto de reglas. El *EDA* es, más que nada, un estado mental. Durante las fases iniciales del *EDA* deberías ser libre de investigar todas las ideas que se te ocurran. Algunas de estas ideas prosperarán, mientras que otras serán como callejones sin salida. A medida que tu exploración continúa, te concentrarás en ciertas áreas particularmente productivas sobre las que eventualmente escribirás y comunicarás a otras personas.

El *EDA* es una parte importante de cualquier análisis, aun si las preguntas están servidas en bandeja, pues siempre tendrás que examinar la calidad de tus datos. La limpieza de datos es una aplicación del *EDA*: haces preguntas acerca de si tus datos cumplen con tus expectativas o no. Para limpiar tus datos tendrás que desplegar todas las herramientas del *EDA*: visualización, transformación y modelado.

7.1.1 Requisitos indispensables

En este capítulo combinaremos lo que aprendiste sobre *dplyr* y *ggplot2* para formular preguntas de manera interactiva, encontrar las respuestas en los datos, para luego plantear nuevas preguntas.

```
library(tidyverse)
library(datos)
```

Copy

7.2 Preguntas

> “No hay tal cosa como preguntas estadísticas rutinarias, solo rutinas estadísticas cuestionables.” — Sir David Cox

> “Es preferible una respuesta aproximada a la pregunta correcta, que frecuentemente es formulada de manera imprecisa, que una respuesta exacta a la pregunta incorrecta, que siempre puede ser formulada de manera precisa.” — John Tukey

Tu objetivo durante el *EDA* es desarrollar un entendimiento de tus datos. La manera más fácil de lograrlo es usar preguntas como herramientas para guiar tu investigación. Cuando formulas una pregunta, esta orienta tu atención en una parte específica de tu conjunto de datos y te ayuda a decidir qué gráficos, modelos o transformaciones son necesarios.

El *EDA* es un proceso fundamentalmente creativo. Y como la mayoría de los procesos creativos, la clave para formular preguntas *de calidad* es generar una gran *cantidad* de preguntas. Es difícil hacer preguntas reveladoras al inicio de tu análisis pues aún no sabes qué percepciones o conocimientos están contenidos en tu conjunto de datos. Por otro lado, cada nueva pregunta que te plantees revelará un nuevo aspecto de tus datos y las probabilidades de hacer un descubrimiento serán mayores. Puedes profundizar en las partes más interesantes de tus datos—y desarrollar un conjunto de preguntas que estimularán tu pensamiento—si a cada pregunta le das seguimiento con un nuevo interrogante basado en tus descubrimientos.

No hay reglas específicas sobre qué preguntas deberías hacerte para guiar tu investigación. Sin embargo, hay dos tipos de preguntas que siempre te serán útiles para hacer descubrimientos dentro tus datos. Puedes formular dichas preguntas de la siguiente manera:

1. ¿Qué tipo de variación existe dentro de cada una de mis variables?

2. ¿Qué tipo de covariación ocurre entre mis diferentes variables?

El resto de este capítulo examinará las preguntas anteriores. Explicaremos qué es variación y covariación, y te mostraremos diferentes maneras de responder cada pregunta. Para facilitar esta discusión, definamos algunos términos:

- Una **variable** es una cantidad, cualidad o característica mesurable, es decir, que se puede medir.

- Un **valor** es el estado de la variable en el momento en que fue medida. El valor de una variable puede cambiar de una medición a otra.

- Una **observación** es un conjunto de mediciones realizadas en condiciones similares (usualmente todas las mediciones de una observación son realizadas al mismo tiempo y sobre el mismo objeto). Una observación contiene muchos valores, cada uno asociado a una variable diferente. En algunas ocasiones nos referiremos
  a una observación como un punto específico (*data point* en inglés).

- Los **datos tabulares** (*tabular data* en inglés) son un conjunto de valores, cada uno asociado a una variable y a una observación. Los datos tabulares están ordenados si cada valor está almacenado en su propia "celda", cada variable cuenta con su propia columna, y cada observación corresponde a una fila.

Todos los datos que has visto hasta ahora han estado ordenados. En la vida real, la mayor parte de los datos no lo están, así que retomaremos estas ideas de nuevo en datos ordenados.

7.3 Variación

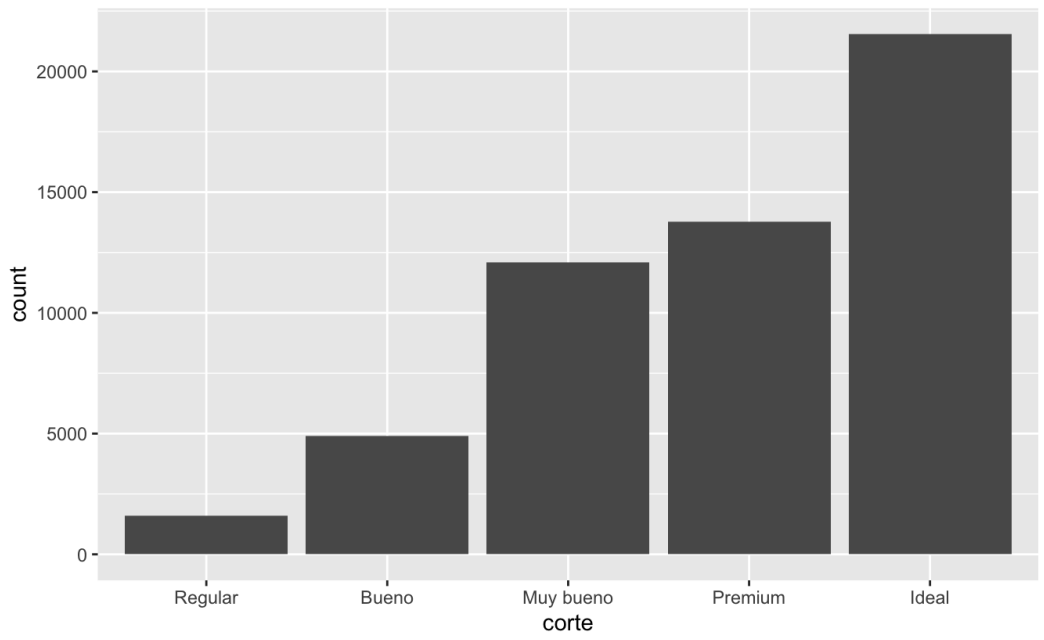
La **variación** es la tendencia de los valores de una variable a cambiar de una medición a otra. Es fácil observar dicha variación en la vida real; si mides cualquier variable continua dos veces obtendrás diferentes resultados. Esto es verdad aun si mides cantidades que son constantes, como la velocidad de la luz. Cada una de tus mediciones incluirá una cantidad pequeña de error que variará de medición a medición. Las variables categóricas también pueden variar si realizas mediciones con diferentes sujetos (p. ej., el color de ojos de diferentes personas) o en momentos diferentes (p.ej., los niveles de energía que un electrón posee en momentos distintos). La variación de cada variable sigue un patrón específico y este puede revelar información interesante. La mejor manera de entender dicho patrón es visualizando la distribución de los valores de la variable.

7.3.1 Visualizando distribuciones

Cómo visualizar la distribución de una variable dependerá de si la variable es categórica o continua. Una variable es **categórica** si únicamente puede adoptar un valor correspondiente a un grupo limitado de valores. En R las variables categóricas usualmente son guardadas como vectores de factores o de caracteres. Para examinar la distribución de una variable categórica, usa un gráfico de barras:

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

Copy

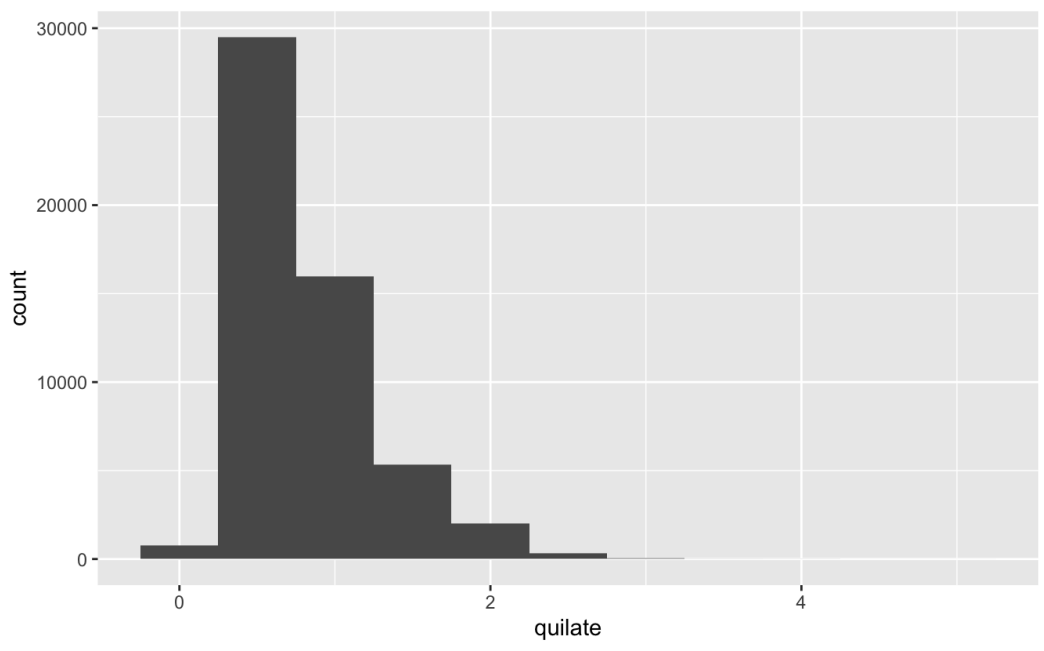


La altura de las barras muestra cuántas observaciones corresponden a cada valor de x. Puedes calcular estos valores con `dplyr::count()`:

```
diamantes %>%
  count(corte)
#> # A tibble: 5 x 2
#>   corte         n
#> * <ord>     <int>
#> 1 Regular    1610
#> 2 Bueno      4906
#> 3 Muy bueno 12082
#> 4 Premium   13791
#> 5 Ideal     21551
```

Una variable es **continua** si puede adoptar un set infinito de valores ordenados. Los números y fechas-horas son dos ejemplos de variables continuas. Para examinar la distribución de una variable continua, usa un histograma:

```
ggplot(data = diamantes) +
  geom_histogram(mapping = aes(x = quilate), binwidth = 0.5)
```



Puedes calcular esto manualmente combinando `dplyr::count()` y `ggplot2::cut_width()`:

```
diamantes %>%
  count(cut_width(quilate, 0.5))
#> # A tibble: 11 x 2
#>   `cut_width(quilate, 0.5)`     n
#> * <fct>                     <int>
#> 1 [-0.25,0.25]                785
#> 2 (0.25,0.75]               29498
#> 3 (0.75,1.25]               15977
#> 4 (1.25,1.75]                5313
#> 5 (1.75,2.25]                2002
#> 6 (2.25,2.75]                 322
#> # … with 5 more rows
```

On this page

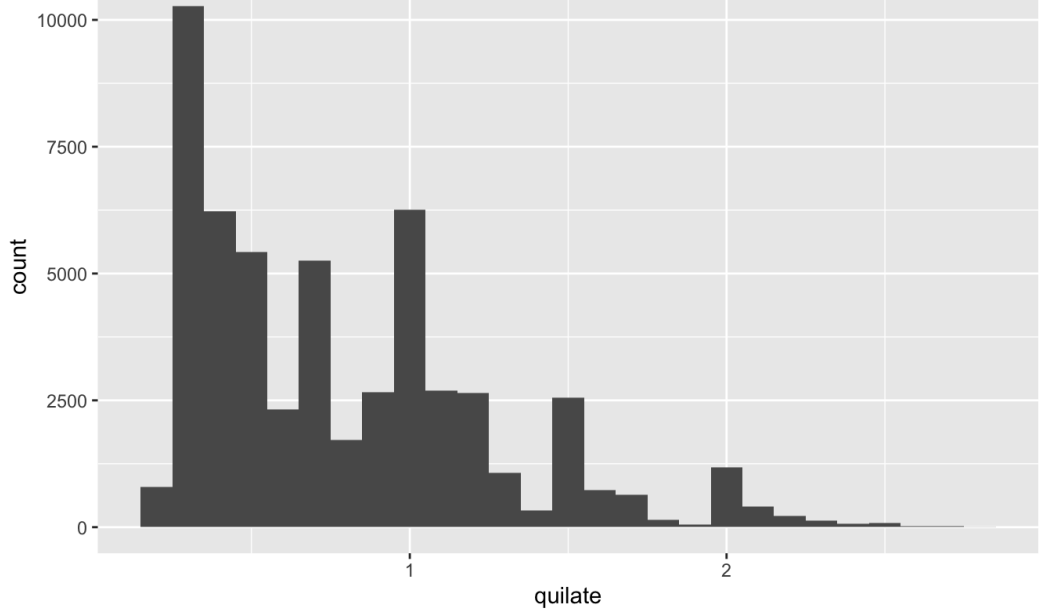
View source
Edit this page

Un histograma segmenta el eje horizontal en rangos equidistantes y después hace uso de la altura de la barra para mostrar el número de observaciones que corresponden a cada unidad o barra. En el gráfico anterior, la barra más alta muestra que al menos 30,000 observaciones tienen un valor de `quilate` entre 0.25 y 0.75, que son los bordes izquierdo y derecho de la barra.

Puedes definir el ancho de los intervalos de un histograma con el argumento `binwidth` (*ancho del contenedor* en inglés), que es medido en las unidades de la variable `x`. Siempre deberías explorar una variedad de distintas medidas para el ancho del intervalo cuando estés trabajando con histogramas, pues distintas medidas pueden revelar diferentes patrones. Por ejemplo, así es como luce la gráfica anterior cuando acercamos la imagen a solo los diamantes con un tamaño menor a tres quilates y escogemos un intervalo más pequeño.

```
pequenos <- diamantes %>%
  filter(quilate < 3)

ggplot(data = pequenos, mapping = aes(x = quilate)) +
  geom_histogram(binwidth = 0.1)
```

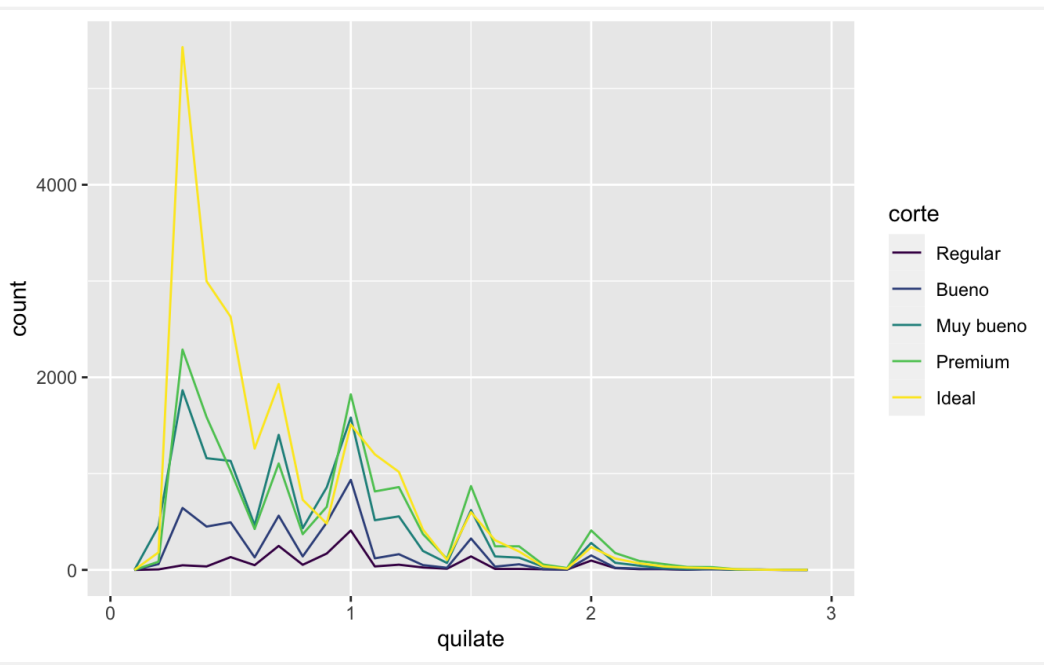
Copy



Si deseas sobreponer múltiples histogramas en la misma gráfica, te recomendamos usar `geom_freqpoly()` (*polígonos de frecuencia*) en lugar de `geom_histogram()`. `geom_freqpoly()` realiza el mismo cálculo que `geom_histogram()`, pero usa líneas en lugar de barras para mostrar los totales. Es mucho más fácil entender líneas que barras que se sobreponen.

```
ggplot(data = pequenos, mapping = aes(x = quilate, colour = corte)) +
  geom_freqpoly(binwidth = 0.1)
```

Copy



Hay ciertas limitantes con este tipo de gráfico, pero volveremos a ellas en la sección sobre visualizar una variable categórica y una variable continua.

Ahora que puedes visualizar la variación en los datos, ¿qué deberías buscar en tus gráficos? ¿Y qué tipo de preguntas de seguimiento deberías hacerte? Hemos generado una lista de los tipos más útiles de información que puedes encontrar en tus gráficos, además de algunas preguntas de seguimiento para cada tipo de información. La clave para hacer buenas preguntas será confiar en tu curiosidad (¿sobre qué quieres aprender más?), así como en tu escepticismo (¿de qué manera podría ser esto engañoso?).

7.3.2 Valores típicos

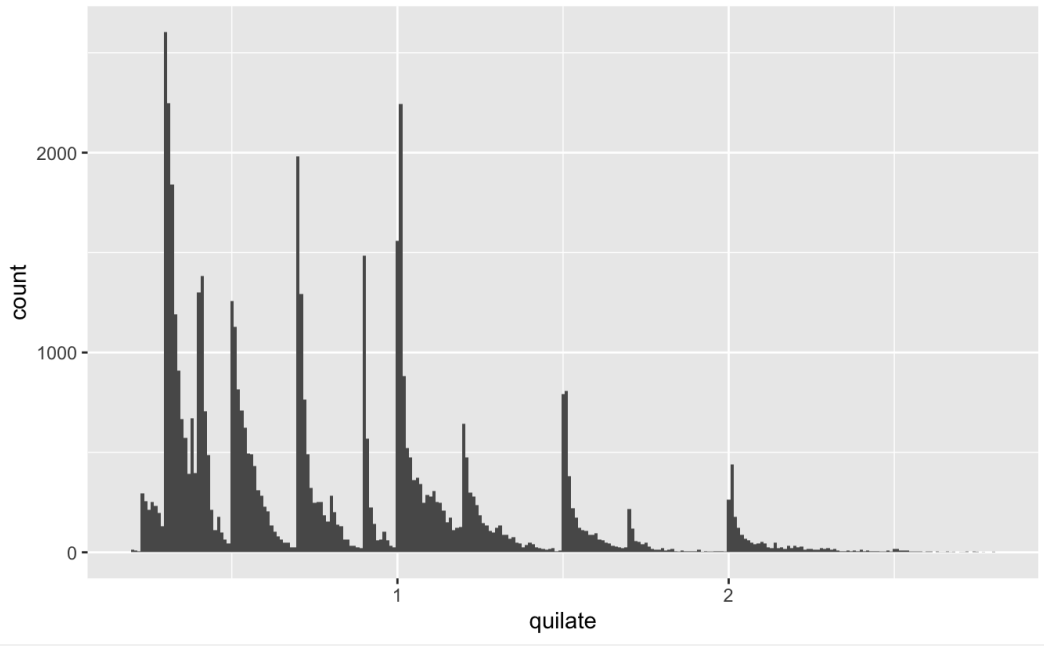
Tanto en gráficos de barra como en histogramas, las barras altas muestran los valores más comunes de una variable y las barras más cortas muestran valores menos comunes. Espacios que no tienen barras revelan valores que no fueron observados en tus datos. Para convertir esta información en preguntas útiles, busca cosas que sean inesperadas:

- ¿Qué valores son los más comunes? ¿Por qué?
- ¿Qué valores son infrecuentes? ¿Por qué? ¿Cumple esto tus expectativas?
- ¿Puedes ver patrones inusuales? ¿Qué podría explicarlos?

Por ejemplo, el siguiente histograma sugiere varias preguntas interesantes:

- ¿Por qué hay más diamantes en quilates completos y fracciones comunes de quilates?
- ¿Por qué hay más diamantes hacia la derecha de cada sección que hacia la izquierda?
- ¿Por qué no hay diamantes más grandes que 3 quilates?

```
ggplot(data = pequenos, mapping = aes(x = quilate)) +
  geom_histogram(binwidth = 0.01)
```

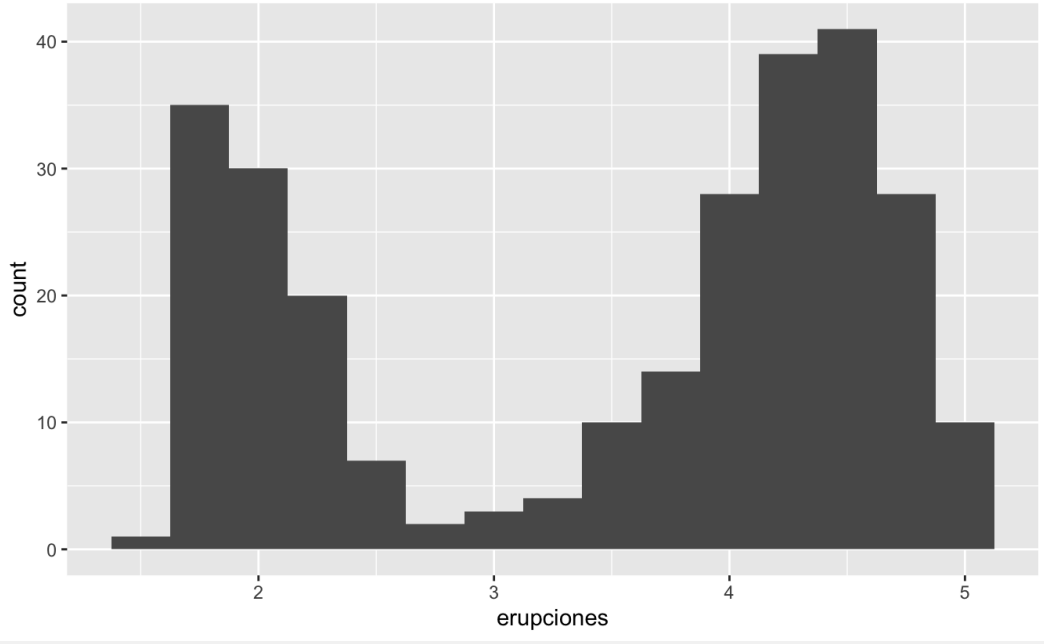


Las agrupaciones de valores similares sugieren que ciertos subgrupos existen en tus datos. Para entender un poco mejor los subgrupos, pregúntate lo siguiente:

- ¿De qué manera son similares entre sí las observaciones dentro de cada grupo?
- ¿De qué manera son diferentes las observaciones que corresponden a grupos diferentes?
- ¿Cómo puedes explicar o definir los grupos?
- ¿Por qué puede ser engañosa la apariencia de los grupos?

El siguiente histograma muestra la duración (en minutos) de 272 erupciones del géiser Viejo Fiel (Old Faithful) en el Parque Nacional Yellowstone. La duración de las erupciones parece estar agrupada en dos conjuntos: erupciones cortas (con duración de alrededor de dos minutos) y erupciones largas (4-5 minutos), y pocos datos en el intervalo intermedio.

```
ggplot(data = fiel, mapping = aes(x = erupciones)) +
  geom_histogram(binwidth = 0.25)
```



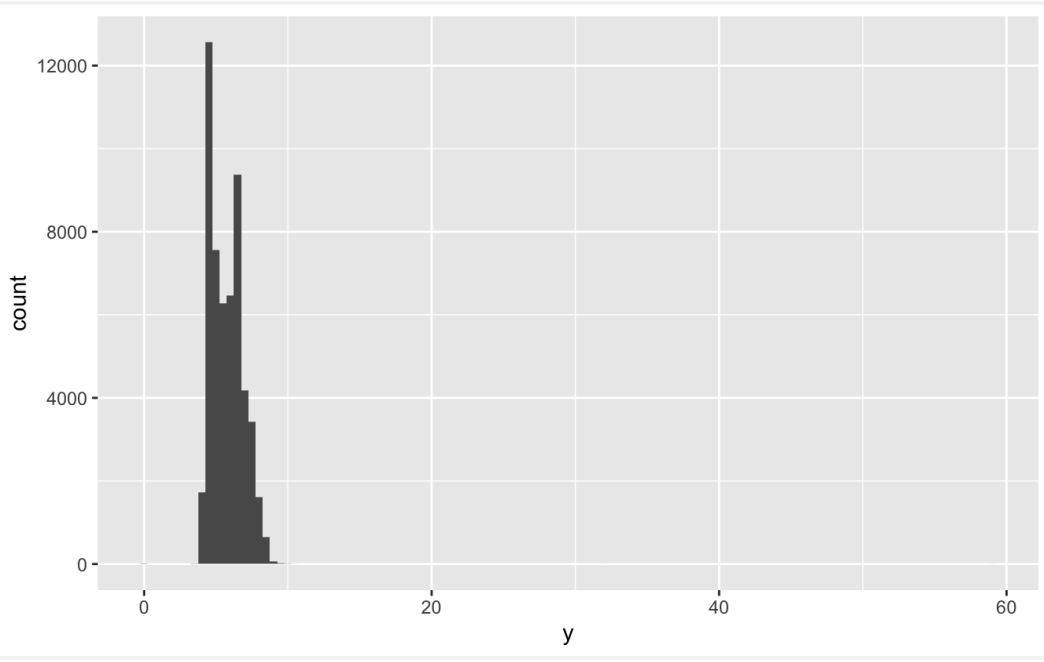
Muchas de las preguntas anteriores te ayudarán a explorar la relación *entre* variables, por ejemplo, para ver si los valores de una variable pueden explicar el comportamiento de otra variable. Pronto llegaremos a ese punto.

7.3.3 **Valores inusuales**

Los valores atípicos, conocidos en inglés como *outliers*, son puntos en los datos que parecen no ajustarse al patrón. Algunas veces dichos valores atípicos son errores cometidos durante la ingesta de datos; otras veces sugieren nueva información. Cuando tienes una gran cantidad de datos, es difícil identificar los valores atípicos en un histograma. Por ejemplo, toma la distribución de la variable `y` del set de datos de diamantes. La única evidencia de la existencia de valores atípicos son límites inusualmente anchos en el eje horizontal.

```
ggplot(diamantes) +
  geom_histogram(mapping = aes(x = y), binwidth = 0.5)
```

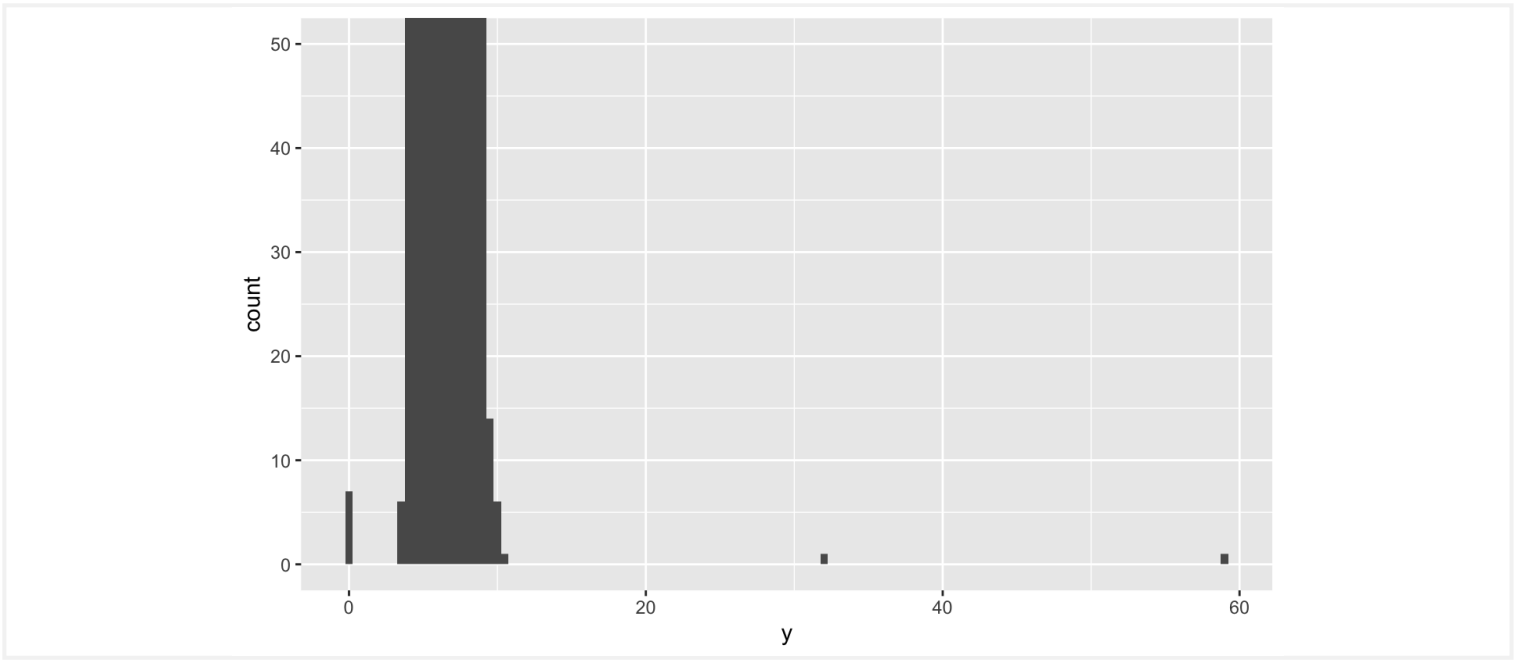
Copy



Hay tantas observaciones en las barras comunes que las barras infrecuentes son tan cortas que no es posible verlas a simple vista (aunque quizá si observas con mucha atención el 0 podrías encontrar algo). Para facilitar la tarea de visualizar valores inusuales, necesitamos acercar la imagen a los valores más pequeños del eje vertical con `coord_cartesian()`:

```
ggplot(diamantes) +
  geom_histogram(mapping = aes(x = y), binwidth = 0.5) +
  coord_cartesian(ylim = c(0, 50))
```

Copy



(`coord_cartesian()` también tiene un argumento `xlim()` para cuando es necesario acercar la imagen sobre el eje horizontal. ggplot2 también tiene funciones `xlim()` e `ylim()` que funcionan de una manera distinta: ignoran los datos que se encuentran fuera de los límites.)

Esto nos permite ver que hay tres valores inusuales: 0, ~30, y ~60. Podemos removerlos con dplyr:

```
inusual <- diamantes %>%
  filter(y < 3 | y > 20) %>%
  select(precio, x, y, z) %>%
  arrange(y)
inusual
#> # A tibble: 9 x 4
#>   precio     x     y     z
#>    <int> <dbl> <dbl> <dbl>
#> 1   5139  0      0    0
#> 2   6381  0      0    0
#> 3  12800  0      0    0
#> 4  15686  0      0    0
#> 5  18034  0      0    0
#> 6   2130  0      0    0
#> 7   2130  0      0    0
#> 8   2075  5.15  31.8  5.12
#> 9  12210  8.09  58.9  8.06
```

Copy

La variable `y` mide una de las tres dimensiones de estos diamantes en mm (milímetros). Sabemos que los diamantes no pueden tener una anchura de 0mm, así que estos valores deben ser incorrectos. También podríamos sospechar que las medidas de 32mm y 59mm son improbables: esos diamantes medirían más de una pulgada de largo, ¡pero no cuestan cientos de miles de dólares!

Es un buen hábito repetir tu análisis con y sin los valores inusuales. Si tienen un efecto mínimo en los resultados y no puedes descubrir por qué están en los datos, es razonable reemplazarlos con valores ausentes y seguir adelante con tu análisis. Sin embargo, si tienen un efecto sustancial en tus resultados, no deberías ignorarlos sin justificación. Deberás descubrir qué los causó (p.ej., un error en la entrada de datos) y explicitar que los removiste en tu reporte escrito.

7.3.4 **Ejercicios**

1. Explora la distribución de cada una de las variables `x`, `y`, y `z` en el set de datos `diamantes`. ¿Qué aprendiste? Piensa en un diamante y cómo decidirías qué dimensiones corresponden a la longitud, ancho y profundidad.

2. Explora la distribución de `precio`. ¿Ves algo inusual o sorprendente? (Sugerencia: Piensa detenidamente en `binwidth` y asegúrate de usar un rango largo de valores.)

3. ¿Cuántos diamantes tienen 0.99 quilates? ¿Cuántos son de 1 quilate? ¿Qué piensas que puede ser la causa de dicha diferencia?

4. Compara y contrasta `coord_cartesian()` contra `xlim()` o `ylim()` en cuanto a acercar la imagen en un histograma. ¿Qué pasa si no modificas el valor de `binwidth`? ¿Qué pasa si intentas acercar la imagen de manera que solo aparezca la mitad de una barra?

7.4 Valores faltantes

Si has encontrado valores inusuales en tu set de datos y simplemente quieres seguir con el resto de tu análisis, tienes dos opciones.

1. Desecha la fila completa donde están los valores inusuales:

```
diamantes2 <- diamantes %>%
  filter(between(y, 3, 20))
```

   No recomiendo esta opción porque el hecho de que una medida sea inválida no significa que todas las mediciones lo sean. Además, si la calidad de tus datos es baja, después de aplicar esta estrategia en todas tus variables ¡quizás te des cuenta que ya no tienes datos con los que trabajar!

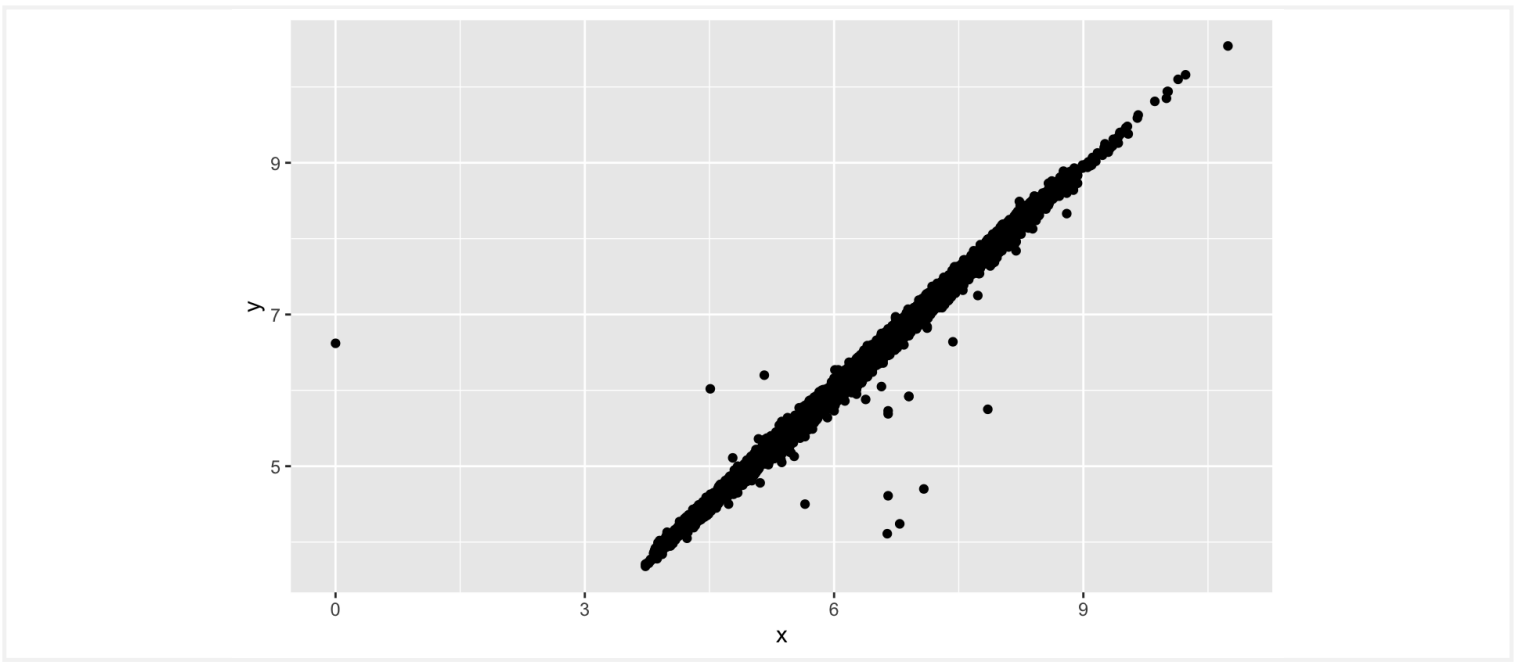
2. En lugar de eso, recomiendo reemplazar los valores inusuales con valores faltantes. La manera más fácil de hacerlo es usar `mutate()` para reemplazar la variable con una copia editada de la misma. Puedes usar la función `ifelse()` para reemplazar valores inusuales con `NA`:

```
diamantes2 <- diamantes %>%
  mutate(y = ifelse(y < 3 | y > 20, NA, y))
```

`ifelse()` tiene tres argumentos. El primer argumento `test` debe ser un vector lógico. El resultado contendrá el valor del segundo argumento, `sí`, cuando `test` sea `VERDADERO`, y el valor del tercer argumento, `no`, cuando sea falso. Como alternativa a **ifelse**, usa `dplyr::case_when()` (*caso cuando*). `case_when()` es particularmente útil dentro de **mutate** cuando quieras crear una nueva variable que dependa de una combinación compleja de variables existentes.

Como R, ggplot2 suscribe la filosofía de que los valores faltantes nunca deberían desaparecer silenciosamente. La cuestión de dónde graficar los valores faltantes no es trivial, así que ggplot2 no los incluye en los gráficos, pero emite una advertencia acerca de que fueron removidos:

```
ggplot(data = diamantes2, mapping = aes(x = x, y = y)) +
  geom_point()
#> Warning: Removed 9 rows containing missing values (geom_point).
```

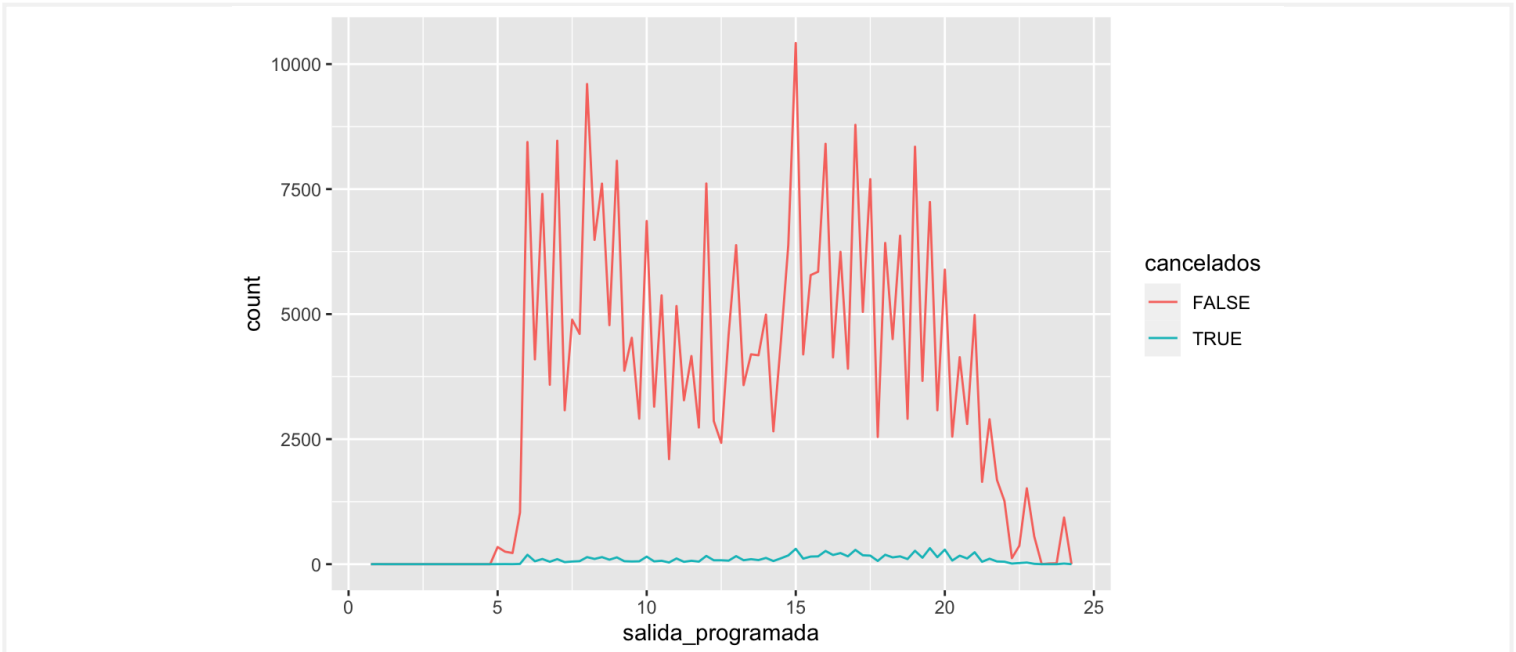


Para eliminar esa alerta, define `na.rm = TRUE`:

```
ggplot(data = diamantes2, mapping = aes(x = x, y = y)) +
  geom_point(na.rm = TRUE)
```

En otras ocasiones querrás entender qué diferencias hay entre las observaciones con valores faltantes y las observaciones con valores registrados. Por ejemplo, en `datos::vuelos` los valores faltantes en la variable `horario_salida` indicaban que el vuelo había sido cancelado, por lo que uno querría comparar el horario de salida programado para los vuelos cancelados y los no cancelados. Puedes hacer esto creando una nueva variable con `is.na()`.

```r
datos::vuelos %>%
  mutate(
    cancelados = is.na(horario_salida),
    hora_programada = salida_programada %/% 100,
    minuto_programado = salida_programada %% 100,
    salida_programada = hora_programada + minuto_programado / 60
  ) %>%
  ggplot(mapping = aes(salida_programada)) +
    geom_freqpoly(mapping = aes(colour = cancelados), binwidth = 1/4)
```



Sin embargo, este gráfico no es tan bueno porque hay muchos más vuelos no cancelados que vuelos cancelados. En la siguiente sección exploraremos algunas de las técnicas para mejorar esta comparación.

7.4.1 **Ejercicios**

1. ¿Qué sucede con los valores faltantes en un histograma? ¿Qué pasa con los valores faltantes en una gráfica de barras? ¿Cuál es la razón detrás de esta diferencia?

2. ¿Qué efecto tiene usar `na.rm = TRUE` en `mean()` (*media*) y `sum()` (*suma*)?

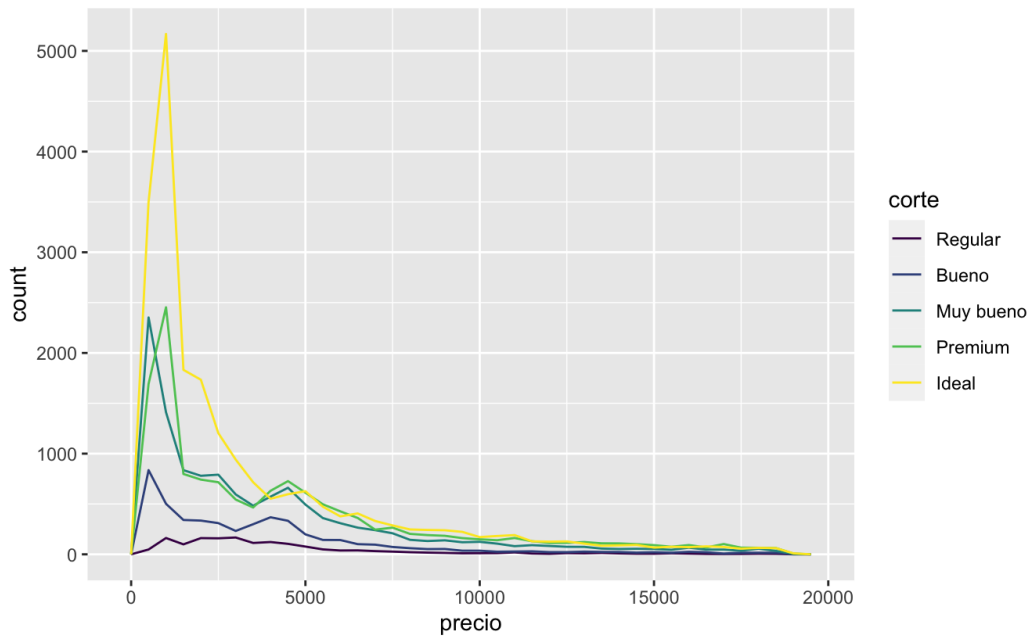
7.5 **Covariación**

Si la variación describe el comportamiento *dentro* de una variable, la covariación describe el comportamiento *entre* variables. La **covariación** es la tendencia de los valores de dos o más variables a variar simultáneamente de una manera relacionada La mejor manera de reconocer que existe covariación en tus datos es visualizar la relación entre dos o más variables. Cómo hacerlo dependerá de los tipos de variables involucradas.

7.5.1 **Una variable categórica y una variable continua**

Es común querer explorar la distribución de una variable continua agrupada por una variable categórica, como lo hicimos en el polígono de frecuencia anterior. La apariencia automática de `geom_freqpoly()` no es tan útil para este tipo de comparación, pues la altura está dada por la cuenta total. Eso significa que si uno de los grupos es mucho más pequeño que los demás, será difícil ver las diferencias en altura. Por ejemplo, exploremos cómo varía el precio dependiendo de la calidad de un diamante:

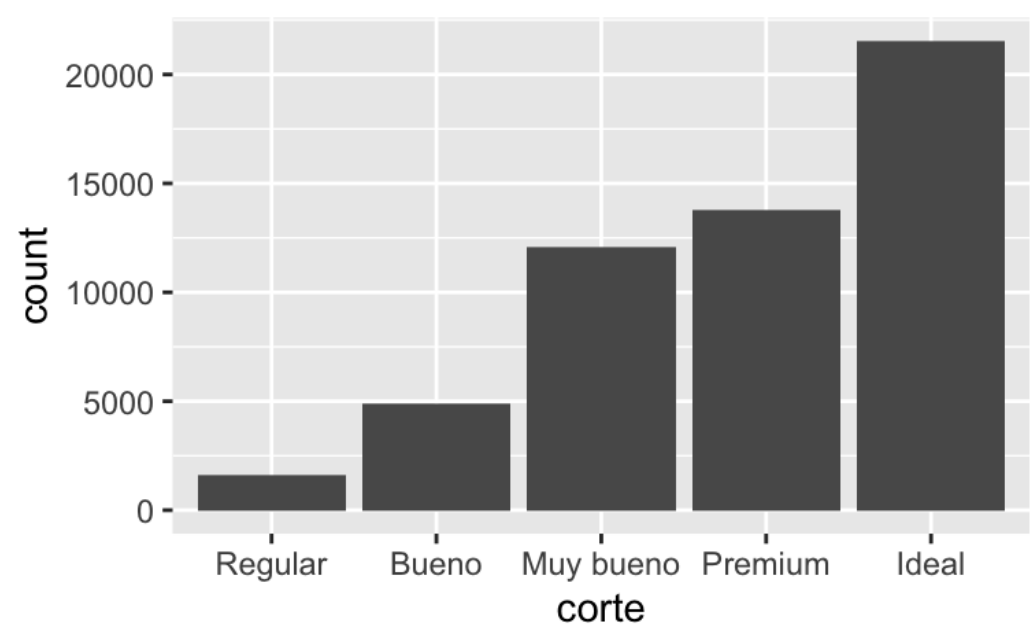
```r
ggplot(data = diamantes, mapping = aes(x = precio)) +
  geom_freqpoly(mapping = aes(colour = corte), binwidth = 500)
```



Resulta difícil observar las diferencias de las distribuciones porque las cuentas totales difieren en gran medida:

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

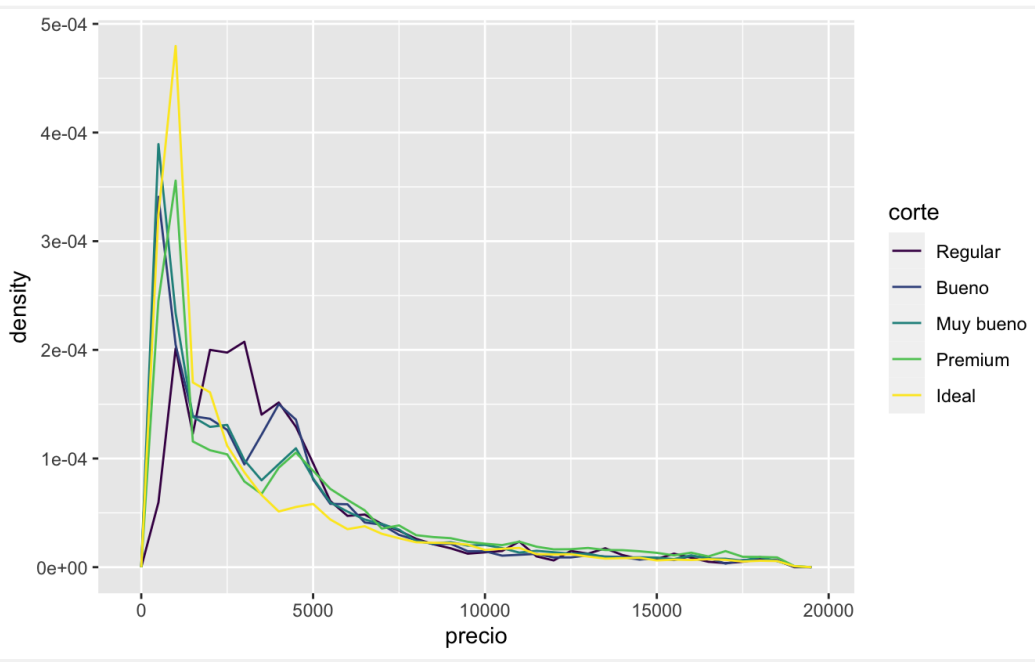
Copy



Para facilitar esta comparación necesitamos cambiar lo que se muestra en el eje vertical. En lugar de mostrar la cuenta total, mostraremos la **densidad**, que es lo mismo que la cuenta estandarizada de manera que el área bajo cada polígono es igual a uno.

```
ggplot(data = diamantes, mapping = aes(x = precio, y = ..density..)) +
  geom_freqpoly(mapping = aes(colour = corte), binwidth = 500)
```

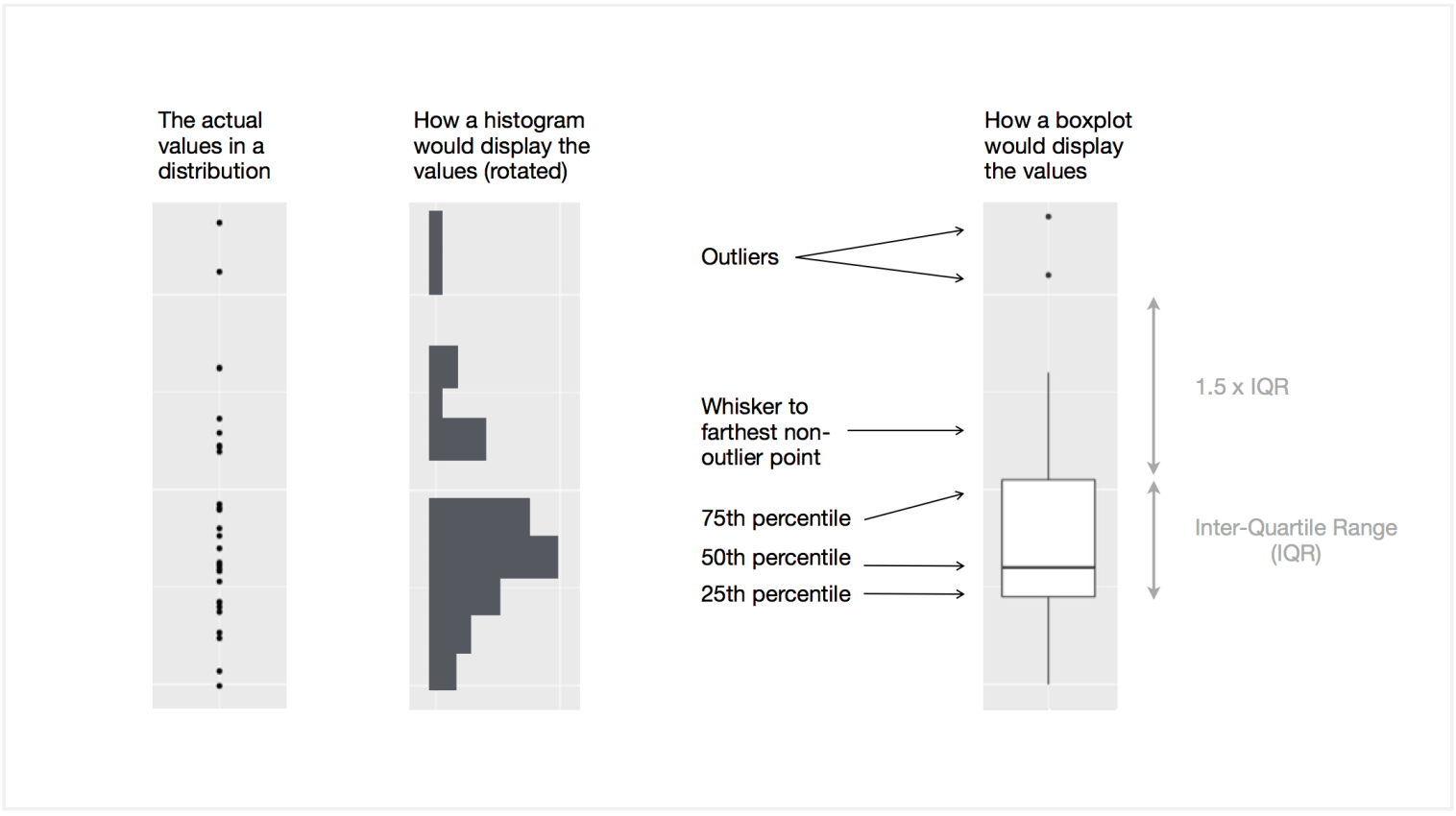
Copy



En este gráfico hay algo un tanto sorprendente: ¡parece que los diamantes regulares (la calidad más baja) tienen, en promedio, el precio más alto! Pero quizá eso sea porque los polígonos de frecuencia son difíciles de interpretar; hay mucho que decir sobre lo que este gráfico muestra.

Otra alternativa para mostrar la distribución de una variable continua agrupada en relación con otra variable categórica es el diagrama de caja (*boxplot* en inglés). Un **diagrama de caja** es un tipo de representación visual para la distribución de valores que es popular entre las personas que se dedican a la estadística. Cada diagrama de caja está integrado por:

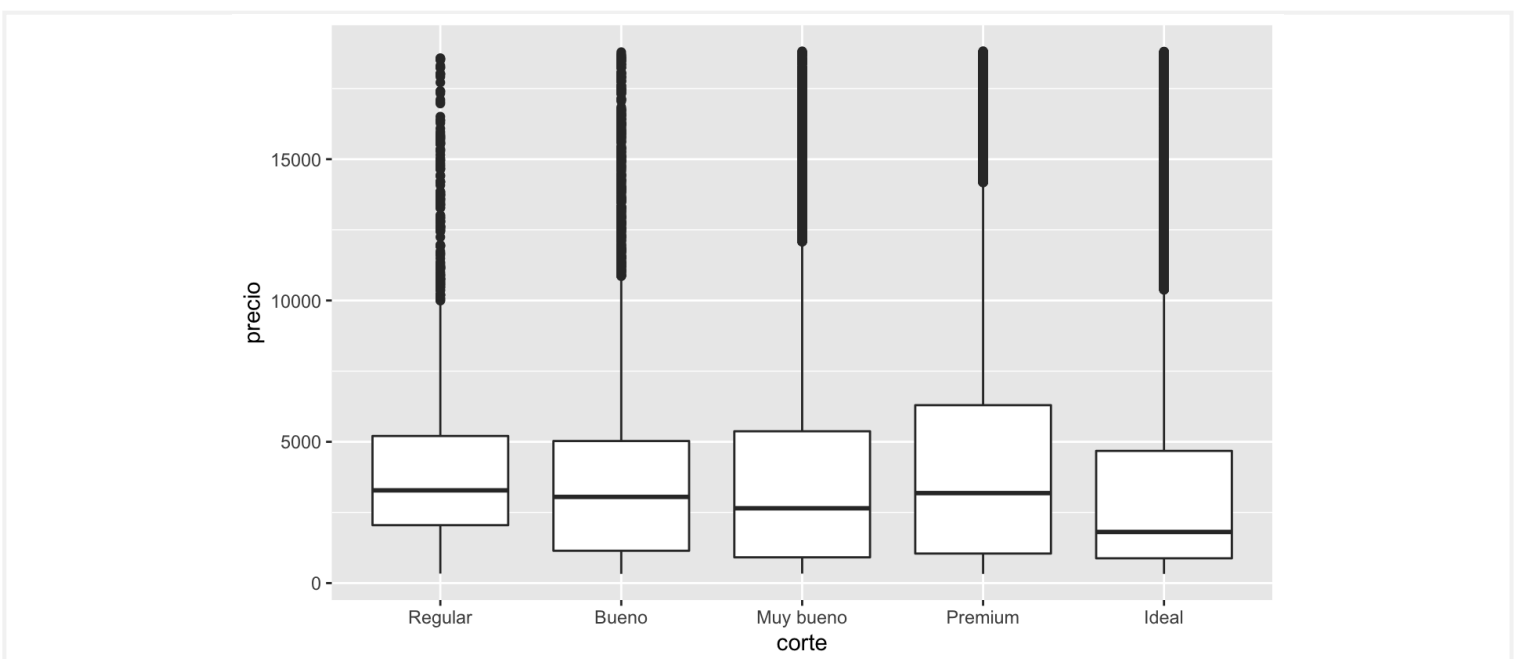
- Una caja que comprende desde el percentil 25 de la distribución hasta el percentil 75, distancia que se conoce como rango intercuartil (*interquartile range* o *IQR* en inglés). En el centro de la caja se encuentra una línea que señala la mediana (es decir, el percentil 50), de la distribución. Estas tres líneas te darán una idea sobre la dispersión de la distribución, así como también te mostrarán si la distribución es simétrica en torno a la mediana o si está sesgada hacia uno de los lados.
- Los puntos que representan observaciones que se encuentran a más de 1.5 veces el *IQR* a partir de cualquier borde de la caja. Estos puntos son inusuales en los datos, de manera que son graficados individualmente.
- Una línea (o bigote) que se extiende a partir de cada borde de la caja hasta el punto más lejano de la distribución que no sea considerado como un valor atípico.



Veamos la distribución de la variable precio en relación con el corte usando `geom_boxplot()` :

```
ggplot(data = diamantes, mapping = aes(x = corte, y = precio)) +
  geom_boxplot()
```

Copy



Si bien vemos mucha menos información sobre la distribución, los diagramas de caja son más compactos, lo que hace que sea más fácil compararlos (y presentar más en un mismo gráfico). Esto respalda el contradictorio resultado al que llegamos antes sobre que los diamantes de mayor calidad son en promedio más baratos. En los ejercicios tendrás que descubrir por qué.

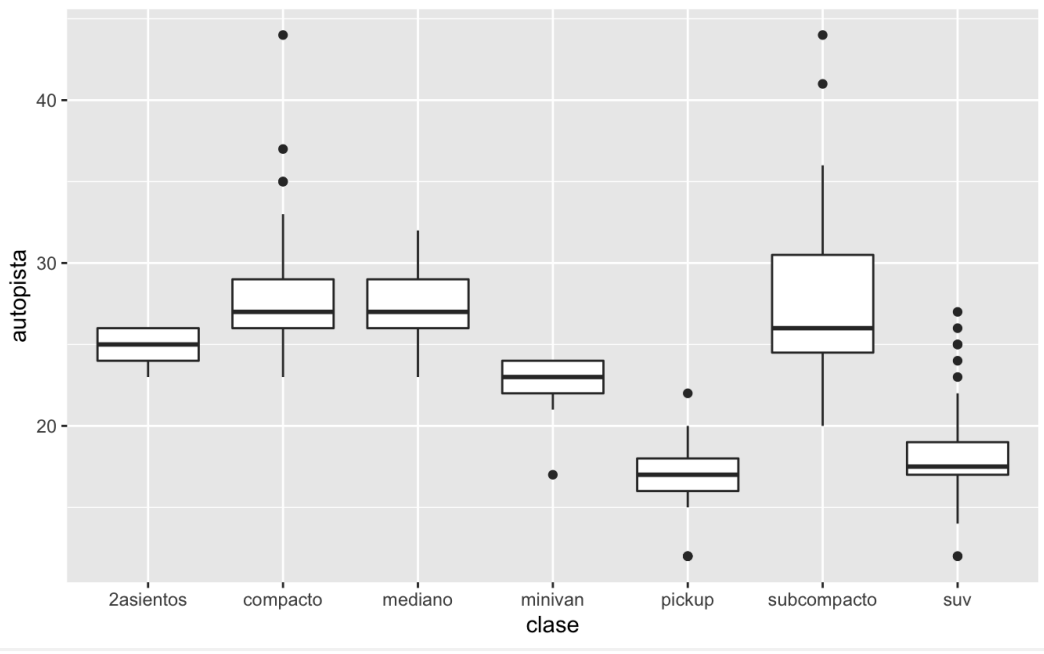
`corte` es un factor ordenado: regular es peor que bueno, que a su vez es peor que muy bueno y así sucesivamente. Muchas variables categóricas no tienen un orden intrínseco como tal, de manera que puedes reordenarlas para mostrarlas de manera más informativa. Una manera de hacerlo es usar la función

`reorder()` (*reordenar*).

Por ejemplo, el caso de la variable `clase` del set de datos `millas`. Podría interesarte saber cómo varía el rendimiento en autopista según la clase del vehículo:

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

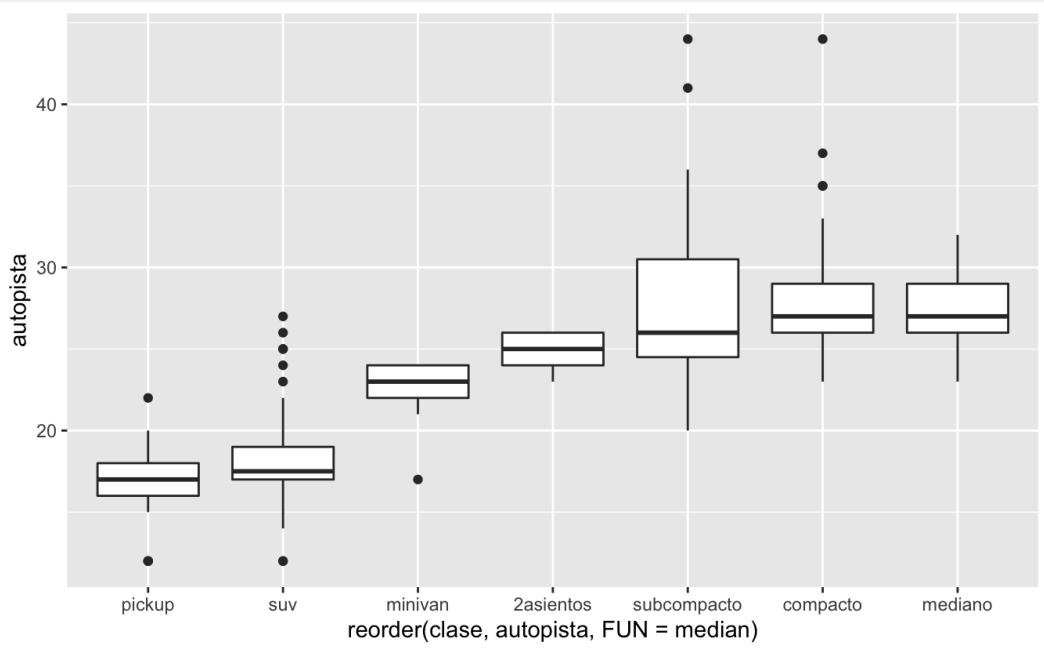
Copy



Para hacer que la tendencia sea más fácil de ver, podemos reordenar la variable `clase` respecto a la mediana de la variable `autopista`:

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

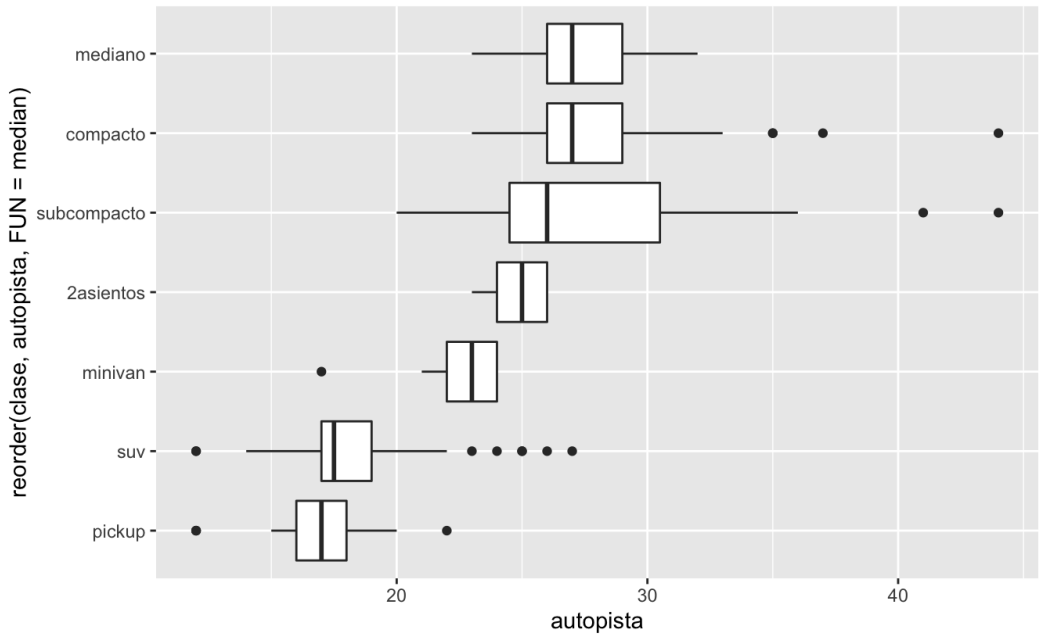
Copy



Si los nombres de tus variables son muy largos, `geom_boxplot()` funcionará mejor si giras el gráfico en 90°. Puedes hacer esto agregando `coord_flip()` (*voltear coordenadas*).

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

Copy



7.5.1.1 **Ejercicios**

1. Usa lo que has aprendido para mejorar la visualización de los tiempos de salida de los vuelos cancelados versus los no cancelados.

2. ¿Qué variable del conjunto de datos de diamantes es más importante para predecir el precio de un diamante? ¿Cómo está correlacionada esta variable con el corte? ¿Por qué la combinación de estas dos relaciones conlleva que los diamantes de menor calidad sean más costosos?

3. Instala el paquete **ggstance**, y crea un diagrama de caja horizontal. ¿Cómo se compara esto a usar `coord_flip()` ?

4. Un problema con los diagramas de caja es que fueron desarrollados en un tiempo en que los set de datos eran más pequeños y por ende tienden a mostrar un número muy grande de "valores atípicos". Una estrategia para remediar este problema es el diagrama **letter value**. Instala el paquete **lvplot**, e intenta usar `geom_lv()` para mostrar la distribución de precio vs corte. ¿Qué observas? ¿Cómo interpretas los gráficos?

5. Compara y contrasta `geom_violin()` con un `geom_histogram()` dividido en facetas, o un `geom_freqpoly()` codificado por colores. ¿Cuáles son las ventajas y desventajas de cada método?

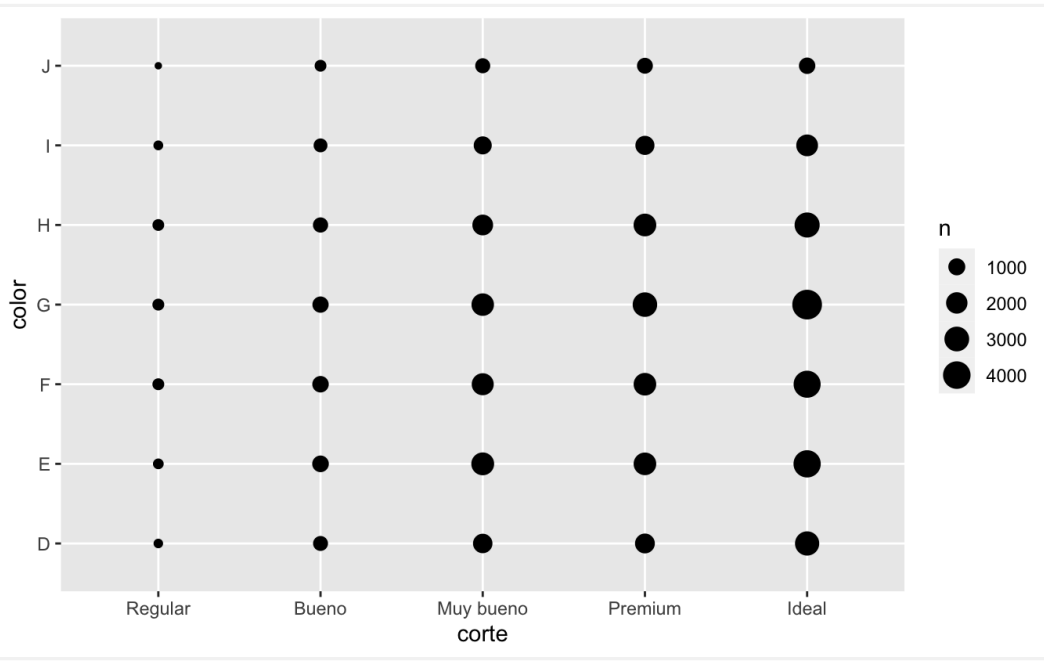
6. Si tu set de datos es pequeño, a veces resulta útil usar `geom_jitter()` para ver la relación entre una variable continua y una categórica El paquete **ggbeeswarm** provee de un número de métodos similares a `geom_jitter()` . Enlístalos y describe brevemente qué hace cada uno.

7.5.2 **Dos variables categóricas**

Para visualizar la variación entre variables categóricas, deberás contar el número de observaciones de cada combinación. Una manera de hacerlo es empleando la función integrada `geom_count()` (del inglés *contar*):

```
ggplot(data = diamantes) +
  geom_count(mapping = aes(x = corte, y = color))
```

Copy



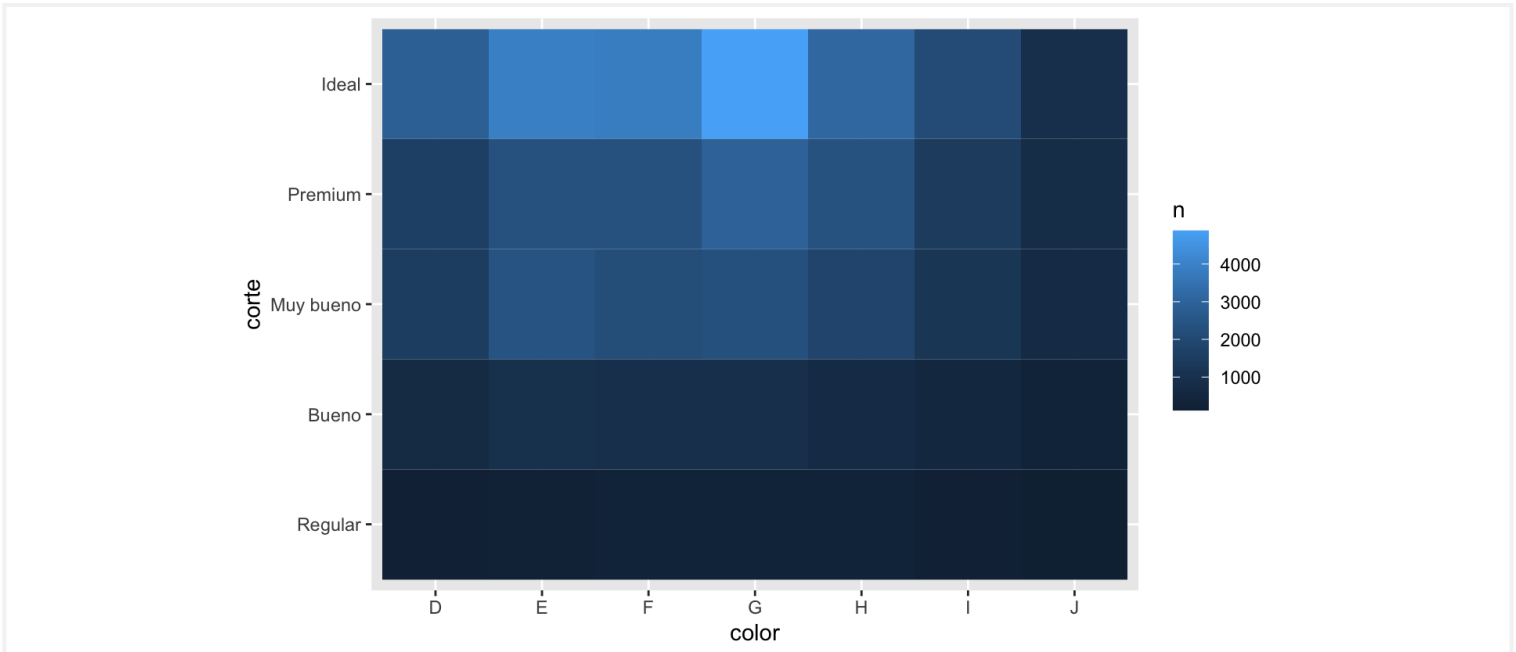
El tamaño de cada círculo en la gráfica muestra cuántas observaciones corresponden a cada combinación de valores. La covariación lucirá como una correlación fuerte entre valores específicos de x y y.

Otra estrategia es calcular el recuento con *dplyr*:

```
diamantes %>%
  count(color, corte)
#> # A tibble: 35 x 3
#>   color corte         n
#>   <ord> <ord>     <int>
#> 1 D     Regular     163
#> 2 D     Bueno       662
#> 3 D     Muy bueno  1513
#> 4 D     Premium    1603
#> 5 D     Ideal      2834
#> 6 E     Regular     224
#> # … with 29 more rows
```

Después podemos visualizar con `geom_tile()` y adaptar la estética de relleno (*fill*):

```
diamantes %>%
  count(color, corte) %>%
  ggplot(mapping = aes(x = color, y = corte)) +
    geom_tile(mapping = aes(fill = n))
```



Si las variables categóricas no siguen un orden particular, podrías usar el paquete **seriation** para reordenar las filas y columnas simultáneamente, de manera que sea más fácil detectar patrones interesantes. En el caso de gráficos más grandes, intenta usar los paquetes **d3heatmap** o **heatmaply**, que crean gráficos interactivos.

7.5.2.1 Ejercicios

1. ¿Cómo podrías cambiar la escala del conjunto de datos anterior para mostrar de manera más clara la distribución del corte dentro del color, o del color dentro de la variable corte?

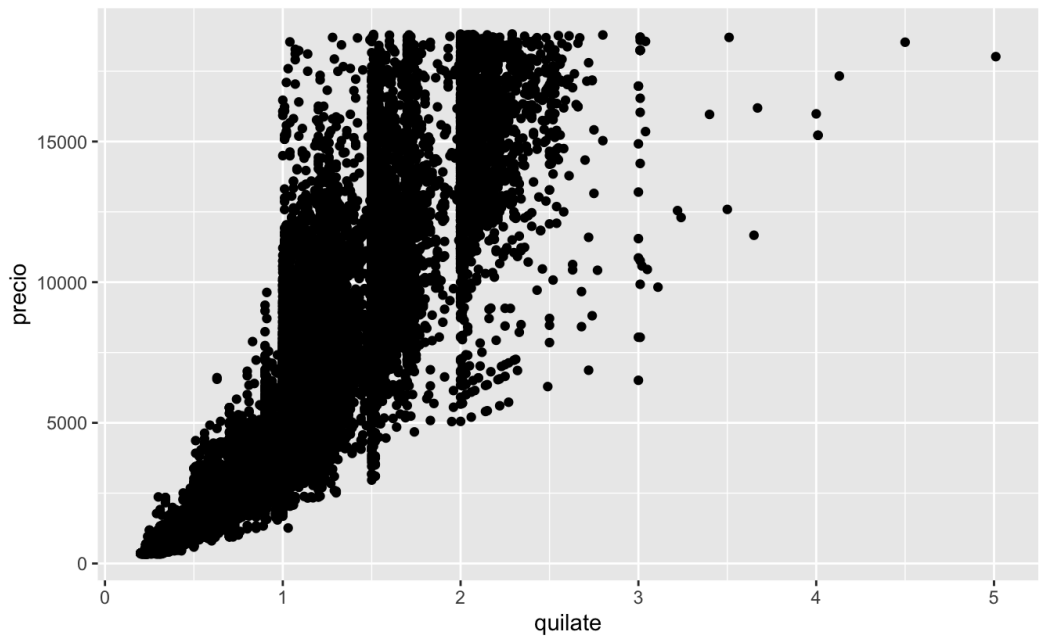
2. Usa `geom_tile()` junto con *dplyr* para explorar la variación del retraso promedio de los vuelos en relación al destino y mes del año. ¿Qué hace que este gráfico sea difícil de leer? ¿Cómo podrías mejorarlo?

3. ¿Por qué es un poco mejor usar `aes(x = color, y = corte)` en lugar de `aes(x = corte, y = color)` en el ejemplo anterior?

7.5.3 Dos variables continuas

Ya viste una manera muy buena de visualizar la covariación entre dos variables continuas: dibujar un diagrama de dispersión (o *scatterplot*) con `geom_point()`. Puedes identificar la covariación como un patrón en los puntos. Por ejemplo, puedes observar la relación exponencial entre los quilates y el precio de un diamante.

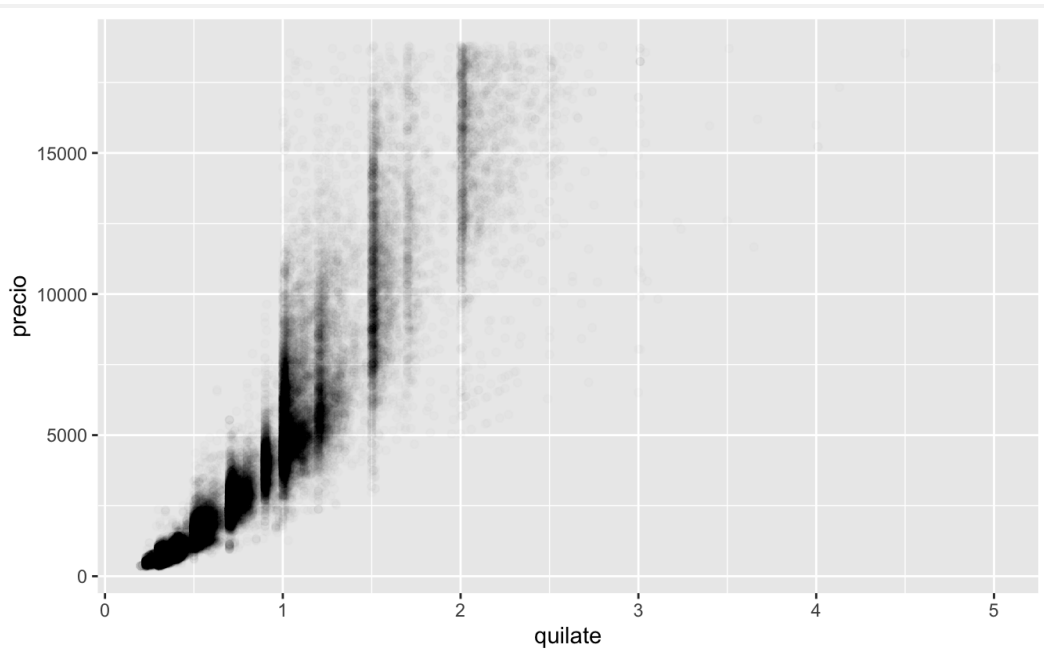
```
ggplot(data = diamantes) +
  geom_point(mapping = aes(x = quilate, y = precio))
```



Los diagramas de dispersión resultan menos útiles a medida que tu set de datos crece, pues los puntos empiezan a superponerse y amontonarse en áreas oscuras uniformes (como en el ejemplo anterior). Ya vimos una manera de revisar el problema: usando el parámetro `alpha` para agregar transparencia.

```
ggplot(data = diamantes) +
  geom_point(mapping = aes(x = quilate, y = precio), alpha = 1 / 100)
```

Copy

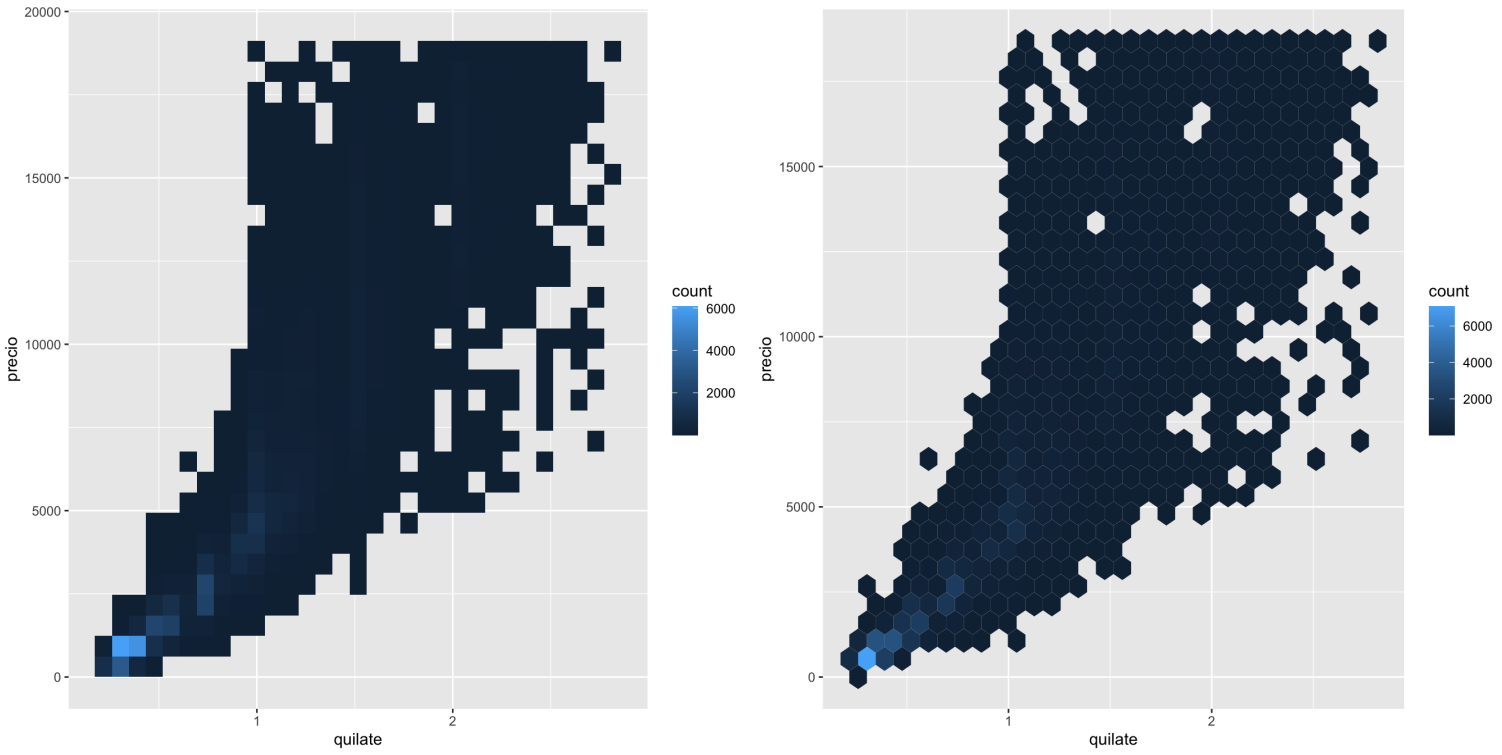


Pero agregar transparencia puede ser difícil para conjuntos de datos muy grandes. Otra solución es modificar el parámetro *bin* (del inglés *contenedor*, que en este caso alude a la idea de *rango* o *unidad*). Anteriormente usaste `geom_histogram()` y `geom_freqpoly()` para segmentar una variable en rangos de manera unidimensional. Ahora aprenderás a usar `geom_bin2d()` y `geom_hex()` para hacerlo en dos dimensiones.

`geom_bin2d()` y `geom_hex()` dividen el plano cartesiano en unidades o contenedores bidimensionales y luego usan un color de relleno para mostrar cuántos puntos pueden ser clasificados en cada contenedor `geom_bin2d()` crea unidades rectangulares. `geom_hex()` crea unidades hexagonales. Deberás instalar el paquete **hexbin** para usar `geom_hex()`.

```
ggplot(data = pequenos) +
  geom_bin2d(mapping = aes(x = quilate, y = precio))

# install.packages("hexbin")
ggplot(data = pequenos) +
  geom_hex(mapping = aes(x = quilate, y = precio))
```

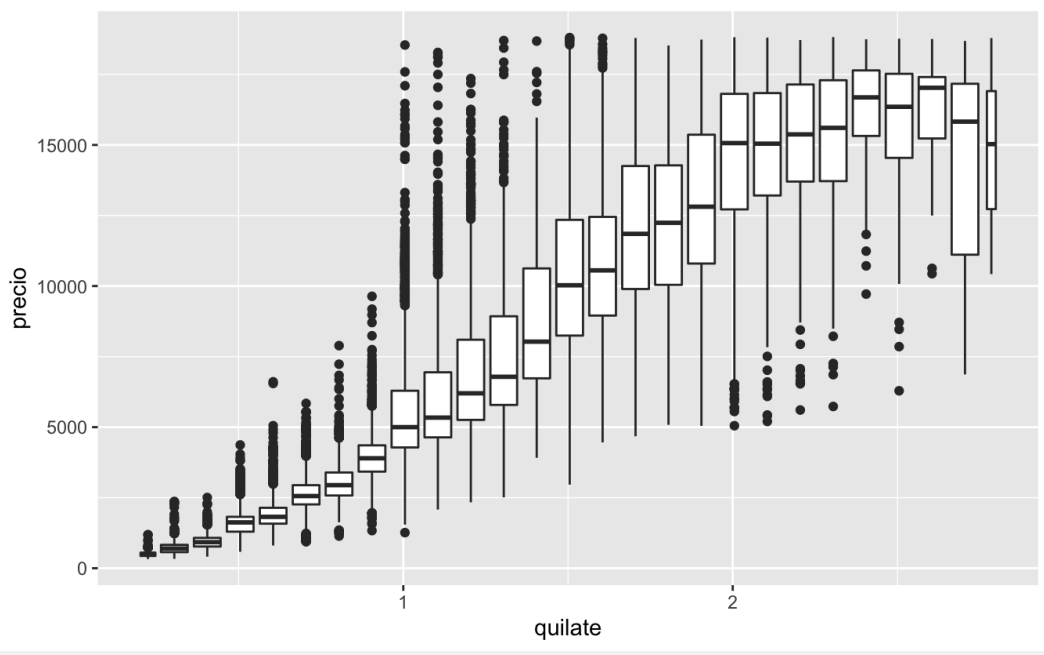
Copy



Otra opción es crear contenedores o intervalos con una de las variables continuas de manera de que pueda ser tratada como una variable categórica. Luego, puedes usar alguna de las técnicas de visualización empleadas para representar la combinación de una variable categórica con una variable continua. Por ejemplo, podrías segmentar la variable `quilate` y definir unidades o intervalos, para después graficar una diagrama de cajas para cada grupo:

```
ggplot(data = pequenos, mapping = aes(x = quilate, y = precio)) +
  geom_boxplot(mapping = aes(group = cut_width(quilate, 0.1)))
```

Copy

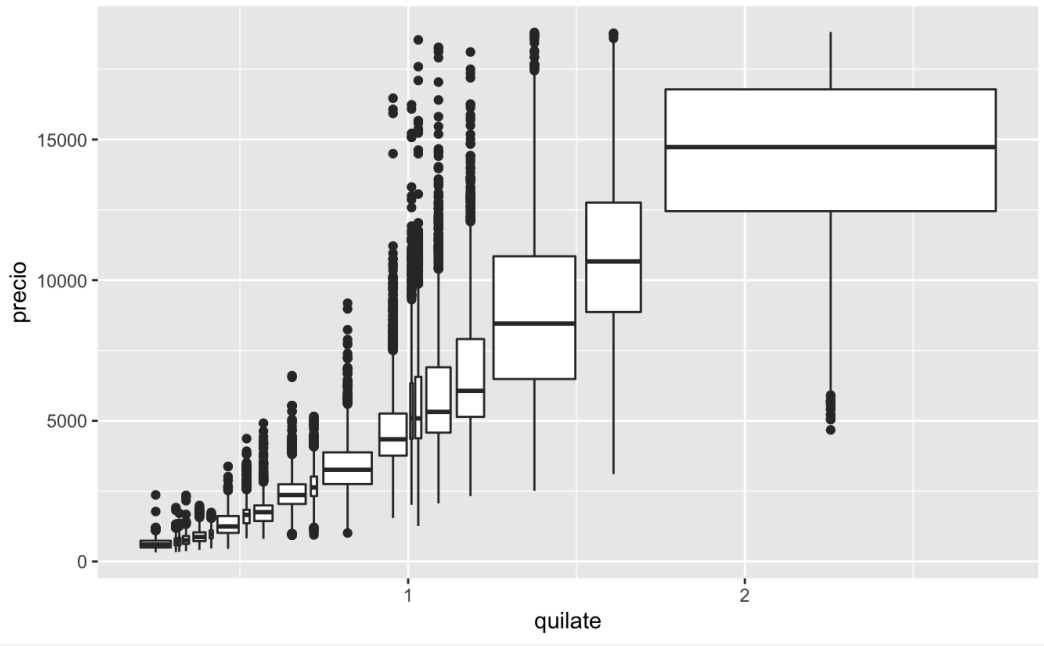


`cut_width(x, width)`, como se muestra en el ejemplo anterior, divide la variable `x` en intervalos de ancho `width`. Los diagramas de caja parecen muy similares por default sin importar el número de observaciones (salvo el número de valores atípicos, o *outliers*), de manera que es difícil entender que cada diagrama de caja representa un número de datos diferente. Una manera de mostrar esto es hacer que el ancho del diagrama de caja sea proporcional al número de datos contenidos en cada caja con `varwidth = TRUE`.

Otra solución es mostrar aproximadamente el mismo número de datos en cada intervalo o unidad. Esto puedes lograrlo con `cut_number()`:

```
ggplot(data = pequenos, mapping = aes(x = quilate, y = precio)) +
  geom_boxplot(mapping = aes(group = cut_number(quilate, 20)))
```

Copy



7.5.3.1 **Ejercicios**

1. En lugar de resumir la distribución condicional con un diagrama de caja, podrías usar un polígono de frecuencia. ¿Qué deberías considerar cuando usas `cut_width()` en comparación con `cut_number()` ? ¿Qué impacto tiene este parámetro en la visualización bidimensional de `quilate` y `precio` ?

2. Visualiza la distribución de `quilate` , segmentada según la variable `precio` .

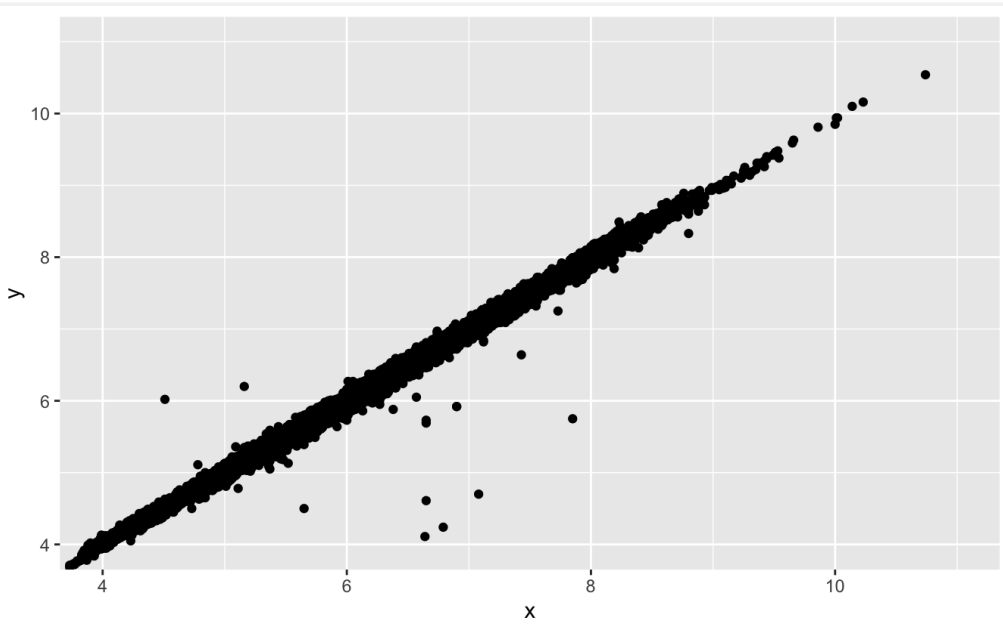
3. ¿Cómo es la distribución del precio de diamantes muy grandes en comparación con aquella de diamantes más pequeños? ¿Es como esperabas, o te resulta sorprendente?

4. Combina dos de las técnicas que has aprendido para visualizar la distribución combinada de las variables `corte` , `quilate` y `precio` .

5. Los gráficos bidimensionales revelan observaciones atípicas que podrían no ser visibles en gráficos unidimensionales. Por ejemplo, algunos puntos en la gráfica a continuación tienen una combinación inusual de valores `x` y `y` , que hace que algunos puntos sean valores atípicos aún cuando sus valores `x` e `y` parecen normales cuando son examinados de manera individual.

```
ggplot(data = diamantes) +
  geom_point(mapping = aes(x = x, y = y)) +
  coord_cartesian(xlim = c(4, 11), ylim = c(4, 11))
```

Copy



¿Por qué es mejor usar un diagrama de dispersión que un diagrama basado en rangos en este caso?

7.6 **Patrones y modelos**

Los patrones en tus datos entregan pistas acerca de las relaciones entre variables. Si existe una relación sistemática entre dos variables, esto aparecerá como un patrón en tus datos. Si encuentras un patrón, hazte las siguientes preguntas:

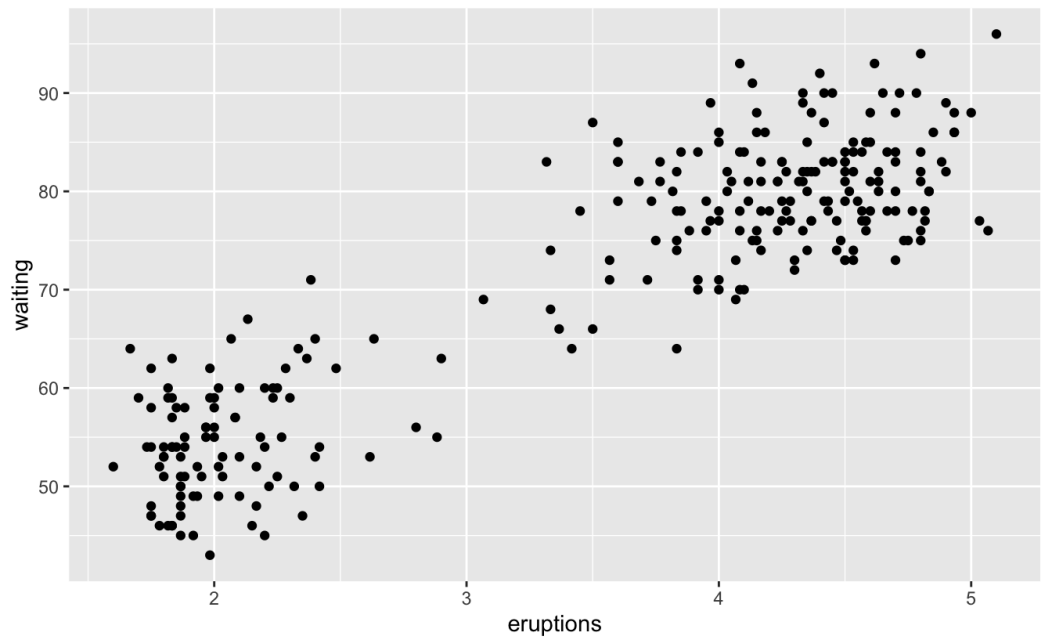
- ¿Este patrón podría ser mera coincidencia (p. ej., producto de probabilidades aleatorias)?
- ¿Cómo podrías describir la relación sugerida por este patrón?

- ¿Qué tan fuerte es la relación sugerida por este patrón?
- ¿Qué otras variables podrían afectar la relación?
- ¿Cambia esta relación si examinas de manera individual distintos subgrupos de datos?

Un diagrama de dispersión de la duración de las erupciones del géiser Viejo Fiel contra el tiempo de espera entre erupciones muestra un patrón: tiempos de espera más largos están asociados con erupciones más largas. El diagrama de dispersión también muestra los dos grupos que identificamos antes.

```
ggplot(data = faithful) +
  geom_point(mapping = aes(x = eruptions, y = waiting))
```

Copy

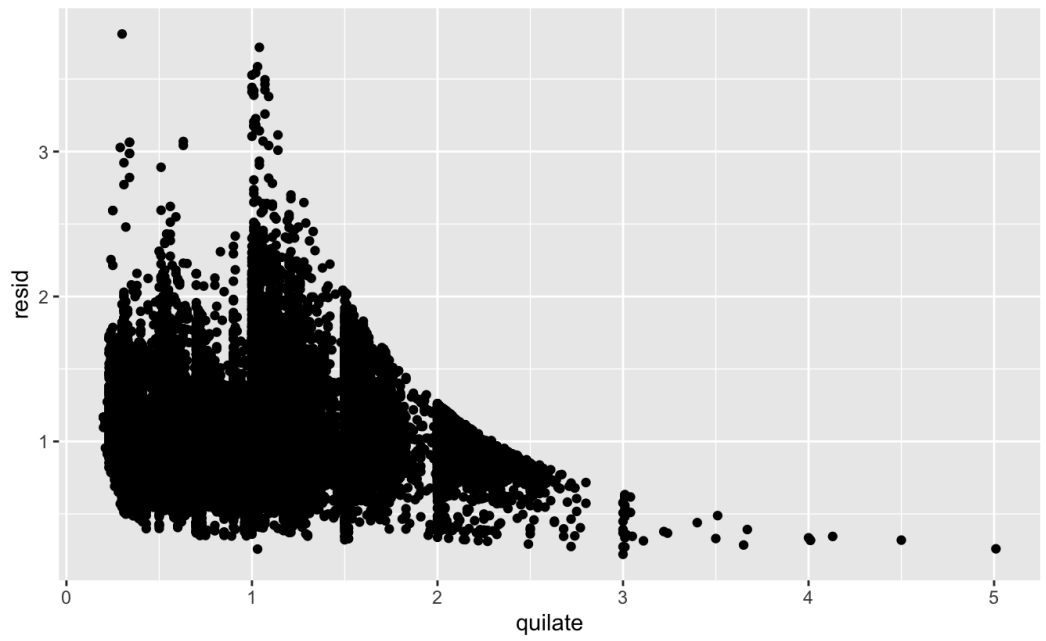


Los patrones son una de las herramientas más útiles para quienes hacen ciencia de datos, pues permiten revelar covariación. Si piensas en la variación como un fenómeno que crea incertidumbre, la covariación es un fenómeno que la reduce. Si dos variables varían de manera conjunta, puedes usar los valores de una variable para hacer mejores predicciones sobre valores de la segunda. Si la covariación es producto de una relación causal (un caso especial), entonces puedes usar el valor de una variable para controlar el valor de la segunda.

Los modelos son una herramienta para extraer patrones de los datos. Por ejemplo, considera los datos sobre diamantes. Resulta difícil entender la relación entre corte y precio, pues corte y quilate, así como quilate y precio, están estrechamente relacionadas. Es posible usar un modelo para remover la fuerte relación entre precio y quilate de manera que podamos explorar las sutilezas que quedan en los datos. El código mostrado a continuación crea un modelo que predice el `precio` a partir de la variable `quilate` y después calcula los residuales (la diferencia entre la variable predecida y el valor real). Los residuales nos informan acerca del precio de un diamante, una vez que el efecto que los quilates tienen sobre esta variable ha sido removido.

```
library(modelr)

mod <- lm(log(precio) ~ log(quilate), data = diamantes)

diamantes2 <- diamantes %>%
  add_residuals(mod) %>%
  mutate(resid = exp(resid))

ggplot(data = diamantes2) +
  geom_point(mapping = aes(x = quilate, y = resid))
```

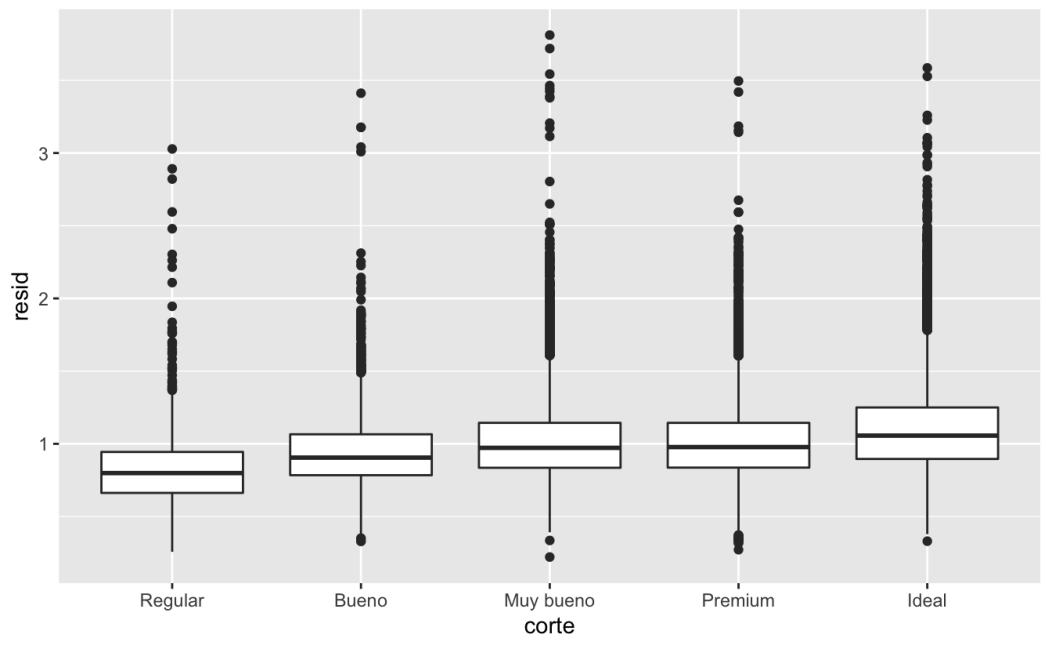
Copy



Una vez que has removido la fuerte relación entre quilate y precio, puedes observar lo que esperarías sobre la relación entre corte y precio: los diamantes de mejor calidad son más costosos según su tamaño.

```
ggplot(data = diamantes2) +
  geom_boxplot(mapping = aes(x = corte, y = resid))
```

Copy



Aprenderás cómo funciona el modelado estadístico y el paquete **modelr** en una de las secciones finales de este libro, Modelos. Dejamos el modelado para más adelante pues es más fácil entender qué son los modelos y cómo funcionan una vez que cuentes con herramientas para manejo de datos y de programación.

7.7 **Argumentos en ggplot2**

A medida que nos alejamos de estos capítulos introductorios, usaremos expresiones más concisas para escribir código de ggplot2. Hasta ahora hemos sido muy explícitos en el código, lo que es muy útil cuando estás aprendiendo:

```
ggplot(data = faithful, mapping = aes(x = eruptions)) +
  geom_freqpoly(binwidth = 0.25)
```

Copy

Usualmente los primeros argumentos de una función son tan importantes que deberías saberlos de memoria. Los primeros dos argumentos de `ggplot()` son `data` y `mapping`, y los primeros dos argumentos de `aes()` son `x` e `y`. En lo que resta del libro no escribiremos esos nombres. Esto nos ahorra teclear y, al reducir la cantidad de palabras repetidas, será más fácil identificar lo que es distinto entre diferentes gráficos. Esta es una cuestión importante en cuanto a programación se refiere, y hablaremos más sobre esto en el capítulo sobre funciones.

Escribir el código para el gráfico anterior de manera más precisa resulta en:

```
ggplot(faithful, aes(eruptions)) +
  geom_freqpoly(binwidth = 0.25)
```

Copy

Algunas veces el producto obtenido en el último paso del proceso de transformación de datos lo convertiremos en un gráfico. Observa también la transición de `%>%` a `+`. Nos gustaría que esta transición no fuera necesaria, pero desafortunadamente `ggplot2` fue creado antes de que el *pipe* fuera descubierto.

```
diamantes %>%
  count(corte, claridad) %>%
  ggplot(aes(claridad, corte, fill = n)) +
    geom_tile()
```

Copy

7.8 Aprendiendo más

Si quieres aprender más acerca de la mecánica de `ggplot2`, te recomendamos consultar una copia del *libro de ggplot2* (disponible en inglés). Fue actualizado recientemente, de manera que incluye código para dplyr y tidyr, y tiene mucho más espacio para explorar todas las facetas de visualización. Desafortunadamente, el libro no está disponible sin costo, pero si tienes alguna relación con una universidad es probable que puedas obtener una versión electrónica del libro mediante SpringerLink.

Otro recurso útil es el *R Graphics Cookbook* por Winston Chang. Gran parte del contenido de este libro está disponible en línea en http://www.cookbook-r.com/Graphs/.

También recomiendo *Graphical Data Analysis with R*, de Antony Unwin. Este libro trata el material comprendido en este capítulo, pero tiene el espacio para explicar con mayor detalle y profundidad.

« 6 Flujo de trabajo: Scripts

8 Flujo de trabajo: proyectos »

8 Flujo de trabajo: proyectos

Llegará el día en que tendrás que salir de R, ir a hacer otra cosa y después volver a tu análisis al día siguiente. Llegará el día en que estés trabajando simultáneamente en múltiples análisis en R y quieras mantenerlos separados. Llegará el día en que tendrás que importar datos del mundo exterior a R y exportar resultados numéricos y gráficos creados en R hacia el exterior. Para manejar estas situaciones de la vida real, tienes que tomar dos decisiones:

1. ¿Qué parte de tu análisis es "real"? Es decir, ¿qué vas a guardar como registro permanente de lo sucedido durante el análisis?

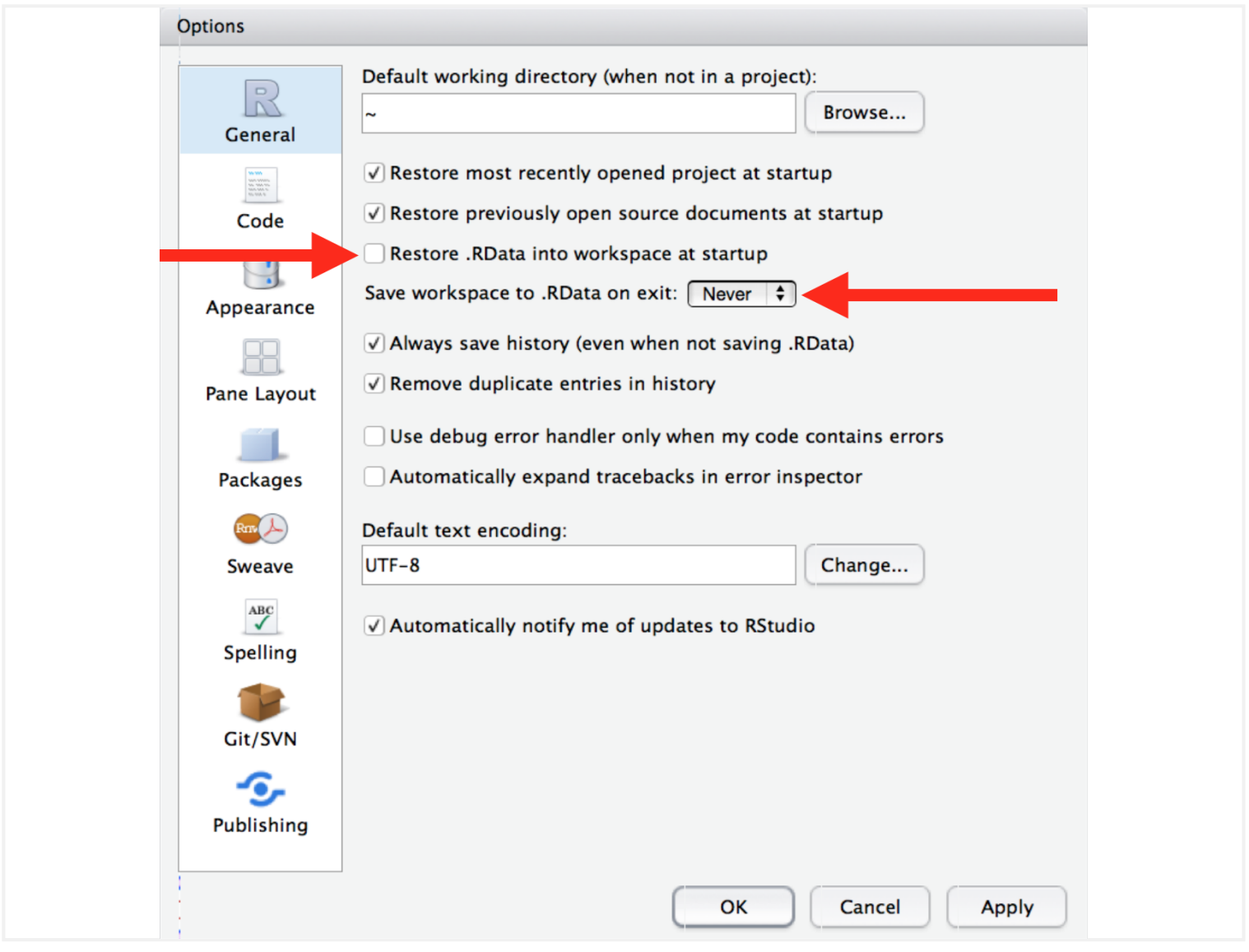
2. ¿Dónde "vive" tu análisis?

8.1 ¿Qué es real?

Si estás recién empezando a utilizar R, está bien que consideres "real" tu ambiente (esto es, los objetos listados en el panel *Environment*). Sin embargo, a largo plazo tendrás una mejor experiencia usando R si consideras que tus *scripts* son lo "real".

Con tus *scripts* de R (y tus archivos con datos) puedes volver a crear el ambiente. ¡Es mucho más difícil volver a crear tus *scripts* a partir de tu ambiente! Para hacerlo, tendrías que reescribir muchas líneas de código de memoria (cometiendo errores durante el proceso), o analizar con muchísima atención tu historial en R.

Para promover este comportamiento, te recomiendo que configures RStudio para que no guarde tu espacio de trabajo (*workspace*) entre sesiones:



Esto te va a hacer sufrir a corto plazo, pues cuando reinicies RStudio este no recordará los resultados del código que ejecutaste la última vez. Pero este sufrimiento momentáneo te va a salvar de una agonía a largo plazo, ya que te obligará a capturar en tu código todas las interacciones importantes. No hay nada peor que descubrir tres meses más tarde que sólo guardaste los resultados de un cálculo importante en tu espacio de trabajo y no el cálculo en sí en el código.

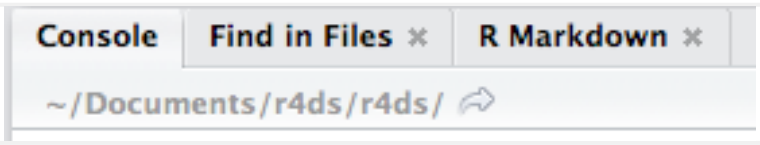
Hay un par de métodos abreviados de teclado (*keyboard shortcuts*) geniales que te ayudarán a asegurarte de que las partes importantes de tu código están capturadas en el editor:

1. Presiona Cmd/Ctrl + Shift + F10 para reiniciar RStudio.
2. Presiona Cmd/Ctrl + Shift + S para volver a ejecutar el *script* actual.

Nosotros usamos este patrón cientos de veces a la semana.

8.2 ¿Dónde vive tu análisis?

R posee el poderoso concepto de **directorio de trabajo** (*working directory* en inglés). Aquí es donde R busca los archivos que le pides que lea y donde colocará todos los archivos que le pidas que guarde. RStudio muestra tu directorio de trabajo actual en la parte superior de la consola:



Puedes imprimir esto en código de R ejecutando el comando `getwd()` (en inglés *get working directory*, que significa *obtener directorio de trabajo*):

```
getwd()
# > [1] "/Users/hadley/Documents/r4ds/r4ds"
```

Copy

Si estás recién empezando a utilizar R, está bien que tu directorio *home* (carpeta personal o de usuario), tu directorio de documentos o cualquier otro directorio en tu computadora sea el directorio de trabajo de R. Pero estás en el capítulo seis de este libro y ya no eres un/a principiante. Pronto deberías dar el siguiente paso y comenzar a organizar tus proyectos de análisis en directorios y, ya trabajando en un proyecto, definir que el directorio de trabajo de R sea el directorio asociado a dicho proyecto en particular.

__No lo recomendamos, pero también es posible definir el directorio de trabajo desde R:

```
setwd("/ruta/a/mi/ProyectoInteresante")
```

Copy

Sin embargo, nunca deberías hacerlo de esta forma, pues existe una mejor manera, una que te lleva por la ruta para manejar tu trabajo en R como una persona experta.

8.3 Rutas y directorios

Las rutas (del inglés *paths*) y los directorios son un poco complicados porque existen dos estilos básicos de rutas: Mac/Linux y Windows. Las tres diferencias principales entre estos estilos son:

1. Cómo separas los componentes de la ruta. Mac y Linux usan barras (*slash*) (p. ej. `graficos/diamantes.pdf`), mientras que Windows utiliza barras invertidas (*backslash*) (p. ej., `graficos\diamantes.pdf`). R puede trabajar con ambos tipos (sin importar qué plataforma estés usando actualmente). Desafortunadamente, las barras invertidas tienen un significado especial en R y para poder escribir utilizarlas en la ruta de un archivo, ¡tendrás que teclear dos barras invertidas! Eso hace que la vida sea frustrante, así que recomendamos usar siempre el estilo Linux/Mac con las barras (`/`).

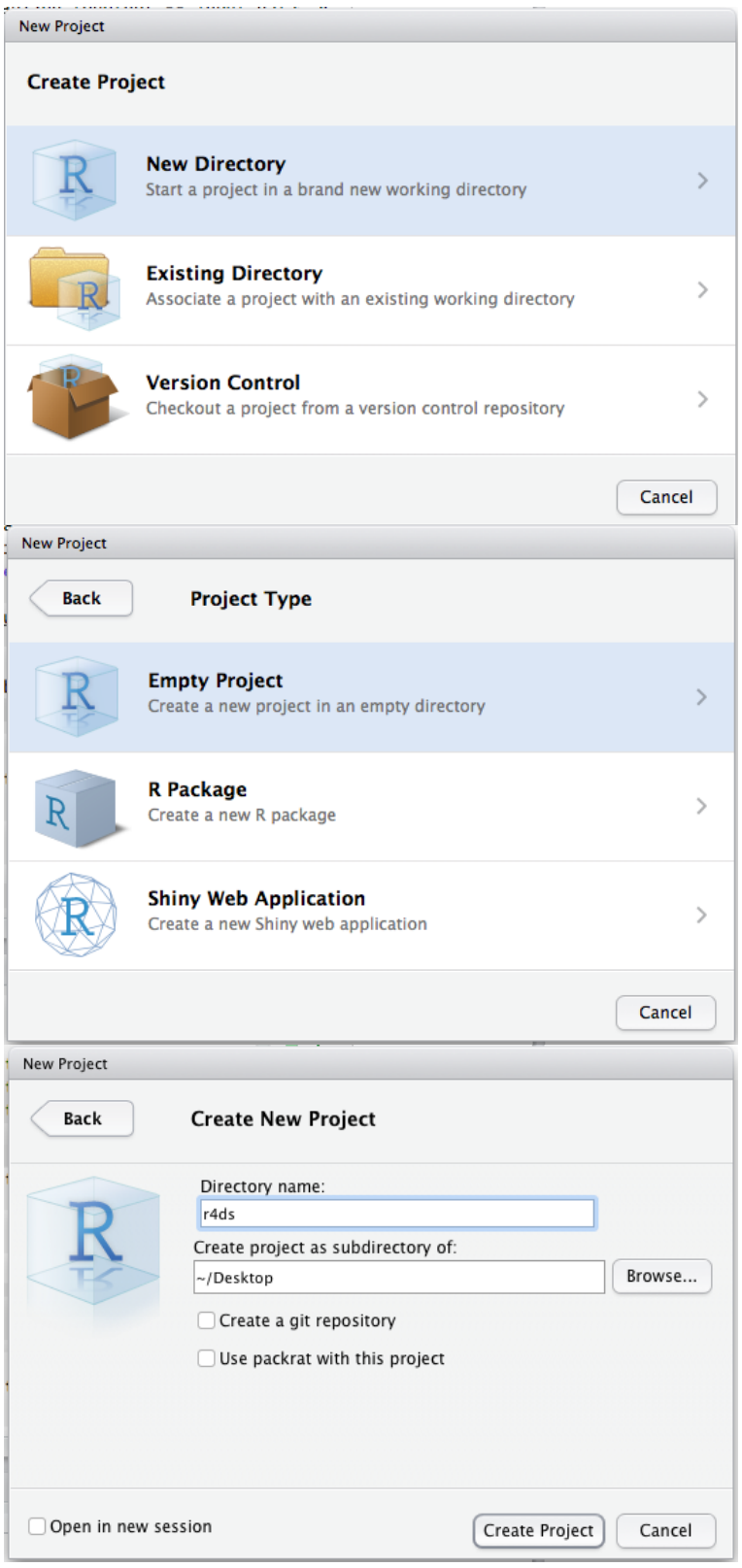
2. Las rutas absolutas (es decir, las rutas que te llevan a la misma ubicación sin importar tu directorio de trabajo) lucen diferentes. En Windows dichas rutas comienzan con la letra del disco (p. ej. `C:`) o dos barras invertidas (p. ej. `\\nombredelservidor`) y en Mac/Linux comienzan con una barra "/" (p. ej. `/users/hadley`). **Nunca** deberías usar rutas absolutas en tus *scripts*, ya que afectan negativamente el proceso de compartir archivos: nadie más va a tener exactamente la misma configuración de directorios que tú.

3. La última diferencia es la ubicación a la que `~` se refiere. `~` es un conveniente acceso directo a tu directorio personal. Windows no cuenta realmente con la noción de un directorio personal, así que se refiere a tu directorio de documentos.

8.4 Proyectos en RStudio

Las personas expertas en R mantienen todos los archivos asociados a un proyecto en un mismo lugar — datos de entrada, *scripts*, resultados, gráficos. Esta es una práctica tan acertada y común, que RStudio cuenta con soporte integrado para esto por medio de los **proyectos**.

Hagamos un proyecto para que puedas usarlo mientras trabajas con el resto de este libro. Haz clic en File > New Proyect, y después:



Escribe `r4ds` como nombre de tu proyecto y piensa en qué *subdirectorio* quieres guardar el proyecto. Si no lo guardas en una ubicación razonable ¡será difícil encontrarlo en el futuro!

Una vez que hayas terminado con este proceso, tendrás un nuevo proyecto en RStudio para este libro. Verifica que el directorio "personal" de tu proyecto sea el actual directorio de trabajo:

```
getwd()
#> [1] /Users/hadley/Documents/r4ds/r4ds
```

Copy

Siempre que hagas referencia a un archivo con una ruta relativa, RStudio lo buscará en este directorio.

Ahora teclea los siguientes comandos en el editor de *scripts* y luego guarda el archivo como "diamantes.R". A continuación, ejecuta el *script* completo; esto guardará un archivo PDF y un archivo CSV en el directorio de tu proyecto. No te preocupes por los detalles, aprenderás acerca de ellos más adelante en este libro.

On this page

View source

Edit this page

```
library(tidyverse)
library(datos)


ggplot(diamantes, aes(quilate, precio)) +
  geom_hex()
ggsave("diamantes.pdf")

write_csv(diamantes, "diamantes.csv")
```

Copy

Cierra RStudio. Inspecciona la carpeta asociada a tu proyecto — encontrarás que hay un archivo `.Rproj` allí. Haz doble clic en ese archivo para reabrir el proyecto. Nota que al hacer esto, vuelves al punto donde estabas justo antes de cerrar RStudio: es el mismo directorio de trabajo e historial de comandos, y todos los archivos con los que estabas trabajando siguen abiertos. Sin embargo, dado que seguiste las instrucciones que te dimos anteriormente, tendrás un ambiente (*Environment*) completamente limpio, lo que te asegura que estás empezando desde cero.

Según tu sistema operativo y de la manera que prefieras, busca en tu computadora el archivo `diamantes.pdf` . Encontrarás el archivo PDF (lo que no es precisamente una sorpresa), pero *también el script que lo creó* (`diamantes.R`). ¡Esta es una ventaja increíble! Un día vas a querer crear un gráfico o simplemente entender de dónde vino. Si siempre guardas gráficos por medio de *comandos en código* y nunca con el ratón o el portapapeles, ¡serás capaz de reproducir fácilmente trabajo pasado!

8.5 Resumen

En resumen, los proyectos en RStudio te ofrecen la oportunidad de mantener un flujo de trabajo consistente que te será de mucha utilidad en el futuro:

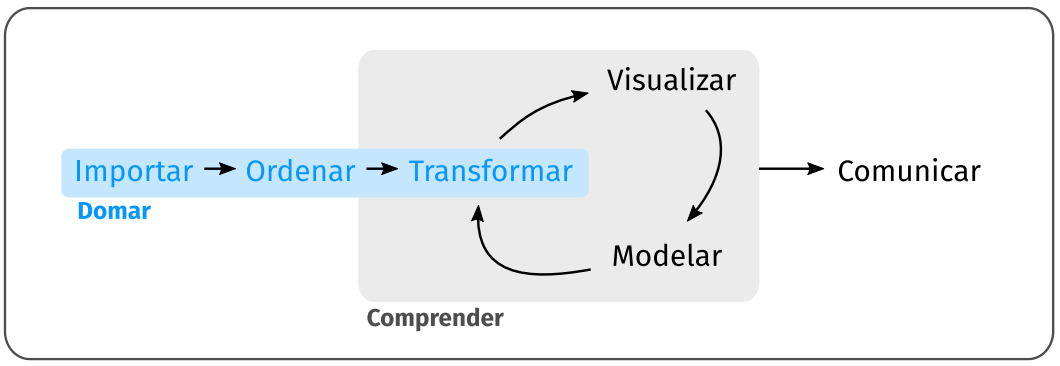
- Crea un proyecto RStudio para cada proyecto de análisis de datos.
- Mantén los archivos ahí; ya hablaremos de cómo leerlos en R en el capítulo sobre importación de datos.
- Conserva los *scripts* ahí también; edítalos, ejecutálos por partes o en su totalidad.
- Guarda todos los resultados de tu trabajo (gráficos y sets de datos limpios) ahí.
- Siempre usa rutas relativas, nunca rutas absolutas.

Todo lo que necesites siempre estará concentrado en una sola ubicación y claramente separado de los otros proyectos en los que estés trabajando.

« 7 Análisis exploratorio de datos (EDA)
9 Introducción »

9 Introducción

En esta parte del libro, aprenderás cómo manejar o "domar" datos (*data wrangling*, en inglés): el arte de tener tus datos en R de una forma conveniente para su visualización y modelado. Si bien en español también se suele hacer referencia a esta etapa como *manipulación* de datos, hemos mantenido también la traducción literal ya que, como se señala en la introducción, la noción de domar (*wrangling*) apunta a lo difícil que puede ser a veces este proceso. La doma de datos es muy importante: ¡sin ella no puedes trabajar con tus propios datos! Este proceso tiene tres partes principales:



Esta parte del libro continúa de la siguiente forma:

- En Tibbles aprenderás sobre la variante de *data frame* que usamos en este libro: el **tibble**. Conocerás qué los hace diferentes de los data frames comunes y cómo puedes construirlos "a mano".
- En Importación de datos aprenderás cómo traer tus datos del disco hacia R. Nos enfocaremos en los formatos rectangulares de texto plano, pero daremos referencias a paquetes que ayudan con otros tipos de datos.
- En Datos ordenados aprenderás una manera consistente de almacenar tus datos que facilita la transformación, la visualización y el modelado. Aprenderás los principios subyacentes y cómo poner tus datos en una forma ordenada.

La doma de datos también abarca la transformación de los mismos, algo sobre lo que ya has aprendido un poco. Ahora nos enfocaremos en nuevas habilidades para cuatro tipos de datos específicos que encontrarás frecuentemente en la práctica:

- La sección Datos relacionales te dará herramientas para trabajar con múltiples conjuntos de datos interrelacionados.
- Cadenas de caracteres te introducirá en las expresiones regulares (*regular expressions*), una herramienta poderosa para manipular cadenas de caracteres (*strings*).
- En Factores veremos cómo R almacena los datos categóricos. Los factores se utilizan cuando una variable tiene un conjunto fijo de posibles valores, o cuando quieres usar una cadena de caracteres en un orden distinto al alfabético.
- Fechas y horas te dará herramientas clave para trabajar con fechas y fecha-horas.

« 8 Flujo de trabajo: proyectos

10 Tibbles »

View source

Edit this page

10 Tibbles

10.1 Introducción

A lo largo de este libro trabajaremos con "tibbles" (que se pronuncia /tibls/) en lugar de los tradicionales `data.frame` de R. Los tibbles **son** *data frames*, pero modifican algunas características antiguas para hacernos la vida más fácil. R es un lenguaje viejo y algunas cosas que eran útiles hace 10 o 20 años actualmente pueden resultar inconvenientes. Es difícil modificar R base sin romper código existente, así que la mayor parte de la innovación ocurre a través de paquetes. Aquí describiremos el paquete **tibble**, que provee una versión de *data frame* que facilita el trabajo con el tidyverse. La mayoría de las veces usaremos el término *tibble* y *data frame* de manera indistinta; cuando queramos referirnos de manera particular al *data frame* que viene incluido en R lo llamaremos `data.frame`.

Si luego de leer este capítulo te quedas con ganas de aprender más sobre tibbles, quizás disfrutes `vignette("tibble")`.

10.1.1 Prerequisitos

En este capítulo exploraremos el paquete **tibble**, parte de los paquetes principales del tidyverse. Para ejemplificar utilizaremos *datasets* incluidos en el paquete **datos**.

```
library(tidyverse)
library(datos)
```

Copy

10.2 Creando tibbles

La mayoría de las funciones que usarás en este libro producen tibbles, ya que son una de las características trasversales del tidyverse. La mayoría de los paquetes de R suelen usar data frames clásicos, así que algo que podrías querer hacer es convertir un data frame en un tibble. Esto lo puedes hacer con `as_tibble()`:

```
as_tibble(flores)
#> # A tibble: 150 x 5
#>   Largo.Sepalo Ancho.Sepalo Largo.Petalo Ancho.Petalo Especies
#>          <dbl>        <dbl>        <dbl>        <dbl> <fct>
#> 1          5.1          3.5          1.4          0.2 setosa
#> 2          4.9          3            1.4          0.2 setosa
#> 3          4.7          3.2          1.3          0.2 setosa
#> 4          4.6          3.1          1.5          0.2 setosa
#> 5          5            3.6          1.4          0.2 setosa
#> 6          5.4          3.9          1.7          0.4 setosa
#> # … with 144 more rows
```

Copy

Puedes crear un nuevo tibble a partir de vectores individuales con `tibble()`. Esta función recicla vectores de longitud 1 automáticamente y te permite usar variables creadas dentro de la propia función, como se muestra abajo.

```
tibble(
  x = 1:5,
  y = 1,
  z = x^2 + y
)
#> # A tibble: 5 x 3
#>       x     y     z
#>   <int> <dbl> <dbl>
#> 1     1     1     2
#> 2     2     1     5
#> 3     3     1    10
#> 4     4     1    17
#> 5     5     1    26
```

Si ya te has familiarizado con `data.frame()`, es importante que tomes en cuenta que `tibble()` hace menos cosas: nunca cambia el tipo de los inputs (p. ej., ¡nunca convierte caracteres en factores!), nunca cambia el nombre de las variables y nunca asigna nombres a las filas.

Un tibble puede usar nombres de columnas que no son nombres de variables válidos en R (también conocidos como nombres **no sintácticos**). Por ejemplo, pueden empezar con un caracter diferente a una letra o contener caracteres poco comunes, como espacios. Para referirse a estas variables, tienes que rodearlos de acentos graves, `` ` ``:

```
tb <- tibble(
  `:)` = "sonrisa",
  ` ` = "espacio",
  `2000` = "número"
)
tb
#> # A tibble: 1 x 3
#>   `:)`    ` `     `2000`
#>   <chr>   <chr>   <chr>
#> 1 sonrisa espacio número
```

También necesitarás los acentos graves al trabajar con estas variables en otros paquetes, como ggplot2, dplyr y tidyr.

Otra forma de crear un tibble es con `tribble()`, que es una abreviación de tibble **tr**anspuesto. Esta función está pensada para realizar la entrada de datos en el código: los nombres de las columnas se definen con fórmulas (esto es, comienzan con `~`) y cada entrada está separada por comas. Esto permite escribir pocos datos de manera legible.

```
tribble(
  ~x, ~y, ~z,
  #--|--|----
  "a", 2, 3.6,
  "b", 1, 8.5
)
#> # A tibble: 2 x 3
#>   x         y     z
#>   <chr> <dbl> <dbl>
#> 1 a         2   3.6
#> 2 b         1   8.5
```

Usualmente agregamos un comentario para dejar en claro cuál es el encabezado (esta línea debe empezar con `#`).

10.3 Tibbles vs. data.frame

Existen dos diferencias principales entre el uso de un tibble y un `data.frame` clásico: la impresión en la consola y la selección de los subconjuntos.

10.3.1 Impresión en la consola

Los tibbles tienen un método de impresión en la consola refinado: solo muestran las primeras 10 filas y solo aquellas columnas que entran en el ancho de la pantalla. Esto simplifica y facilita trabajar con bases de datos grandes. Además del nombre, cada columna muestra su tipo. Esto último es una gran característica tomada de `str()`.

```
tibble(
  a = lubridate::now() + runif(1e3) * 86400,
  b = lubridate::today() + runif(1e3) * 30,
  c = 1:1e3,
  d = runif(1e3),
  e = sample(letters, 1e3, replace = TRUE)
)
#> # A tibble: 1,000 x 5
#>   a                   b              c     d e
#>   <dttm>              <date>     <int> <dbl> <chr>
#> 1 2021-01-18 19:40:47 2021-01-25     1 0.368 n
#> 2 2021-01-19 13:45:57 2021-01-30     2 0.612 l
#> 3 2021-01-19 08:09:36 2021-02-09     3 0.415 p
#> 4 2021-01-18 21:30:53 2021-02-08     4 0.212 m
#> 5 2021-01-18 17:55:10 2021-02-05     5 0.733 i
#> 6 2021-01-19 04:56:07 2021-02-01     6 0.460 n
#> # … with 994 more rows
```

Copy

Los tibbles están diseñados para no inundar tu consola accidentalmente al mirar data frames muy grandes. Sin embargo, a veces es necesario un output mayor que el que se obtiene por defecto. Existen algunas opciones que pueden ayudar.

Primero, puedes usar `print()` en el data frame y controlar el número de filas (`n`) y el ancho (`width`) mostrado. Por otro lado, `width = Inf` muestra todas las columnas:

```
datos::vuelos %>%
  print(n = 10, width = Inf)
```

Copy

También puedes controlar las características de impresión, modificando las opciones que están determinadas por default.

- `options(tibble.print_max = n, tibble.print_min = m)`: si hay más de `n` filas, mostrar solo `m` filas.
- Usa `options(tibble.print_min = Inf)` para mostrar siempre todas las filas.
- Usa `options(tibble.width = Inf)` para mostrar siempre todas las columnas sin importar el ancho de la pantalla.

Puedes ver una lista completa de opciones en la ayuda del paquete con `package?tibble`.

La opción final es usar el visualizador de datos de RStudio para obtener una versión interactiva del set de datos completo. Esto también es útil luego de realizar una larga cadena de manipulaciones.

```
datos::vuelos %>%
  View()
```

Copy

10.3.2 **Selección de subconjuntos**

Hasta ahora, todas las herramientas que aprendiste funcionan con el data frame completo. Si quieres recuperar una variable individual, necesitas algunas herramientas nuevas: `$` y `[[`. Mientras que `[[` permite extraer variables usando tanto su nombre como su posición, con `$` sólo se puede extraer mediante el nombre. La única diferencia es que `$` implica escribir un poco menos.

```r
df <- tibble(
  x = runif(5),
  y = rnorm(5)
)

# Extraer usando el nombre
df$x
#> [1] 0.73296674 0.23436542 0.66035540 0.03285612 0.46049161
df[["x"]]
#> [1] 0.73296674 0.23436542 0.66035540 0.03285612 0.46049161

# Extraer indicando la posición
df[[1]]
#> [1] 0.73296674 0.23436542 0.66035540 0.03285612 0.46049161
```

Para usar estas herramientas dentro de un pipe, necesitarás usar el marcador de posición `.` :

```r
df %>% .$x
#> [1] 0.73296674 0.23436542 0.66035540 0.03285612 0.46049161
df %>% .[["x"]]
#> [1] 0.73296674 0.23436542 0.66035540 0.03285612 0.46049161
```

En comparación a un `data.frame` , los tibbles son más estrictos: nunca funcionan con coincidencias parciales y generan una advertencia si la columna a la que intentas de acceder no existe.

10.4 Interactuando con código antiguo

Algunas funciones más antiguas no funcionan con los tibbles. Si te encuentras en uno de esos casos, usa `as.data.frame()` para convertir un tibble de nuevo en un `data.frame` :

```r
class(as.data.frame(tb))
#> [1] "data.frame"
```

La principal razón de que algunas funciones previas no funcionen con tibbles es la función `[` . En este libro no usamos mucho `[` porque `dplyr::filter()` y `dplyr::select()` resuelven los mismos problemas con un código más claro (aprenderás un poco sobre ello en el capítulo sobre subjoncuntos de vectores). Con los data frames de R base, `[` a veces devuelve un data frame y a veces devuelve un vector. Con tibbles, `[` siempre devuelve otro tibble.

10.5 Ejercicios

1. ¿Cómo puedes saber si un objeto es un tibble? (Sugerencia: imprime `mtautos` en consola, que es un data frame clásico).

2. Compara y contrasta las siguientes operaciones aplicadas a un `data.frame` y a un tibble equivalente. ¿Qué es diferente? ¿Por qué podría causarte problemas el comportamiento por defecto del data frame?

```r
df <- data.frame(abc = 1, xyz = "a")
df$x
df[, "xyz"]
df[, c("abc", "xyz")]
```

3. Si tienes el nombre de una variable guardada en un objeto, p.e., `var <- "mpg"` , ¿cómo puedes extraer esta variable de un tibble?

4. Practica referenciar nombres no sintácticos en el siguiente data frame:

    1. Extrayendo la variable llamada `1` .

    2. Generando un gráfico de dispersión de `1` vs `2` .

    3. Creando una nueva columna llamada `3` que sea el resultado de la división de `2` por `1` .

4. Renombrando las columnas como `uno`, `dos` y `tres`.

```
molesto <- tibble(
  `1` = 1:10,
  `2` = `1` * 2 + rnorm(length(`1`))
)
```

Copy

5. ¿Qué hace `tibble::enframe()`? ¿Cuándo lo usarías?

6. ¿Qué opción controla cuántos nombres de columnas adicionales se muestran al pie de un tibble?

« 9 Introducción

11 Importación de datos »

11 Importación de datos

11.1 Introducción

Trabajar con datos incluidos en paquetes de R es una muy buena forma de empezar a conocer las herramientas de la ciencia de datos. Sin embargo, en algún punto deberás parar de aprender y comenzar a trabajar con tus propios datos. En este capítulo aprenderás cómo leer en R archivos rectangulares de texto plano. Si bien solo tocaremos superficialmente el tema de importación, muchos de los principios que veremos son aplicables al trabajo con otras formas de datos. Finalizaremos sugiriendo algunos paquetes que son útiles para otros formatos.

11.1.1 Prerrequisitos

En este capítulo aprenderás cómo cargar archivos planos en R con **readr**, uno de los paquetes principales de **tidyverse**.

```
library(tidyverse)
```

Copy

11.2 Comenzando

La mayoría de las funciones de **readr** se enfocan en transformar archivos planos en *data frames*:

- `read_csv()` lee archivos delimitados por coma, `read_csv2()` lee archivos separados por punto y coma (comunes en países donde ',' es utilizada para separar decimales), `read_tsv()` lee archivos delimitados por tabulaciones y `read_delim()` archivos con cualquier delimitador.
- `read_fwf()` lee archivos de ancho fijo. Puedes especificar los campos ya sea por su ancho, con `fwf_widths()`, o por su ubicación, con `fwf_positions()`. `read_table()` lee una variación común de estos archivos de ancho fijo en los que las columnas se encuentran separadas por espacios.
- `read_log()` lee archivos de registro estilo Apache. (Revisa también webreadr, que está construido sobre `read_log()` y proporciona muchas otras herramientas útiles).

Todas estas funciones tienen una sintaxis similar, por lo que una vez que dominas una, puedes utilizar todas las demás con facilidad. En el resto del capítulo nos enfocaremos en `read_csv()`. Los archivos csv no solo son una de las formas de almacenamiento más comunes, sino que una vez que comprendas `read_csv()` podrás aplicar fácilmente tus conocimientos a todas las otras funciones de **readr**.

El primer argumento de `read_csv()` es el más importante: es la ruta al archivo a leer.

```
alturas <- read_csv("data/alturas.csv")
#>
#> ── Column specification ────────────────────────────────────────────────────────
#> cols(
#>   earn = col_double(),
#>   height = col_double(),
#>   sex = col_character(),
#>   ed = col_double(),
#>   age = col_double(),
#>   race = col_character()
#> )
```

Copy

Cuando ejecutas `read_csv()`, la función devuelve el nombre y tipo de datos con que se importó cada columna. Esta es una parte importante de **readr**, sobre la cual volveremos luego en segmentar un archivo.

Puedes también definir un archivo CSV "en línea" (*inline*). Esto es útil para experimentar con **readr** y para crear ejemplos reproducibles para ser compartidos.

On this page

View source
Edit this page

```
read_csv("a,b,c
1,2,3
4,5,6")
#> # A tibble: 2 x 3
#>       a     b     c
#>   <dbl> <dbl> <dbl>
#> 1     1     2     3
#> 2     4     5     6
```

Copy

En ambos casos `read_csv()` emplea la primera línea de los datos para los nombres de columna, lo que es una convención muy común. Hay dos casos en los que podrías querer ajustar este comportamiento:

1. A veces hay unas pocas líneas de metadatos al comienzo del archivo. Puedes usar `skip = n` para omitir las primeras `n` líneas, o bien, o usar `comment = "#"` para quitar todas las líneas que comienzan con, por ejemplo, `#`.

   ```
   read_csv("La primera línea de metadata
     La segunda línea de metadata
     x,y,z
     1,2,3", skip = 2)
   #> # A tibble: 1 x 3
   #>       x     y     z
   #>   <dbl> <dbl> <dbl>
   #> 1     1     2     3

   read_csv("# Un comentario que quiero ignorar
     x,y,z
     1,2,3", comment = "#")
   #> # A tibble: 1 x 3
   #>       x     y     z
   #>   <dbl> <dbl> <dbl>
   #> 1     1     2     3
   ```

   Copy

2. Los datos pueden no tener nombres de columna. En ese caso, puedes utilizar `col_names = FALSE` para decirle a `read_csv()` que no trate la primera fila como encabezados y que, en lugar de eso, los etiquete secuencialmente desde `X1` a `Xn`:

   ```
   read_csv("1,2,3\n4,5,6", col_names = FALSE)
   #> # A tibble: 2 x 3
   #>      X1    X2    X3
   #>   <dbl> <dbl> <dbl>
   #> 1     1     2     3
   #> 2     4     5     6
   ```

   Copy

   (`"\n"` es un atajo conveniente para agregar una línea nueva. Aprenderás más acerca de él y otros modos de evitar texto en la sección Cadenas: elementos básicos).

   Alternativamente puedes utilizar `col_names` con el vector de caracteres que será utilizado como nombres de columna:

   ```
   read_csv("1,2,3\n4,5,6", col_names = c("x", "y", "z"))
   #> # A tibble: 2 x 3
   #>       x     y     z
   #>   <dbl> <dbl> <dbl>
   #> 1     1     2     3
   #> 2     4     5     6
   ```

   Copy

Otra opción que comúnmente necesita ajustes es `na` (del inglés, *"not available"*: Esto especifica el valor (o valores) que se utilizan para representar los valores faltantes en tu archivo:

```
read_csv("a,b,c\n1,2,.", na = ".")
#> # A tibble: 1 x 3
#>       a     b c
#>   <dbl> <dbl> <lgl>
#> 1     1     2 NA
```

Copy

Esto es todo lo que necesitas saber para leer el ~75% de los archivos csv con los que te encontrarás en la práctica. También puedes adaptar fácilmente lo que has aprendido para leer archivos separados por tabuladores con `read_tsv()` y archivos de ancho fijo con `read_fwf()`. Para leer archivos más desafiantes, necesitas aprender un poco más sobre cómo **readr** segmenta cada columna y las transforma en vectores de R.

11.2.1 Comparación con R base

Si has utilizado R anteriormente, tal vez te preguntas por qué no usamos `read.csv()`. Hay unas pocas buenas razones para preferir las funciones de **readr** sobre las equivalentes de R base:

- Generalmente son mucho más rápidas (~10x) que sus equivalentes. Los trabajos que tienen un tiempo de ejecución prolongado poseen una barra de progreso para que puedas ver qué está ocurriendo. Si solo te interesa la velocidad, prueba `data.table::fread()`. No se ajusta tan bien con el **tidyverse**, pero puede ser bastante más rápido.
- Producen tibbles, no convierten los vectores de caracteres a factores, no usan nombres de filas ni distorsionan los nombres de columnas. Estas son fuentes comunes de frustración al utilizar las funciones de R base.
- Son más reproducibles. Las funciones de R base heredan ciertos comportamientos de tu sistema operativo y de las variables del ambiente, de modo que importar código que funciona bien en tu computadora puede no funcionar en la de otros.

11.2.2 Ejercicios

1. ¿Qué función utilizarías para leer un archivo donde los campos están separados con “|”?
2. Además de `file`, `skip` y `comment`, ¿qué otros argumentos tienen en común `read_csv()` y `read_tsv()`?
3. ¿Cuáles son los argumentos más importantes de `read_fwf()`?
4. Algunas veces las cadenas de caracteres en un archivo csv contienen comas. Para evitar que causen problemas, deben estar rodeadas por comillas, como `"` o `'`. Por convención, `read_csv()` asume que el caracter de separación será `"`.¿Qué argumentos debes especificar para leer el siguiente texto en un *data frame*?

```
"x,y\n1,'a,b'"
```

Copy

5. Identifica qué está mal en cada una de los siguientes archivos csv en línea (*inline*). ¿Qué pasa cuando corres el código?

```
read_csv("a,b\n1,2,3\n4,5,6")
read_csv("a,b,c\n1,2\n1,2,3,4")
read_csv("a,b\n\"1")
read_csv("a,b\n1,2\na,b")
read_csv("a;b\n1;3")
```

Copy

11.3 Segmentar un vector

Antes de entrar en detalles sobre cómo **readr** lee archivos del disco, necesitamos desviarnos un poco para hablar sobre las funciones `parse_*()` (del inglés *analizar*, *segmentar*). Estas funciones toman un vector de caracteres y devuelven un vector más especializado, como un vector lógico, numérico o una fecha:

```
str(parse_logical(c("TRUE", "FALSE", "NA")))
#>  logi [1:3] TRUE FALSE NA
str(parse_integer(c("1", "2", "3")))
#>  int [1:3] 1 2 3
str(parse_date(c("2010-01-01", "1979-10-14")))
#>  Date[1:2], format: "2010-01-01" "1979-10-14"
```

Copy

Estas funciones son útiles por sí mismas, pero también son un bloque estructural importante para **readr**. Una vez que aprendas en esta sección cómo funcionan los segmentadores individuales, en la próxima volveremos atrás y veremos cómo se combinan entre ellos para analizar un archivo completo.

Como todas las funciones dentro del **tidyverse**, las funciones `parse_*()` son uniformes: el primer argumento es un vector de caracteres a analizar y el argumento `na` especifica qué cadenas deberían ser tratadas como faltantes:

```
parse_integer(c("1", "231", ".", "456"), na = ".")
#> [1]   1 231  NA 456
```

Copy

Si la segmentación falla, obtendrás una advertencia:

```
x <- parse_integer(c("123", "345", "abc", "123.45"))
#> Warning: 2 parsing failures.
#> row col               expected actual
#>   3  -- an integer             abc
#>   4  -- no trailing characters 123.45
```

Copy

Y las fallas aparecerán como faltantes en el output:

```
x
#> [1] 123 345  NA  NA
#> attr(,"problems")
#> # A tibble: 2 x 4
#>     row   col expected               actual
#>   <int> <int> <chr>                  <chr>
#> 1     3    NA an integer             abc
#> 2     4    NA no trailing characters 123.45
```

Copy

Si hay muchas fallas de segmentación, necesitarás utilizar `problems()` (del inglés *problemas*) para obtener la totalidad de ellas. Esto devuelve un tibble que puedes luego manipular con **dplyr**.

```
problems(x)
#> # A tibble: 2 x 4
#>     row   col expected               actual
#>   <int> <int> <chr>                  <chr>
#> 1     3    NA an integer             abc
#> 2     4    NA no trailing characters 123.45
```

Copy

Utilizar segmentadores es más que nada una cuestión de entender qué está disponible y cómo enfrentar diferentes tipos de input. Hay ocho segmentadores particularmente importantes:

1. `parse_logical()` y `parse_integer()` analizan valores lógicos y números enteros respectivamente. No hay prácticamente nada que pueda salir mal con estos segmentadores, así que no los describiremos con detalle aquí.

2. `parse_double()` es un segmentador numérico estricto, y `parse_number()` es un segmentador numérico flexible. Son más complicados de lo que podrías esperar debido a que los números se escriben de diferentes formas en distintas partes del mundo.

3. `parse_character()` parece tan simple que no debiera ser necesario. Pero una complicación lo hace bastante importante: la codificación de caracteres (el *encoding*).

4. `parse_factor()` crea factores, la estructura de datos que R usa para representar variables categóricas con valores fijos y conocidos.

5. `parse_datetime()`, `parse_date()` y `parse_time()` te permiten analizar diversas especificaciones de fechas y horas. Estos son los más complicados, ya que hay muchas formas diferentes de escribir las fechas.

Las secciones siguientes describen estos analizadores en mayor detalle.

11.3.1 **Números**

Pareciera que analizar un número debiese ser algo sencillo, pero hay tres problemas que pueden complicar el proceso:

1. Las personas escriben los números de forma distinta en diferentes partes del mundo. Por ejemplo, algunos países utilizan `.` entre el entero y la fracción de un número real, mientras que otros utilizan `,`.

2. A menudo los números están rodeados por otros caracteres que proporcionan algún contexto, como "$1000" o "10%".

3. Los números frecuentemente contienen caracteres de "agrupación" para hacerlos más fáciles de leer, como "1,000,000". Estos caracteres de agrupación varían alrededor del mundo.

Para enfrentar al primer problema, **readr** tiene el concepto de "*locale*", un objeto que especifica las opciones de segmentación que difieren de un lugar a otro. Cuando segmentamos números, la opción más importante es el caracter que utilizas como símbolo decimal. Puedes sobreescribir el valor por defecto `.` creando un nuevo locale y estableciendo el argumento `decimal_mark` (del inglés *marca decimal*):

```
parse_double("1.23")
#> [1] 1.23
parse_double("1,23", locale = locale(decimal_mark = ","))
#> [1] 1.23
```

El locale por defecto de **readr** es EEUU-céntrico, porque generalmente R es EEUU-céntrico (por ejemplo, la documentación de R base está escrita en inglés norteamericano). Una aproximación alternativa podría ser probar y adivinar las opciones por defecto de tu sistema operativo. Esto es difícil de hacer y, lo que es más importante, hace que tu código sea frágil. Incluso si funciona en tu computadora, puede fallar cuando lo envíes a un/a colega en otro país.

`parse_number()` responde al segundo problema: ignora los caracteres no-numéricos antes y después del número. Esto es particularmente útil para monedas y porcentajes, pero también sirve para extraer números insertos en texto.

```
parse_number("$100")
#> [1] 100
parse_number("20%")
#> [1] 20
parse_number("It cost $123.45")
#> [1] 123.45
```

El problema final se puede enfrentar combinando `parse_number()` y el locale, ya que `parse_number()` ignorará el "símbolo decimal":

```
# Utilizado en América
parse_number("$123,456,789")
#> [1] 123456789
# Utilizado en muchas regiones de Europa
parse_number("123.456.789", locale = locale(grouping_mark = "."))
#> [1] 123456789
# Utilizado en Suiza
parse_number("123'456'789", locale = locale(grouping_mark = "'"))
#> [1] 123456789
```

11.3.2 **Cadenas de texto (*strings*)**

En apariencia, `parse_character()` debería ser realmente simple — podría tan solo devolver su input. Desafortunadamente, la vida no es tan simple, dado que existen múltiples formas de representar la misma cadena de texto. Para entender qué está pasando, necesitamos profundizar en los detalles de cómo las

computadoras representan las cadenas de texto. En R, podemos acceder a su representación subyacente empleando `charToRaw()`:

```
charToRaw("Hadley")
#> [1] 48 61 64 6c 65 79
```

Copy

Cada número hexadecimal representa un byte de información: `48` es *H*, `61` es *a*, y así. El mapeo desde un número hexadecimal a caracteres se denomina codificación o *encoding* y, en este caso, la codificación utilizada se llama ASCII. ASCII hace un muy buen trabajo representando caracteres del inglés, ya que es el ***American*** *Standard Code for Information Interchange* (del inglés *Código Americano estandarizado para el intercambio de información*).

Las cosas se complican un poco más para lenguas distintas al inglés. En los comienzos de la computación existían muchos estándares de codificación para caracteres no-ingleses compitiendo. Para poder interpretar correctamente una cadena de texto se necesita conocer tanto los valores como la codificación. Por ejemplo, dos codificaciones comunes son Latin1 (conocida también como ISO-8859-1 y utilizada para las lenguas del oeste de Europa) y Latin2 (o ISO-8859-2, utilizada para las lenguas de Europa del este). En Latin1, el byte 'b1' es "Â±", pero en Latin2, ¡es "ą"! Afortunadamente, en la actualidad hay un estándar que tiene soporte casi en todos lados: UTF-8. UTF-8 puede codificar casi cualquier caracter utilizado por humanos, así como muchos símbolos adicionales (¡como los emoji!).

readr utiliza UTF-8 en todas partes: asume que tus datos están codificados en UTF-8 cuando los lee y lo emplea siempre cuando los escribe. Esta es una buena opción por defecto, pero fallará con datos producidos por sistemas más viejos que no entienden UTF-8. Si te sucede esto, tus cadenas de texto se verán extrañas cuando las imprimas en la consola. Algunas veces solo uno o dos caracteres estarán errados. Otras veces obtendrás un total jeroglífico. Por ejemplo:

```
x1 <- "El Ni\xf1o was particularly bad this year"
x2 <- "\x82\xb1\x82\xf1\x82\xc9\x82\xbf\x82\xcd"
x1
#> [1] "El Ni\xf1o was particularly bad this year"
x2
#> [1] "\x82\xb1\x82\xf1\x82\u0242\xbf\x82\xcd"
```

Copy

Para corregir el problema necesitas especificar la codificación en `parse_character()`:

```
parse_character(x1, locale = locale(encoding = "Latin1"))
#> [1] "El Niño was particularly bad this year"
parse_character(x2, locale = locale(encoding = "Shift-JIS"))
#> [1] "こんにちは"
```

Copy

¿Cómo encontrar la codificación correcta? Si tienes suerte, estará incluida en alguna parte de la documentación de los datos. Desafortunadamente raras veces es ese el caso, así que **readr** provee la función `guess_encoding()` para ayudarte a adivinarla. No es a prueba de tontos y funciona mejor cuando tienes mucho texto (a diferencia de aquí), pero es un punto de inicio razonable. Es esperable hacer varias pruebas con diferentes codificaciones antes de encontrar la correcta.

```
guess_encoding(charToRaw(x1))
#> # A tibble: 2 x 2
#>   encoding   confidence
#>   <chr>           <dbl>
#> 1 ISO-8859-1       0.46
#> 2 ISO-8859-9       0.23
guess_encoding(charToRaw(x2))
#> # A tibble: 1 x 2
#>   encoding confidence
#>   <chr>         <dbl>
#> 1 KOI8-R         0.42
```

Copy

El primer argumento para `guess_encoding()` puede ser la ruta a un archivo o, como en este caso, un vector en bruto (útil si el texto ya se encuentra en R). Las codificaciones son un tema rico y complejo y solo te hemos mostrado la superficie acá. Si quieres aprender más al respecto, te recomendamos que leas la explicación detallada en http://kunststube.net/encoding/.

11.3.3 Factores

R utiliza factores para representar las variables categóricas que tienen un conjunto conocido de valores posibles. Puedes darle a `parse_factor()` un vector de niveles conocidos (`levels`) para generar una advertencia cada vez que haya un valor inesperado:

```
fruta <- c("manzana", "banana")
parse_factor(c("manzana", "banana", "bananana"), levels = fruta)
#> Warning: 1 parsing failure.
#> row col           expected   actual
#>   3  -- value in level set bananana
#> [1] manzana banana  <NA>
#> attr(,"problems")
#> # A tibble: 1 x 4
#>     row   col expected           actual
#>   <int> <int> <chr>              <chr>
#> 1     3    NA value in level set bananana
#> Levels: manzana banana
```

Copy

Si tienes muchas entradas problemáticas, a menudo es más fácil dejarlas como vectores de caracteres y luego utilizar las herramientas sobre las que aprenderás en los capítulos Cadenas de caracteres y Factores para limpiarlas.

11.3.4 Fechas, fechas-horas, y horas

Debes elegir entre tres segmentadores dependiendo de si quieres una fecha (el número de los días desde el 01-01-1970), una fecha-hora (el número de segundos desde la medianoche del 01-01-1970) o una hora (el número de segundos desde la medianoche). Cuando se llaman sin argumentos adicionales:

- `parse_datetime()` asume una fecha-hora ISO8601. ISO8601 es un estándar internacional en el que los componentes de una fecha están organizados de mayor a menor: año, mes, día, hora, minuto, segundo.

```
parse_datetime("2010-10-01T2010")
#> [1] "2010-10-01 20:10:00 UTC"
# Si se omite la hora, será determinada como medianoche.
parse_datetime("20101010")
#> [1] "2010-10-10 UTC"
```

Copy

Esta es la estandarización de fecha/hora más importante. Si trabajas con fechas y horas frecuentemente, te recomendamos que leas https://en.wikipedia.org/wiki/ISO_8601

- `parse_date()` asume un año de cuatro dígitos, un guión `-` o `/`, el mes, un guión `-` o `/` y luego el día.

```
parse_date("2010-10-01")
#> [1] "2010-10-01"
```

Copy

- `parse_time()` espera la hora, `:`, minutos, opcionalmente `:` y segundos, y un especificador opcional am/pm:

```
library(hms)
parse_time("01:10 am")
#> 01:10:00
parse_time("20:10:01")
#> 20:10:01
```

Copy

  R base no tiene incorporada una muy buena clase para datos temporales, por lo que usamos la provista en el paquete **hms**.

Si esos valores por defecto no funcionan con tus datos, puedes proporcionar tu propio formato fecha-hora construido con las siguientes piezas:

Año

`%Y` (4 dígitos).

`%y` (2 dígitos); 00-69 -> 2000-2069, 70-99 -> 1970-1999.

Mes

`%m` (2 dígitos).

`%b` (nombre abreviado, como “ene”).

`%B` (nombre completo, “enero”).

Día

`%d` (2 dígitos).

`%e` (espacio opcional destacado).

Hora

`%H` 0-23 horas.

`%I` 0-12, debe utilizarse con `%p` .

`%p` indicador AM/PM.

`%M` minutos.

`%S` segundos enteros.

`%OS` segundos reales.

`%Z` Zona horaria (como nombre, por ejemplo, `America/Chicago`). Advertencia sobre abreviaturas: si eres de EEUU, ten en cuenta que “EST” es una zona horaria canadiense que no tiene cambios de horario.¡**No** es la hora Estandar del Este! Retomaremos esto más adelante en la sección Husos horarios.

`%z` (como complemento para las UTC, por ejemplo, `+0800`).

No-dígitos

`%.` se salta un caracter no-dígito.

`%*` se salta cualquier número de caracteres no-dígitos.

La mejor manera de deducir el formato correcto es crear unos pocos ejemplos en un vector de caracteres y probarlos con una de las funciones de segmentación. Por ejemplo:

```
parse_date("01/02/15", "%m/%d/%y")
#> [1] "2015-01-02"
parse_date("01/02/15", "%d/%m/%y")
#> [1] "2015-02-01"
parse_date("01/02/15", "%y/%m/%d")
#> [1] "2001-02-15"
```

Si estás utilizando `%b` o `%B` con nombres de meses no ingleses, necesitarás ajustar el argumento `lang` para `locale()` . Mira la lista de lenguas incorporados en `date_names_langs()` . Si tu lengua no está incluida, puedes crearla con `date_names()` .

```
parse_date("1 janvier 2015", "%d %B %Y", locale = locale("fr"))
#> [1] "2015-01-01"
```

11.3.5 Ejercicios

1. ¿Cuáles son los argumentos más importantes para `locale()` ?

2. ¿Qué pasa si intentas establecer `decimal_mark` y `grouping_mark` como el mismo caracter? ¿Qué pasa con el valor por defecto de `grouping_mark` cuando estableces `decimal_mark` como `,` ? ¿Qué pasa con el valor por defecto de `decimal_mark` cuando estableces `grouping_mark` como `.` ?

3. No discutimos las opciones de `date_format` y `time_format` para `locale()`. ¿Qué hacen? Construye un ejemplo que muestre cuándo podrían ser útiles.

4. Si vives fuera de EEUU, crea un nuevo objeto locale que contenga las opciones para los tipos de archivo que lees más comúnmente.

5. ¿Cuál es la diferencia entre `read_csv()` y `read_csv2()`?

6. ¿Cuáles son las codificaciones más comunes empleadas en Europa? ¿Cuáles son las codificaciones más comunes utilizadas en Asia? ¿Y en América Latina? Googlea un poco para descubrirlo.

7. Genera el formato correcto de texto para segmentar cada una de las siguientes fechas y horas:

```
d1 <- "Enero 1, 2010"
d2 <- "2015-Ene-07"
d3 <- "06-Jun-2017"
d4 <- c("Augosto 19 (2015)", "Julio 1 (2015)")
d5 <- "12/30/14" # Dec 30, 2014
t1 <- "1705"
t2 <- "11:15:10.12 PM"
```

Copy

11.4 Segmentar un archivo

Ahora que aprendiste cómo analizar un vector individual, es tiempo de volver al comienzo y explorar cómo **readr** analiza un archivo. Hay dos cosas nuevas que aprenderás al respecto en esta sección:

1. Cómo **readr** deduce automáticamente el tipo de cada columna.
2. Cómo sobreescribir las especificaciones por defecto.

11.4.1 Estrategia

readr utiliza una heurística para deducir el tipo de cada columna: lee las primeras 1000 filas y utiliza una heurística (moderadamente conservadora) para deducir el formato de las columnas. Puedes simular este proceso con un vector de caracteres utilizando `guess_parser()`, que devuelve la mejor deducción de **readr**, y `parse_guess()` que utiliza esa deducción para analizar la columna:

```
guess_parser("2010-10-01")
#> [1] "date"
guess_parser("15:01")
#> [1] "time"
guess_parser(c("TRUE", "FALSE"))
#> [1] "logical"
guess_parser(c("1", "5", "9"))
#> [1] "double"
guess_parser(c("12,352,561"))
#> [1] "number"
str(parse_guess("2010-10-10"))
#>  Date[1:1], format: "2010-10-10"
```

Copy

La heurística prueba cada uno de los siguientes tipos y se detiene cuando encuentra una coincidencia:

- lógico: contiene solo “F”, “T”, “FALSE”, o “TRUE”.
- entero: contiene solo caracteres numéricos (y ‘-’).
- doble: contiene solo dobles válidos (incluyendo números como ‘4.5e-5’).
- número: contiene dobles válidos con la marca de agrupamiento en su interior.
- hora: coincide con el formato horario por defecto (`time_format`).
- fecha: coincide con el formato fecha por defecto (`date_format`).
- fecha-hora: cualquier fecha ISO8601.

Si ninguna de esas reglas se aplica, entonces la columna quedará como un vector de cadenas de caracteres.

11.4.2 Problemas

Esos valores por defecto no siempre funcionan para archivos de gran tamaño. Hay dos problemas básicos:

1. Las primeras mil filas podrían ser un caso especial y **readr** estaría deduciendo un formato que no es suficientemente general. Por ejemplo, podrías tener una columna de dobles que solo contiene enteros en las primeras 1000 filas.
2. La columna podría contener muchos valores faltantes. Si las primeras 1000 filas contienen solo `NA`, **readr** deducirá que es un vector lógico, mientras que tú probablemente quieras analizarlo como algo más específico.

readr contiene un archivo csv desafiante que ilustra ambos problemas:

```
desafio <- read_csv(readr_example("challenge.csv"))
#>
#> ── Column specification ────────────────────────────────────────────────────────
#> cols(
#>   x = col_double(),
#>   y = col_logical()
#> )
#> Warning: 1000 parsing failures.
#>  row col           expected     actual
file
#> 1001   y 1/0/T/F/TRUE/FALSE 2015-01-16
'/Users/runner/work/_temp/Library/readr/extdata/challenge.csv'
#> 1002   y 1/0/T/F/TRUE/FALSE 2018-05-18
'/Users/runner/work/_temp/Library/readr/extdata/challenge.csv'
#> 1003   y 1/0/T/F/TRUE/FALSE 2015-09-05
'/Users/runner/work/_temp/Library/readr/extdata/challenge.csv'
#> 1004   y 1/0/T/F/TRUE/FALSE 2012-11-28
'/Users/runner/work/_temp/Library/readr/extdata/challenge.csv'
#> 1005   y 1/0/T/F/TRUE/FALSE 2020-01-13
'/Users/runner/work/_temp/Library/readr/extdata/challenge.csv'
#> .... ... .................. ..........
..........................................................
#> See problems(...) for more details.
```

(Fíjate en el uso de `readr_example()`, que encuentra la ruta a uno de los archivos incluidos en el paquete.)

Hay dos outputs impresos en la consola: la especificación de columna generada al mirar las primeras 1000 filas y las primeras cinco fallas de segmentación. Siempre es una buena idea extraer explícitamente los problemas con `problems()`, así puedes explorarlos en mayor profundidad:

```
problems(desafio)
#> # A tibble: 1,000 x 5
#>     row col   expected        actual   file
#>   <int> <chr> <chr>           <chr>    <chr>
#> 1  1001 y     1/0/T/F/TRUE/F… 2015-01… '/Users/runner/work/_temp/Library/readr/…
#> 2  1002 y     1/0/T/F/TRUE/F… 2018-05… '/Users/runner/work/_temp/Library/readr/…
#> 3  1003 y     1/0/T/F/TRUE/F… 2015-09… '/Users/runner/work/_temp/Library/readr/…
#> 4  1004 y     1/0/T/F/TRUE/F… 2012-11… '/Users/runner/work/_temp/Library/readr/…
#> 5  1005 y     1/0/T/F/TRUE/F… 2020-01… '/Users/runner/work/_temp/Library/readr/…
#> 6  1006 y     1/0/T/F/TRUE/F… 2016-04… '/Users/runner/work/_temp/Library/readr/…
#> # … with 994 more rows
```

Una buena estrategia es trabajar columna por columna hasta que no queden problemas. Aquí podemos ver que hubo muchos problemas de análisis con la columna `y`. Si miramos la últimas líneas, verás que hay fechas almacenadas en un vector de caracteres.

```
tail(desafio)
#> # A tibble: 6 x 2
#>       x y
#>   <dbl> <lgl>
#> 1 0.805 NA
#> 2 0.164 NA
#> 3 0.472 NA
#> 4 0.718 NA
#> 5 0.270 NA
#> 6 0.608 NA
```

Esto sugiere que mejor sería utilizar un segmentador de fechas. Para arreglar este problema, copia y pega la especificación de las columnas que habías obtenido inicialmente y agrégalas a tu código:

```
desafio <- read_csv(
  readr_example("challenge.csv"),
  col_types = cols(
    x = col_double(),
    y = col_logical()
  )
)
```

Y luego ajusta el tipo de la columna `y` especificando que se trata de una fecha:

```
desafio <- read_csv(
  readr_example("challenge.csv"),
  col_types = cols(
    x = col_double(),
    y = col_date()
  )
)
tail(desafio)
#> # A tibble: 6 x 2
#>       x y
#>   <dbl> <date>
#> 1 0.805 2019-11-21
#> 2 0.164 2018-03-29
#> 3 0.472 2014-08-04
#> 4 0.718 2015-08-16
#> 5 0.270 2020-02-04
#> 6 0.608 2019-01-06
```

Cada función `parse_*()` tiene su correspondiente función `col_*()`. Se utiliza `parse_*()` cuando los datos se encuentran en un vector de caracteres que ya está disponible en R; `col_*` para cuando quieres decirle a **readr** cómo cargar los datos.

Te recomendamos proporcionar la estructura para `col_types` a partir de la impresión en consola provista por **readr**. Esto asegura que tienes un script para importar datos consistente y reproducible. Si confías en las deducciones por defecto y tus datos cambian, **readr** continuará leyéndolos. Si quieres ser realmente estricto/a, emplea `stop_for_problems()` (*detenerse en problemas*): esto devolverá un mensaje de error y detendrá tu script si hay cualquier problema con la segmentación.

11.4.3 Otras estrategias

Existen algunas estrategias generales más para ayudarte a segmentar archivos:

- En el ejemplo previo simplemente tuvimos mala suerta: si miramos solo una fila más que el número por defecto, podemos segmentar correctamente en un solo intento:

```
desafio2 <- read_csv(readr_example("challenge.csv"), guess_max = 1001)
#>
#> ── Column specification ────────────────────────────────────────────────────────
#> cols(
#>   x = col_double(),
#>   y = col_date(format = "")
#> )
desafio2
#> # A tibble: 2,000 x 2
#>       x y
#>   <dbl> <date>
#> 1   404 NA
#> 2  4172 NA
#> 3  3004 NA
#> 4   787 NA
#> 5    37 NA
#> 6  2332 NA
#> # … with 1,994 more rows
```

- Algunas veces es más fácil diagnosticar problemas si lees todas las columnas como vectores de caracteres:

```
desafio <- read_csv(readr_example("challenge.csv"),
  col_types = cols(.default = col_character())
)
```

Esto es particularmente útil en combinación con 'type_convert()', que aplica la heurística de segmentación a las columnas de caracteres en un data frame.

```
  df <- tribble(
    ~x,  ~y,
    "1", "1.21",
    "2", "2.32",
    "3", "4.56"
  )
  df
#> # A tibble: 3 x 2
#>   x     y
#>   <chr> <chr>
#> 1 1     1.21
#> 2 2     2.32
#> 3 3     4.56

# Fíjate en los tipos de columna
type_convert(df)
#>
#> ── Column specification ────────────────────────────────────────────────────────
#> cols(
#>   x = col_double(),
#>   y = col_double()
#> )
#> # A tibble: 3 x 2
#>       x     y
#>   <dbl> <dbl>
#> 1     1  1.21
#> 2     2  2.32
#> 3     3  4.56
```

- Si estás leyendo un archivo muy largo, podrías querer seleccionar `n_max` a un número pequeño como 10000 o 100000. Esto acelerará las iteraciones a la vez que eliminarás problemas comunes.

- Si tienes problemas de segmentación importantes, a veces es más fácil leer un vector de caracteres de líneas con `read_lines()`, o incluso un vector de caracteres de largo 1 con `read_file()`. Luego puedes utilizar las habilidades sobre segmentación de cadenas de caracteres que aprenderás más adelante para segmentar formatos más exóticos.

11.5 Escribir a un archivo

readr también incluye dos funciones muy útiles para escribir datos de vuelta al disco: `write_csv()` y `write_tsv()`. Ambas funciones incrementan las posibilidades de que el archivo resultante sea leído correctamente al:

- codificar siempre las cadenas de caracteres en UTF-8.
- guardar fechas y fechas-horas en formato ISO8601, por lo que son fácilmente segmentadas en cualquier sitio.

Si quieres exportar un archivo csv a Excel, utiliza `write_excel_csv()` —esto escribe un caracter especial (una marca de orden de bytes) al comienzo del archivo que le dice a Excel que estás utilizando codificación UTF-8. Los argumentos más importantes son `x` (el data frame a guardar) y `path` (la ubicación donde lo guardarás). También puedes especificar cómo se escriben los valores ausentes con `na` y si quieres `append` (agregarlo) a un archivo existente.

```
write_csv(desafio, "desafio.csv")
```

Copy

Fíjate que la información sobre el tipo de datos se pierde cuando guardas en csv:

```
desafio
#> # A tibble: 2,000 x 2
#>   x     y
#>   <chr> <chr>
#> 1 404   <NA>
#> 2 4172  <NA>
#> 3 3004  <NA>
#> 4 787   <NA>
#> 5 37    <NA>
#> 6 2332  <NA>
#> # … with 1,994 more rows
write_csv(desafio, "desafio-2.csv")
read_csv("desafio-2.csv")
#>
#> ── Column specification ────────────────────────────────────────────────────
#> cols(
#>   x = col_double(),
#>   y = col_logical()
#> )
#> # A tibble: 2,000 x 2
#>       x y
#>   <dbl> <lgl>
#> 1   404 NA
#> 2  4172 NA
#> 3  3004 NA
#> 4   787 NA
#> 5    37 NA
#> 6  2332 NA
#> # … with 1,994 more rows
```

Copy

Esto hace a los CSV poco confiables para almacenar en caché los resultados provisorios — necesitas recrear la especificación de las columnas cada vez que los cargas. Hay dos alternativas:

1. `write_rds()` and `read_rds()` son funciones "envoltorio" (*wrappers*) uniformes sobre las funciones base `readRDS()` y `saveRDS()`. Estas almacenan datos en un formato binario propio de R llamado RDS:

```
write_rds(desafio, "desafio.rds")
read_rds("desafio.rds")
#> # A tibble: 2,000 x 2
#>   x     y
#>   <chr> <chr>
#> 1 404   <NA>
#> 2 4172  <NA>
#> 3 3004  <NA>
#> 4 787   <NA>
#> 5 37    <NA>
#> 6 2332  <NA>
#> # … with 1,994 more rows
```

Copy

2. El paquete **feather** implementa un formato rápido de archivos binarios que puede compartirse a través de lenguajes de programación:

```
library(feather)
write_feather(desafio, "desafio.feather")
read_feather("desafio.feather")
#> # A tibble: 2,000 x 2
#>       x      y
#>   <dbl> <date>
#> 1   404   <NA>
#> 2  4172   <NA>
#> 3  3004   <NA>
#> 4   787   <NA>
#> 5    37   <NA>
#> 6  2332   <NA>
#> # ... with 1,994 more rows
```

Copy

Feather tiende a ser más rápido que RDS y es utilizable fuera de R. RDS permite columnas-listas (sobre las que aprenderás en el capítulo Muchos modelos), algo que **feather** no permite actualmente.

11.6 Otros tipos de datos

Para acceder a otros tipos de datos en R te recomendamos comenzar con los paquetes de **tidyverse** listados abajo. Ciertamente no son perfectos, pero son un buen lugar para comenzar. Para datos rectangulares:

- **haven** lee archivos SPSS, Stata y SAS.
- **readxl** lee archivos excel (tanto `.xls` como `.xlsx`).
- **DBI**, junto con un *backend* de base de datos específico (e.g. **RMySQL**, **RSQLite**, **RPostgreSQL**, etc.) te permite correr consultas SQL contra una base de datos y devolver un data frame.

Para datos jerárquicos: utiliza **jsonlite** (de Jeroen Ooms) para json y **xml2** para XML. Jenny Bryan tiene algunos ejemplos muy bien trabajados en https://jennybc.github.io/purrr-tutorial/.

Para otros tipos de archivos, prueba el manual de importación/exportación de datos de R y el paquete **rio**.

« 10 Tibbles

12 Datos ordenados »

12 Datos ordenados

12.1 Introducción

> “Todas las familias felices se parecen unas a otras, pero cada familia infeliz lo es a su manera.” — León Tolstoy

> “Todos los set de datos ordenados se parecen unos a otros, pero cada set de datos desordenado lo es a su manera” — Hadley Wickham

En este capítulo aprenderás una manera consistente para organizar tus datos en R a la que llamaremos **tidy data** (datos ordenados). Llevar tus datos a este formato requiere algo de trabajo previo; sin embargo, dicho trabajo tiene retorno positivo en el largo plazo. Una vez que tengas tus datos ordenados y las herramientas para ordenar datos que provee el tidyverse, vas a gastar mucho menos tiempo pasando de una forma de representar datos a otra, lo que te permitirá destinar más tiempo a las preguntas analíticas.

Este capítulo te dará una introducción práctica a los datos ordenados (o *tidy data*) y a las herramientas que provee el paquete **tidyr**. Si deseas aprender más acerca de la teoría subyacente, puede que te guste el artículo *Tidy Data* publicado en la revista Journal of Statistical Software, http://www.jstatsoft.org/v59/i10/paper.

12.1.1 Prerrequisitos

En este capítulo nos enfocaremos en **tidyr**, un paquete que provee un conjunto de herramientas que te ayudarán a ordenar datos desordenados. **tidyr** es parte del núcleo del tidyverse.

```
library(tidyverse)
library(datos)
```

Copy

12.2 Datos ordenados

Puedes representar los mismos datos subyacentes de múltiples formas. El ejemplo a continuación muestra los mismos datos organizados de cuatro maneras distintas. Cada dataset muestra los mismos valores de cuatro variables —*pais*, *anio*, *poblacion* y *casos*—, pero cada uno organiza los valores de forma distinta.

```
tabla1
#> # A tibble: 6 x 4
#>   pais        anio  casos  poblacion
#>   <chr>      <int>  <int>      <int>
#> 1 Afganistán  1999    745   19987071
#> 2 Afganistán  2000   2666   20595360
#> 3 Brasil      1999  37737  172006362
#> 4 Brasil      2000  80488  174504898
#> 5 China       1999 212258 1272915272
#> 6 China       2000 213766 1280428583
tabla2
#> # A tibble: 12 x 4
#>   pais        anio tipo         cuenta
#>   <chr>      <int> <chr>         <int>
#> 1 Afganistán  1999 casos           745
#> 2 Afganistán  1999 población  19987071
#> 3 Afganistán  2000 casos          2666
#> 4 Afganistán  2000 población  20595360
#> 5 Brasil      1999 casos         37737
#> 6 Brasil      1999 población 172006362
#> # … with 6 more rows
tabla3
#> # A tibble: 6 x 3
#>   pais        anio tasa
#>   <chr>      <int> <chr>
#> 1 Afganistán  1999 745/19987071
#> 2 Afganistán  2000 2666/20595360
#> 3 Brasil      1999 37737/172006362
#> 4 Brasil      2000 80488/174504898
#> 5 China       1999 212258/1272915272
#> 6 China       2000 213766/1280428583

# Dividido en dos tibbles
tabla4a # casos
#> # A tibble: 3 x 3
#>   pais       `1999` `2000`
#>   <chr>       <int>  <int>
#> 1 Afganistán    745   2666
#> 2 Brasil      37737  80488
#> 3 China      212258 213766
tabla4b # poblacion
#> # A tibble: 3 x 3
#>   pais           `1999`     `2000`
#>   <chr>           <int>      <int>
#> 1 Afganistán   19987071   20595360
#> 2 Brasil      172006362  174504898
#> 3 China      1272915272 1280428583
```

Las anteriores son representaciones de los mismos datos subyacentes, pero no todas son igualmente fáciles de usar. Un tipo de conjunto de datos, el conjunto de datos ordenado, será mucho más fácil de trabajar dentro del tidyverse.

Existen tres reglas interrelacionadas que hacen que un conjunto de datos sea ordenado:

1. Cada variable debe tener su propia columna.
2. Cada observación debe tener su propia fila.
3. Cada valor debe tener su propia celda.

La figura 12.1 muestra estas reglas visualmente.

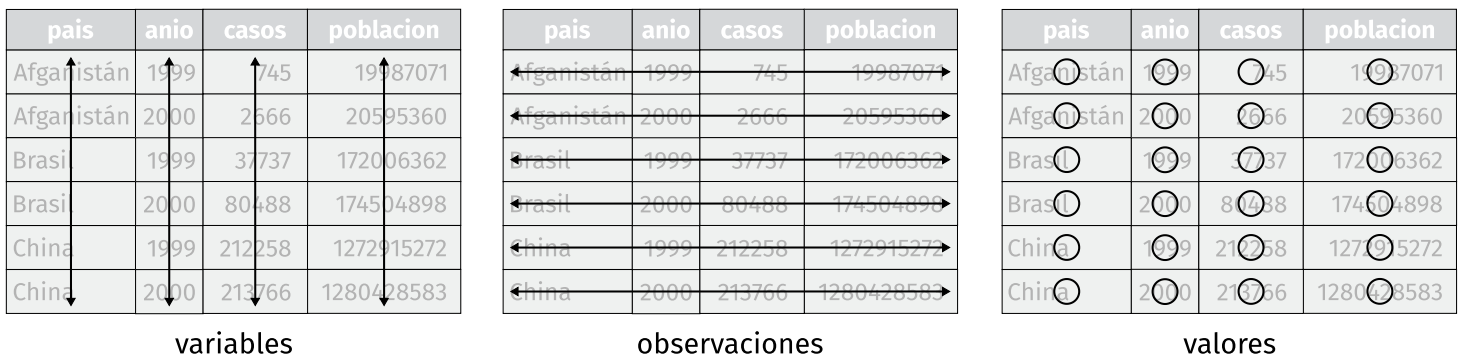


Figure 12.1: Reglas que hacen que un conjunto de datos sea ordenado: las variables están en columnas, las observaciones en filas y los valores en celdas.

Estas reglas están interrelacionadas ya que es imposible cumplir solo dos de las tres. Esta interrelación lleva a un conjunto práctico de instrucciones más simple aún:

1. Coloca cada conjunto de datos en un tibble.
2. Coloca cada variable en una columna.

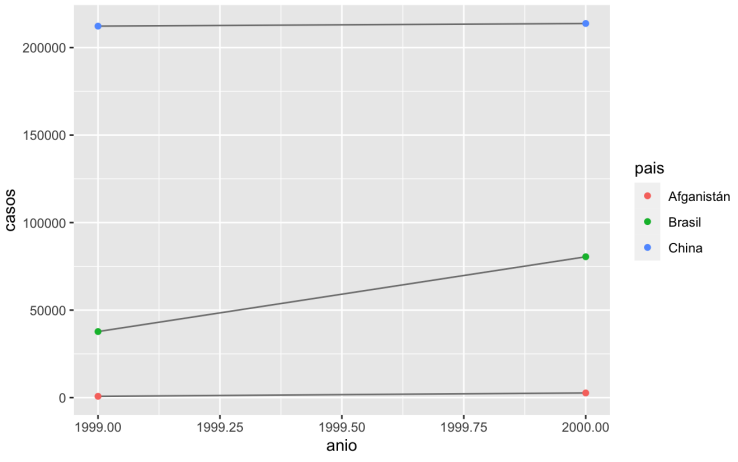
En este ejemplo, solo la `tabla1` está ordenada. Es la única representación en que cada columna es una variable.

¿Por qué asegurarse de que los datos estén ordenados? Existen dos ventajas principales:

1. Existe una ventaja general al elegir una forma consistente de almacenar datos. Si tienes una estructura de datos consistente, es más fácil aprender las herramientas que funcionan con ella ya que tienen una uniformidad subyacente.
2. Existe una ventaja específica al situar las variables en las columnas, ya que permite que la naturaleza vectorizada de R brille. Como habrás aprendido en las secciones sobre crear nuevas variables y resúmenes, muchas de las funciones que vienen con R trabajan con vectores de valores. Esto hace que transformar datos ordenados se perciba como algo casi natural.

dplyr, **ggplot2** y el resto de los paquetes del tidyverse están diseñados para trabajar con datos ordenados. Aquí hay algunos ejemplos de cómo podrías trabajar con `tabla1`.

```
# Calcular tasa por cada 10,000 habitantes
tabla1 %>%
  mutate(tasa = casos / poblacion * 10000)
#> # A tibble: 6 x 5
#>   pais        anio  casos  poblacion  tasa
#>   <chr>      <int>  <int>      <int> <dbl>
#> 1 Afganistán  1999    745   19987071 0.373
#> 2 Afganistán  2000   2666   20595360 1.29
#> 3 Brasil      1999  37737  172006362 2.19
#> 4 Brasil      2000  80488  174504898 4.61
#> 5 China       1999 212258 1272915272 1.67
#> 6 China       2000 213766 1280428583 1.67

# Calcular casos por anio
tabla1 %>%
  count(anio, wt = casos)
#> # A tibble: 2 x 2
#>    anio      n
#> * <int>  <int>
#> 1  1999 250740
#> 2  2000 296920

# Visualizar cambios en el tiempo
library(ggplot2)
ggplot(tabla1, aes(anio, casos)) +
  geom_line(aes(group = pais), colour = "grey50") +
  geom_point(aes(colour = pais))
```



12.2.1 **Ejercicios**

1. Usando prosa, describe cómo las variables y observaciones se organizan en las tablas de ejemplo.

2. Calcula la `tasa` para las tablas `tabla2` y `tabla4a + tabla4b`. Necesitarás las siguientes operaciones:

3. Extraer el número de casos de tuberculosis por país y año.

4. Extraer la población por país y año.

5. Dividir los casos por la población y multiplicarla por 10000.

6. Insertar los datos de vuelta en el lugar adecuado.

¿Cuál representación es más fácil de trabajar? ¿Cuál es la más difícil? ¿Por qué?

1. Recrea el gráfico que muestra el cambio en el número de casos usando la `tabla2` en lugar de la `tabla1`. ¿Qué necesitas hacer en primero?

12.3 **Pivotar**

Los principios sobre datos ordenados parecen tan obvios que te podrías preguntar si alguna vez encontrarás un dataset que no esté ordenado. Desafortunadamente, gran parte de los datos que vas a encontrar están desordenados. Existen dos principales razones para esto:

1. La mayoría de las personas no están familiarizadas con los principios de datos ordenados y es difícil derivarlos por cuenta propia a menos que pases *mucho* tiempo trabajando con datos.

2. Los datos a menudo están organizados para facilitar tareas distintas del análisis. Por ejemplo, los datos se organizan para que su registro sea lo más sencillo posible.

Esto significa que para la mayoría de los análisis necesitarás hacer algún tipo de orden. El primer paso es entender siempre cuáles son las variables y las observaciones. Esto a veces es fácil; otras veces deberás consultar con quienes crearon el dataset. El segundo paso es resolver uno de los siguientes problemas frecuentes:

1. Una variable se extiende por varias columnas

2. Una observación está dispersa entre múltiples filas.

Típicamente, un set de datos tiene uno de estos problemas. Si contiene ambos ¡significa que tienes muy mala suerte! Para solucionar estos problemas necesitarás las dos funciones más importantes de tidyr: `gather()` (reunir) y `spread()` (esparcir/extender).

12.3.1 **Datos “largos”**

Un problema común es cuando en un dataset los nombres de las columnas no representan nombres de variables, sino que representan los *valores* de una variable. Tomando el caso de la `tabla4a` : los nombres de las columnas `1999` y `2000` representan los valores de la variable `año`, los valores en las columnas `1999` y `2000` representan valores de la variable `casos` y cada fila representa dos observaciones en lugar de una.

```
tabla4a
#> # A tibble: 3 x 3
#>   pais       `1999` `2000`
#>   <chr>       <int>  <int>
#> 1 Afganistán    745   2666
#> 2 Brasil      37737  80488
#> 3 China      212258 213766
```

Para ordenar un dataset como este necesitamos **pivotar** las columnas que no cumplen en un nuevo par de variables. Para describir dicha operación necesitamos tres parámetros:

- El conjunto de columnas cuyos nombres son valores y no variables. En este ejemplo son las columnas `1999` y `2000`.
- El nombre de la variable cuyos valores forman los nombres de las columnas. Llamaremos a esto `key` (clave) y en este caso corresponde a `anio`.
- El nombre de la variable cuyos valores están repartidos por las celdas. Llamaremos a esto `value` (valor) y en este caso corresponde al número de `casos`.

Con estos parámetros podemos utilizar la función `pivot_longer()` (*pivotar a lo largo*):

```
tabla4a %>%
  pivot_longer(cols = c(`1999`, `2000`), names_to = "anio", values_to = "casos")
#> # A tibble: 6 x 3
#>   pais       anio   casos
#>   <chr>      <chr>  <int>
#> 1 Afganistán 1999     745
#> 2 Afganistán 2000    2666
#> 3 Brasil     1999   37737
#> 4 Brasil     2000   80488
#> 5 China      1999  212258
#> 6 China      2000  213766
```

Las columnas a girar quedan seleccionadas siguiendo el estilo de notación de `dplyr::select()`. En este caso hay solo dos columnas, por lo que las listamos individualmente. Ten en consideración que “1999” y “2000” son nombres no-sintáxicos (debido a que no comienzan con una letra) por lo que los rodeamos con acentos graves (o_backticks_). Para refrescar tu memoria respecto de la selección de columnas, consulta la sección sobre select. Las variables `anio` y `casos` no existen todavpia en la `tabla4a`, por lo que tenemos que poner sus nombres entre comillas.

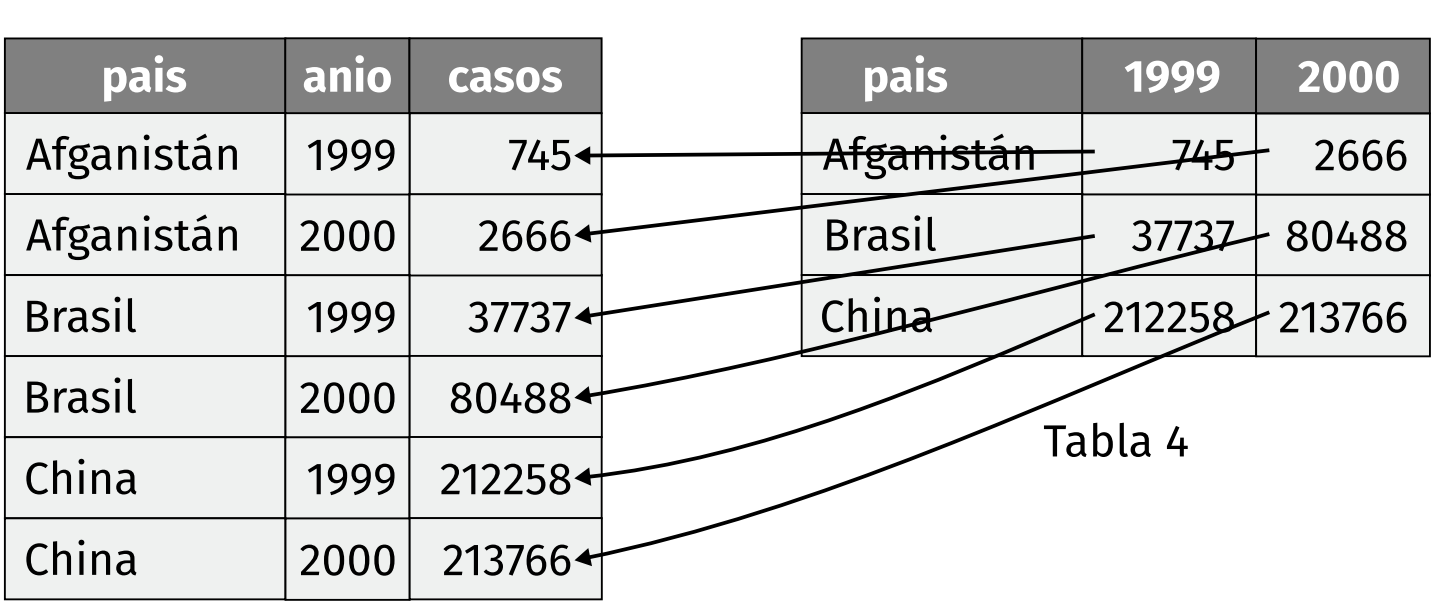


Figure 12.2: Pivotar la `tabla4` para un formato ‘largo’ y ordenado.

En el resultado final, las columnas pivotadas se eliminan y obtenemos la nuevas columnas `anio` y `casos`. La relación entre las variables originales se mantiene, tal como se puede observar en la Figura 12.2. Podemos usar `pivot_longer()` para ordenar `tabla4b` de modo similar. La única diferencia es la variable almacenada en los valores de las celdas:

```
tabla4b %>%
  pivot_longer(cols = c(`1999`, `2000`), names_to = "anio", values_to = "poblacion")
#> # A tibble: 6 x 3
#>   pais       anio   poblacion
#>   <chr>      <chr>      <int>
#> 1 Afganistán 1999    19987071
#> 2 Afganistán 2000    20595360
#> 3 Brasil     1999   172006362
#> 4 Brasil     2000   174504898
#> 5 China      1999  1272915272
#> 6 China      2000  1280428583
```

Para combinar las versiones ordenadas de `tabla4a` y `tabla4b` en un único tibble, necesitamos usar `dplyr::left_join()`, función que aprenderás en el capítulo sobre datos relacionales.

```
tidy4a <- tabla4a %>%
  pivot_longer(cols = c(`1999`, `2000`), names_to = "anio", values_to = "casos")

tidy4b <- tabla4b %>%
  pivot_longer(cols = c(`1999`, `2000`), names_to = "anio", values_to = "poblacion")

left_join(tidy4a, tidy4b)
#> Joining, by = c("pais", "anio")
#> # A tibble: 6 x 4
#>   pais       anio   casos  poblacion
#>   <chr>      <chr>  <int>      <int>
#> 1 Afganistán 1999     745   19987071
#> 2 Afganistán 2000    2666   20595360
#> 3 Brasil     1999   37737  172006362
#> 4 Brasil     2000   80488  174504898
#> 5 China      1999  212258 1272915272
#> 6 China      2000  213766 1280428583
```

12.3.2 Datos “anchos”

`pivot_wider()` (*pivotar a lo ancho*) es lo opuesto de `pivot_longer()`. Se usa cuando una observación aparece en múltiples filas. Por ejemplo, considera la `tabla2`: una observación es un país en un año, pero cada observación aparece en dos filas.

```
tabla2
#> # A tibble: 12 x 4
#>   pais        anio tipo         cuenta
#>   <chr>      <int> <chr>         <int>
#> 1 Afganistán  1999 casos           745
#> 2 Afganistán  1999 población  19987071
#> 3 Afganistán  2000 casos          2666
#> 4 Afganistán  2000 población  20595360
#> 5 Brasil      1999 casos         37737
#> 6 Brasil      1999 población 172006362
#> # … with 6 more rows
```

Para ordenar esto, primero analizamos la representación de un modo similar a cómo se haría con `pivot_longer()`. Esta vez, sin embargo, necesitamos únicamente dos parámetros:

- La columna desde la que obtener los nombres de las variables. En este caso corresponde a `tipo`.
- La columna desde la que obtener los valores. En este caso corresponde a `cuenta`.

Una vez resuelto esto, podemos usar `pivot_wider()`, como se muestra programáticamente abajo y visualmente en la Figura 12.3.

```
tabla2 %>%
  pivot_wider(names_from = tipo, values_from = cuenta)
#> # A tibble: 6 x 4
#>   pais        anio  casos  población
#>   <chr>      <int>  <int>      <int>
#> 1 Afganistán  1999    745   19987071
#> 2 Afganistán  2000   2666   20595360
#> 3 Brasil      1999  37737  172006362
#> 4 Brasil      2000  80488  174504898
#> 5 China       1999 212258 1272915272
#> 6 China       2000 213766 1280428583
```

Copy

pais	anio	tipo	casos
Afganistán	1999	casos	745
Afganistán	1999	población	19987071
Afganistán	2000	casos	2666
Afganistán	2000	población	20595360
Brasil	1999	casos	37737
Brasil	1999	población	172006362
Brasil	2000	casos	80488
Brasil	2000	población	174504898
China	1999	casos	212258
China	1999	población	1272915272
China	2000	casos	213766
China	2000	población	1280428583

Tabla 2

pais	anio	casos	poblacion
Afganistán	1999	745	19987071
Afganistán	2000	2666	20595360
Brasil	1999	37737	172006362
Brasil	2000	80488	17504898
China	1999	212258	1272915272
China	2000	213766	1280428583

Figure 12.3: Pivotar la `tabla2` para un formato 'ancho' y ordenado

Como te habrás dado cuenta a partir de sus nombres, las funciones `pivot_longer()` y `pivot_wider()` son complementarias. `pivot_longer()` genera tablas angostas y largas, `pivot_wider()` genera tablas anchas y cortas.

12.3.3 **Ejercicios**

1. ¿Por qué `pivot_longer()` y `pivot_wider()` no son perfectamente simétricas? Observa cuidadosamente el siguiente ejemplo:

```
acciones <- tibble(
  anio = c(2015, 2015, 2016, 2016),
  semestre = c(1, 2, 1, 2),
  retorno = c(1.88, 0.59, 0.92, 0.17)
)
acciones %>%
  pivot_wider(names_from = anio, values_from = retorno) %>%
  pivot_longer(`2015`:`2016`, names_to = "anio", values_to = "retorno")
```

Copy

(Sugerencia: observa los tipos de variables y piensa en los nombres de las columnas)

```
`pivot_longer()` tiene el argumento `names_ptype`: por ejemplo: `names_ptype = list(year = double())`. ¿Qué es lo que hace dicho argumento?
```

Copy

1. ¿Por qué falla el siguiente código?

```
tabla4a %>%
  pivot_longer(c(1999, 2000), names_to = "anio", values_to = "casos")
#> Error: Can't subset columns that don't exist.
#> ✖ Locations 1999 and 2000 don't exist.
#> ℹ There are only 3 columns.
```

Copy

1. ¿Qué pasaría si trataras de pivotar esta tabla a lo ancho? ¿Por qué? ¿Cómo podrías agregar una nueva columna que identifique de manera única cada valor?

```
personas <- tribble(
  ~nombre, ~nombres, ~valores,
  #-----------------|--------|------
  "Phillip Woods", "edad", 45,
  "Phillip Woods", "estatura", 186,
  "Phillip Woods", "edad", 50,
  "Jessica Cordero", "edad", 37,
  "Jessica Cordero", "estatura", 156
)
```

1. Ordena la siguiente tabla. ¿Necesitas alargarla o ensancharla? ¿Cuáles son las variables?

```
embarazo <- tribble(
  ~embarazo, ~hombre, ~mujer,
  "sí", NA, 10,
  "no", 20, 12
)
```

12.4 Separar y unir

Hasta ahora has aprendido a ordenar las tablas `tabla2` y `tabla4`, pero no la `tabla3`, que tiene un problema diferente: tenemos una columna (`tasa`) que contiene dos variables (`casos` y `poblacion`). Para solucionar este problema, necesitamos la función `separate()` (*separar*). También aprenderás acerca del complemento de `separate()`: `unite()` (*unir*), que se usa cuando una única variable se reparte en varias columnas.

12.4.1 Separar

`separate()` desarma una columna en varias columnas, dividiendo de acuerdo a la posición de un carácter separador. Tomemos la `tabla3`:

```
tabla3
#> # A tibble: 6 x 3
#>   pais        anio tasa
#>   <chr>      <int> <chr>
#> 1 Afganistán  1999 745/19987071
#> 2 Afganistán  2000 2666/20595360
#> 3 Brasil      1999 37737/172006362
#> 4 Brasil      2000 80488/174504898
#> 5 China       1999 212258/1272915272
#> 6 China       2000 213766/1280428583
```

La columna `tasa` contiene tanto los `casos` como la `poblacion`, por lo que necesitamos dividirla en dos variables. La función `separate()` toma el nombre de la columna a separar y el nombre de las columnas a donde irá el resultado, tal como se muestra en la Figura 12.4 y el código a continuación.

```
tabla3 %>%
  separate(tasa, into = c("casos", "poblacion"))
#> # A tibble: 6 x 4
#>   pais        anio casos  poblacion
#>   <chr>      <int> <chr>  <chr>
#> 1 Afganistán  1999 745    19987071
#> 2 Afganistán  2000 2666   20595360
#> 3 Brasil      1999 37737  172006362
#> 4 Brasil      2000 80488  174504898
#> 5 China       1999 212258 1272915272
#> 6 China       2000 213766 1280428583
```

pais	anio	tasa
Afganistán	1999	**745** / 19987071
Afganistán	2000	**2666** / 20595360
Brasil	1999	**37737** / 172006362
Brasil	2000	**80488** / 17504898
China	1999	**212258** / 1272915272
China	2000	**213766** / 1280428583

Tabla 3

pais	anio	casos	poblacion
Afganistán	1999	**745**	19987071
Afganistán	2000	**2666**	20595360
Brasil	1999	**37737**	172006362
Brasil	2000	**80488**	17504898
China	1999	**212258**	1272915272
China	2000	**213766**	1280428583

Figure 12.4: Separar la `tabla3` la vuelve ordenada

Por defecto, `separate()` dividirá una columna donde encuentre un carácter no alfanumérico (esto es, un carácter que no es un número o letra). Por ejemplo, en el siguiente código, `separate()` divide los valores de `tasa` donde aparece una barra (`/`). Si deseas usar un carácter específico para separar una columna, puedes especificarlo en el argumento `sep` de `separate()` . Por ejemplo, el código anterior se puede re-escribir del siguiente modo:

```
tabla3 %>%
  separate(tasa, into = c("casos", "poblacion"), sep = "/")
```

(Formalmente, `sep` es una expresión regular y aprenderás más sobre esto en el capítulo sobre cadenas de caracteres.)

Mira atentamente los tipos de columna: notarás que `casos` y `poblacion` son columnas de tipo carácter. Este es el comportamiento por defecto en `separate()` : preserva el tipo de columna. Aquí, sin embargo, no es muy útil, ya que se trata de números. Podemos pedir a separate() que intente convertir a un tipo más adecuado usando `convert = TRUE` :

```
tabla3 %>%
  separate(tasa, into = c("casos", "poblacion"), convert = TRUE)
#> # A tibble: 6 x 4
#>   pais        anio  casos  poblacion
#>   <chr>      <int>  <int>      <int>
#> 1 Afganistán  1999    745   19987071
#> 2 Afganistán  2000   2666   20595360
#> 3 Brasil      1999  37737  172006362
#> 4 Brasil      2000  80488  174504898
#> 5 China       1999 212258 1272915272
#> 6 China       2000 213766 1280428583
```

También puedes pasar un vector de enteros a `sep` . `separate()` interpreta los enteros como las posiciones donde dividir. Los valores positivos comienzan en 1 al extremo izquierdo de las cadenas de texto; los valores negativos comienzan en -1 al extremo derecho. Cuando uses enteros para separar cadenas de texto, el largo de `sep` debe ser uno menos que el número de nombres en `into` .

Puedes usar este arreglo para separar los últimos dos dígitos de cada año. Esto deja los datos menos ordenados, pero es útil en otros casos, como se verá más adelante.

```
tabla3 %>%
  separate(anio, into = c("siglo", "anio"), sep = 2)
#> # A tibble: 6 x 4
#>   pais       siglo anio  tasa
#>   <chr>      <chr> <chr> <chr>
#> 1 Afganistán 19    99    745/19987071
#> 2 Afganistán 20    00    2666/20595360
#> 3 Brasil     19    99    37737/172006362
#> 4 Brasil     20    00    80488/174504898
#> 5 China      19    99    212258/1272915272
#> 6 China      20    00    213766/1280428583
```

12.4.2 **Unir**

`unite()` es el inverso de `separate()`: combina múltiples columnas en una única columna. Necesitarás esta función con mucha menos frecuencia que `separate()`, pero aún así es una buena herramienta para tener en el bolsillo trasero.

pais	anio	tasa
Afganistán	19**99**	**745** / 19987071
Afganistán	20**00**	**2666** / 20595360
Brasil	19**99**	**37737** / 172006362
Brasil	20**00**	**80488** / 17504898
China	19**99**	**212258** / 1272915272
China	20**00**	**213766** / 1280428583

pais	siglo	casos	tasa
Afganistán	19	**99**	**745** / 19987071
Afganistán	20	**0**	**2666** / 20595360
Brasil	19	**99**	**37737** / 172006362
Brasil	20	**0**	**80488** / 17504898
China	19	**99**	**212258** / 1272915272
China	20	**0**	**213766** / 1280428583

Tabla 6

Figure 12.5: Unir la `tabla5` la vuelve ordenada

Podemos usar `unite()` para unir las columnas *siglo* y *anio* creadas en el ejemplo anterior. Los datos están guardados en `datos::tabla5`. `unite()` toma un data frame, el nombre de la nueva variable a crear y un conjunto de columnas a combinar, las que se especifican siguiendo el estilo de la función `dplyr::select()`:

```
tabla5 %>%
  unite(nueva, siglo, anio)
#> # A tibble: 6 x 3
#>   pais       nueva tasa
#>   <chr>      <chr> <chr>
#> 1 Afganistán 19_99 745/19987071
#> 2 Afganistán 20_00 2666/20595360
#> 3 Brasil     19_99 37737/172006362
#> 4 Brasil     20_00 80488/174504898
#> 5 China      19_99 212258/1272915272
#> 6 China      20_00 213766/1280428583
```

Copy

En este caso también necesitamos el argumento `sep`. Por defecto, pondrá un guión bajo (_) entre los valores de las distintas columnas. Si no queremos ningún separador usamos `""`:

```
tabla5 %>%
  unite(nueva, siglo, anio, sep = "")
#> # A tibble: 6 x 3
#>   pais       nueva tasa
#>   <chr>      <chr> <chr>
#> 1 Afganistán 1999  745/19987071
#> 2 Afganistán 2000  2666/20595360
#> 3 Brasil     1999  37737/172006362
#> 4 Brasil     2000  80488/174504898
#> 5 China      1999  212258/1272915272
#> 6 China      2000  213766/1280428583
```

Copy

12.4.3 **Ejercicios**

1. ¿Qué hacen los argumentos `extra` y `fill` en `separate()`? Experimenta con las diversas opciones a partir de los siguientes datasets de ejemplo.

```
tibble(x = c("a,b,c", "d,e,f,g",   "h,i,j")) %>%
  separate(x, c("one", "two",       "three"))

tibble(x = c("a,b,c", "d,e",   "f,g,i")) %>%
  separate(x, c("one", "two",     "three"))
```

Copy

1. Tanto `unite()` como `separate()` tienen un argumento `remove`. ¿Qué es lo que hace? ¿Por qué lo dejarías en `FALSE`?

2. Compara y contrasta `separate()` y `extract()`. ¿Por qué existen tres variaciones de separación (por posición, separador y grupos), pero solo una forma de unir?

12.5 Valores faltantes

Cambiar la representación de un dataset conlleva el riesgo de generar valores faltantes. Sorprendentemente, un valor puede perderse de dos formas:

- **Explícita**, esto es, aparece como `NA`.
- **Implícita**, esto es, simplemente no aparece en los datos.

Ilustremos esta idea con un dataset muy sencillo:

```
acciones <- tibble(
  anio = c(2015, 2015, 2015, 2015, 2016, 2016, 2016),
  trimestre = c(1, 2, 3, 4, 2, 3, 4),
  retorno = c(1.88, 0.59, 0.35, NA, 0.92, 0.17, 2.66)
)
```

Existen dos valores faltantes en este dataset:

- El retorno del cuarto trimestre de 2015 que está explícitamente perdido, debido a que la celda donde el valor debiera estar contiene `NA`.
- El retorno del primer semestre de 2016 está implícitamente perdido, debido a que simplemente no aparece en el set de datos.

Una forma de pensar respecto de esta diferencia es al estilo de un kōan Zen: Un valor faltante explícito es la presencia de una ausencia; un valor faltante implícito es la ausencia de una presencia.

La forma en que se representa un dataset puede dejar explícitos los valores implícitos. Por ejemplo, podemos volver explícitos los valores faltantes implícitos al mover los años a las columnas:

```
acciones %>%
  spread(anio, retorno)
#> # A tibble: 4 x 3
#>   trimestre `2015` `2016`
#>       <dbl>  <dbl>  <dbl>
#> 1         1   1.88  NA
#> 2         2   0.59   0.92
#> 3         3   0.35   0.17
#> 4         4  NA      2.66
```

Debido a que estos valores faltantes explícitos pueden no ser tan importantes en otras representaciones de los datos, puedes especificar `na.rm = TRUE` en `gather()` para dejar explícitos los valores faltantes implícitos:

```
acciones %>%
  pivot_wider(names_from = anio, values_from = retorno) %>%
  pivot_longer(
    cols = c(`2015`, `2016`),
    names_to = "anio",
    values_to = "retorno",
    values_drop_na = TRUE
  )
#> # A tibble: 6 x 3
#>   trimestre anio  retorno
#>       <dbl> <chr>   <dbl>
#> 1         1 2015     1.88
#> 2         2 2015     0.59
#> 3         2 2016     0.92
#> 4         3 2015     0.35
#> 5         3 2016     0.17
#> 6         4 2016     2.66
```

Otra herramienta importante para hacer explícitos los valores faltantes en datos ordenados es `complete()`:

```
acciones %>%
  complete(anio, trimestre)
#> # A tibble: 8 x 3
#>    anio trimestre retorno
#>   <dbl>     <dbl>   <dbl>
#> 1  2015         1    1.88
#> 2  2015         2    0.59
#> 3  2015         3    0.35
#> 4  2015         4   NA
#> 5  2016         1   NA
#> 6  2016         2    0.92
#> # … with 2 more rows
```

`complete()` toma un conjunto de columnas y encuentra todas las combinaciones únicas. Luego se asegura de que el dataset original contenga todos los valores, completando con `NA`s donde sea necesario.

Existe otra herramienta importante que deberías conocer al momento de trabajar con valores faltantes. En algunos casos en que la fuente de datos se ha usado principalmente para ingresar datos, los valores faltantes indican que el valor previo debe arrastrarse hacia adelante:

```
tratamiento <- tribble(
  ~sujeto, ~tratamiento, ~respuesta,
  "Derrick Whitmore", 1, 7,
  NA, 2, 10,
  NA, 3, 9,
  "Katherine Burke", 1, 4
)
```

Puedes completar los valores faltantes usando `fill()`. Esta función toma un conjunto de columnas sobre las cuales los valores faltantes son reemplazados por el valor anterior más cercano que se haya reportado (también conocido como el método LOCF, del inglés *last observation carried forward*).

```
tratamiento %>%
  fill(sujeto)
#> # A tibble: 4 x 3
#>   sujeto           tratamiento respuesta
#>   <chr>                  <dbl>     <dbl>
#> 1 Derrick Whitmore           1         7
#> 2 Derrick Whitmore           2        10
#> 3 Derrick Whitmore           3         9
#> 4 Katherine Burke            1         4
```

12.5.1 Ejercicios

1. Compara y contrasta el argumento `fill` que se usa en `pivot_wider()` con `complete()`.

2. ¿Qué hace el argumento de dirección en `fill()`?

12.6 Estudio de caso

Para finalizar el capítulo, combinemos todo lo que aprendiste para enfrentar un problema real de ordenamiento de datos. El dataset `datos::oms` contiene datos de tuberculosis (TB) detallados por año, país, edad, sexo y método de diagnóstico. Los datos provienen del *Informe de Tuberculosis de la Organización Mundial de la Salud 2014*, disponible en http://www.who.int/tb/country/data/download/en/.

Existe abundante información epidemiológica en este dataset, pero es complicado trabajar con estos datos tal como son entregados:

```
oms
```

Copy

```
#> # A tibble: 7,240 x 60
#>   pais  iso2  iso3   anio nuevos_fpp_h014 nuevos_fpp_h1524 nuevos_fpp_h2534
#>   <chr> <chr> <chr> <int>           <int>            <int>            <int>
#> 1 Afga… AF    AFG    1980              NA               NA               NA
#> 2 Afga… AF    AFG    1981              NA               NA               NA
#> 3 Afga… AF    AFG    1982              NA               NA               NA
#> 4 Afga… AF    AFG    1983              NA               NA               NA
#> 5 Afga… AF    AFG    1984              NA               NA               NA
#> 6 Afga… AF    AFG    1985              NA               NA               NA
#> # … with 7,234 more rows, and 53 more variables: nuevos_fpp_h3544 <int>,
#> #   nuevos_fpp_h4554 <int>, nuevos_fpp_h5564 <int>, nuevos_fpp_h65 <int>,
#> #   nuevos_fpp_m014 <int>, nuevos_fpp_m1524 <int>, nuevos_fpp_m2534 <int>,
#> #   nuevos_fpp_m3544 <int>, nuevos_fpp_m4554 <int>, nuevos_fpp_m5564 <int>,
#> #   nuevos_fpp_m65 <int>, nuevos_fpn_h014 <int>, nuevos_fpn_h1524 <int>,
#> #   nuevos_fpn_h2534 <int>, nuevos_fpn_h3544 <int>, nuevos_fpn_h4554 <int>,
#> #   nuevos_fpn_h5564 <int>, nuevos_fpn_h65 <int>, nuevos_fpn_m014 <int>,
#> #   nuevos_fpn_m1524 <int>, nuevos_fpn_m2534 <int>, nuevos_fpn_m3544 <int>,
#> #   nuevos_fpn_m4554 <int>, nuevos_fpn_m5564 <int>, nuevos_fpn_m65 <int>,
#> #   nuevos_ep_h014 <int>, nuevos_ep_h1524 <int>, nuevos_ep_h2534 <int>,
#> #   nuevos_ep_h3544 <int>, nuevos_ep_h4554 <int>, nuevos_ep_h5564 <int>,
#> #   nuevos_ep_h65 <int>, nuevos_ep_m014 <int>, nuevos_ep_m1524 <int>,
#> #   nuevos_ep_m2534 <int>, nuevos_ep_m3544 <int>, nuevos_ep_m4554 <int>,
#> #   nuevos_ep_m5564 <int>, nuevos_ep_m65 <int>, nuevosrecaida_h014 <int>,
#> #   nuevosrecaida_h1524 <int>, nuevosrecaida_h2534 <int>,
#> #   nuevosrecaida_h3544 <int>, nuevosrecaida_h4554 <int>,
#> #   nuevosrecaida_h5564 <int>, nuevosrecaida_h65 <int>,
#> #   nuevosrecaida_m014 <int>, nuevosrecaida_m1524 <int>,
#> #   nuevosrecaida_m2534 <int>, nuevosrecaida_m3544 <int>,
#> #   nuevosrecaida_m4554 <int>, nuevosrecaida_m5564 <int>,
#> #   nuevosrecaida_m65 <int>
```

Este es un ejemplo muy típico de un dataset de la vida real. Contiene columnas redundantes, códigos extraños de variables y muchos valores faltantes. En breve, `oms` está desordenado y necesitamos varios pasos para ordenarlo. Al igual que **dplyr**, **tidyr** está diseñado de modo tal que cada función hace bien una cosa. Esto significa que en una situación real deberás encadenar múltiples verbos.

LCasi siempre, la mejor forma de comenzar es reunir las columnas que no representan variables. Miremos lo que hay:

- Pareciera ser que `pais`, `iso2` e `iso3` son variables redundantes que se refieren al país.
- `anio` es claramente una variable.
- No sabemos aún el significado de las otras columnas, pero dada la estructura de los nombres de las variables (e.g. `nuevos_fpp_h014`, `nuevos_ep_h014`, `nuevos_ep_m014`) parecieran ser valores y no variables.

Necesitamos agrupar todas las columnas desde `nuevos_fpp_h014` hasta `recaidas_m65`. No sabemos aún qué representa esto, por lo que le daremos el nombre genérico de `"clave"`. Sabemos que las celdas representan la cuenta de casos, por lo que usaremos la variable `casos`.

Existen múltiples valores faltantes en la representación actual, por lo que de momento usaremos `na.rm` para centrarnos en los valores que están presentes.

```
oms1 <- oms %>%
  pivot_longer(
    cols = nuevos_fpp_h014:nuevosrecaida_m65,
    names_to = "clave",
    values_to = "casos",
    values_drop_na = TRUE
  )
oms1
#> # A tibble: 76,046 x 6
#>   pais       iso2  iso3   anio clave            casos
#>   <chr>      <chr> <chr> <int> <chr>            <int>
#> 1 Afganistán AF    AFG    1997 nuevos_fpp_h014      0
#> 2 Afganistán AF    AFG    1997 nuevos_fpp_h1524    10
#> 3 Afganistán AF    AFG    1997 nuevos_fpp_h2534     6
#> 4 Afganistán AF    AFG    1997 nuevos_fpp_h3544     3
#> 5 Afganistán AF    AFG    1997 nuevos_fpp_h4554     5
#> 6 Afganistán AF    AFG    1997 nuevos_fpp_h5564     2
#> # … with 76,040 more rows
```

Copy

Podemos tener una noción de la estructura de los valores en la nueva columna `clave` si hacemos un conteo:

```
oms1 %>%
  count(clave)
#> # A tibble: 56 x 2
#>   clave               n
#> * <chr>           <int>
#> 1 nuevos_ep_h014   1038
#> 2 nuevos_ep_h1524  1026
#> 3 nuevos_ep_h2534  1020
#> 4 nuevos_ep_h3544  1024
#> 5 nuevos_ep_h4554  1020
#> 6 nuevos_ep_h5564  1015
#> # … with 50 more rows
```

Copy

Podrías resolver esto por tu cuenta pensando y experimentando un poco, pero afortunadamente tenemos el diccionario de datos a mano. Este nos dice lo siguiente:

1. Lo que aparece antes del primer `_` en las columnas indica si la columna contiene casos nuevos o antiguos de tuberculosis. En este dataset, cada columna contiene nuevos casos.

2. Lo que aparece luego de indicar si se refiere casos nuevos o antiguos es el tipo de tuberculosis:

- `recaida` se refiere a casos reincidentes
- `ep` se refiere a tuberculosis extra pulmonar
- `fpn` se refiere a casos de tuberculosis pulmonar que no se pueden detectar mediante examen de frotis pulmonar (frotis pulmonar negativo)
- `fpp` se refiere a casos de tuberculosis pulmonar que se pueden detectar mediante examen de frotis pulmonar (frotis pulmonar positivo)

3. La letra que aparece después del último `_` se refiere al sexo de los pacientes. El conjunto de datos agrupa en hombres (`h`) y mujeres (`m`).

4. Los números finales se refieren al grupo etareo que se ha organizado en siete categorías:

- `014` = 0 – 14 años de edad
- `1524` = 15 – 24 años de edad
- `2534` = 25 – 34 años de edad
- `3544` = 35 – 44 años de edad
- `4554` = 45 – 54 años de edad
- `5564` = 55 – 64 años de edad
- `65` = 65 o más años de edad

Necesitamos hacer un pequeño cambio al formato de los nombres de las columnas: desafortunadamente lo nombres de las columnas son ligeramente inconsistentes debido a que en lugar de `nuevos_recaida` tenemos `nuevosrecaida` (es difícil darse cuenta de esto en esta parte, pero si no lo arreglas habrá errores en los pasos siguientes). Aprenderás sobre `str_replace()` en cadenas de caracteres, pero la idea básica es bastante simple: reemplazar los caracteres "nuevosrecaida" por "nuevos_recaida". Esto genera nombres de variables consistentes.

```
oms2 <- oms1 %>%
  mutate(clave = stringr::str_replace(clave, "nuevosrecaida", "nuevos_recaida"))
oms2
#> # A tibble: 76,046 x 6
#>   pais       iso2  iso3   anio clave            casos
#>   <chr>      <chr> <chr> <int> <chr>            <int>
#> 1 Afganistán AF    AFG    1997 nuevos_fpp_h014      0
#> 2 Afganistán AF    AFG    1997 nuevos_fpp_h1524    10
#> 3 Afganistán AF    AFG    1997 nuevos_fpp_h2534     6
#> 4 Afganistán AF    AFG    1997 nuevos_fpp_h3544     3
#> 5 Afganistán AF    AFG    1997 nuevos_fpp_h4554     5
#> 6 Afganistán AF    AFG    1997 nuevos_fpp_h5564     2
#> # … with 76,040 more rows
```

Copy

Podemos separar los valores en cada código aplicando `separate()` dos veces. La primera aplicación dividirá los códigos en cada `_`.

```
oms3 <- oms2 %>%
  separate(clave, c("nuevos", "tipo", "sexo_edad"), sep = "_")
oms3
#> # A tibble: 76,046 x 8
#>   pais       iso2  iso3   anio nuevos tipo  sexo_edad casos
#>   <chr>      <chr> <chr> <int> <chr>  <chr> <chr>     <int>
#> 1 Afganistán AF    AFG    1997 nuevos fpp   h014          0
#> 2 Afganistán AF    AFG    1997 nuevos fpp   h1524        10
#> 3 Afganistán AF    AFG    1997 nuevos fpp   h2534         6
#> 4 Afganistán AF    AFG    1997 nuevos fpp   h3544         3
#> 5 Afganistán AF    AFG    1997 nuevos fpp   h4554         5
#> 6 Afganistán AF    AFG    1997 nuevos fpp   h5564         2
#> # … with 76,040 more rows
```

Copy

A continuación podemos eliminar la columna `nuevos`, ya que es constante en este dataset. Además eliminaremos `iso2` e `iso3` ya que son redundantes.

```
oms3 %>%
  count(nuevos)
#> # A tibble: 1 x 2
#>   nuevos     n
#> * <chr>  <int>
#> 1 nuevos 76046
oms4 <- oms3 %>%
  select(-nuevos, -iso2, -iso3)
```

Copy

Luego separamos `sexo_edad` en `sexo` y `edad` dividiendo luego del primer carácter:

```
oms5 <- oms4 %>%
  separate(sexo_edad, c("sexo", "edad"), sep = 1)
oms5
#> # A tibble: 76,046 x 6
#>   pais        anio tipo  sexo  edad  casos
#>   <chr>      <int> <chr> <chr> <chr> <int>
#> 1 Afganistán  1997 fpp   h     014       0
#> 2 Afganistán  1997 fpp   h     1524     10
#> 3 Afganistán  1997 fpp   h     2534      6
#> 4 Afganistán  1997 fpp   h     3544      3
#> 5 Afganistán  1997 fpp   h     4554      5
#> 6 Afganistán  1997 fpp   h     5564      2
#> # … with 76,040 more rows
```

Copy

¡Ahora el dataset `oms` está ordenado!

Hemos mostrado el código parte por parte, asignando los resultados intermedios a nuevas variables. Esta no es la forma típica de trabajo. En cambio, lo que se hace es formar incrementalmente un encadenamiento complejo usando *pipes*:

```
oms %>%
  pivot_longer(
    cols = nuevos_fpp_h014:nuevosrecaida_m65,
    names_to = "clave",
    values_to = "valor",
    values_drop_na = TRUE) %>%
  mutate(clave = stringr::str_replace(clave, "nuevosrecaida", "nuevos_recaida")) %>%
  separate(clave, c("nuevos", "tipo", "sexo_edad")) %>%
  select(-nuevos, -iso2, -iso3) %>%
  separate(sexo_edad, c("sexo", "edad"), sep = 1)
```

Copy

12.6.1 Ejercicios

1. En este caso de estudio fijamos `values_drop_na = TRUE` para hacer más simple el verificar que tenemos los valores correctos. ¿Es esto razonable? Piensa en cómo los valores faltantes están representados en este dataset. ¿Existen valores faltantes implícitos? ¿Cuál es la diferencia entre `NA` y cero?

2. ¿Qué ocurre si omites la aplicación de `mutate()` ? (`mutate(clave = stringr::str_replace(clave, "nuevosrecaida", "nuevos_recaida"))`)

3. Afirmamos que `iso2` e `iso3` son redundantes respecto a `pais` . Confirma esta aseveración.

4. Para cada país, año y sexo calcula el total del número de casos de tuberculosis. Crea una visualización informativa de los datos.

12.7 Datos no ordenados

Antes de pasar a otros tópicos, es conveniente referirse brevemente a datos no ordenados. Anteriormente en el capítulo, usamos el término peyorativo “desordenados” para referirnos a datos no ordenados. Esto es una sobresimplificación: existen múltiples estructuras de datos debidamente fundamentadas que no corresponden a datos ordenados. Existen dos principales razones para usar otras estructuras de datos:

- Las representaciones alternativas pueden traer ventajas importantes en términos de desempeño o tamaño.

- Algunas áreas especializadas han evolucionado y tienen sus propias convenciones para almacenar datos, las que pueden diferir respecto de las convenciones de datos ordenados.

Cada uno de estas razones significa que necesitarás algo distinto a un tibble (o data frame). Si tus datos naturalmente se ajustan a una estructura rectangular compuesta de observaciones y variables, pensamos que datos ordenados debería ser tu elección por defecto. Sin embargo, existen buenas razones para usar otras estructuras; los datos ordenados no son la única forma.

Si quieres aprender más acerca de datos no ordenados, recomendamos fuertemente este artículo del blog de Jeff Leek: http://simplystatistics.org/2016/02/17/non-tidy-data/

« 11 Importación de datos

13 Datos relacionales »

13 Datos relacionales

13.1 Introducción

Es raro que un análisis de datos involucre una única tabla de datos. Lo típico es que tengas muchas tablas que debes combinar para responder a tus preguntas de interés. De manera colectiva, se le llama *datos relacionales* a esas múltiples tablas de datos, ya que sus relaciones, y no solo los conjuntos de datos individuales, son importantes.

Las relaciones siempre se definen sobre un par de tablas. Todas las otras relaciones se construyen sobre esta idea simple: las relaciones entre tres o más tablas son siempre una propiedad de las relaciones entre cada par. ¡A veces ambos elementos de un par pueden ser la misma tabla! Esto se necesita si, por ejemplo, tienes una tabla de personas y cada persona tiene una referencia a sus padres.

Para trabajar con datos relacionales necesitas verbos que funcionen con pares de tablas. Existen tres familias de verbos diseñadas para trabajar con datos relacionales:

- **Uniones de transformación** (del inglés *mutating joins*), que agregan nuevas variables a un *data frame* a partir de las observaciones coincidentes en otra tabla.
- **Uniones de filtro** (del inglés *filtering joins*), que filtran observaciones en un *data frame* con base en si coinciden o no con una observación de otra tabla.
- **Operaciones de conjuntos** (del inglés *set operations*), que tratan las observaciones como elementos de un conjunto.

El lugar más común para encontrar datos relacionales es en un sistema *relacional* de administración de bases de datos (*Relational Data Base Management System* en inglés), un concepto que abarca casi todas las bases de datos modernas. Si has usado una base de datos con anterioridad, casi seguramente fue SQL. Si es así, los conceptos de este capítulo debiesen ser familiares, aunque su expresión en **dplyr** es un poco distinta. En términos generales, **dplyr** es un poco más fácil de usar que SQLes, ya que dplyr se especializa en el análisis de datos: facilita las operaciones habituales, a expensas de dificultar otras que no se requieren a menudo para el análisis de datos.

13.1.1 Prerrequisitos

Vamos a explorar datos relacionales contenidos en el paquete **datos** usando los verbos para dos tablas de **dplyr**.

```
library(tidyverse)
library(datos)
```

Copy

13.2 Datos sobre vuelos

Usaremos datos sobre vuelos desde y hacia Nueva York para aprender sobre datos relacionales[1]. El paquete **datos** contiene cuatro tablas que se relacionan con la tabla `vuelos` que se usó en el capítulo sobre transformación de datos:

- `aerolineas` permite observar el nombre completo de la aerolínea a partir de su código abreviado:

```
aerolineas
#> # A tibble: 16 x 2
#>   aerolinea nombre
#>   <chr>     <chr>
#> 1 9E        Endeavor Air Inc.
#> 2 AA        American Airlines Inc.
#> 3 AS        Alaska Airlines Inc.
#> 4 B6        JetBlue Airways
#> 5 DL        Delta Air Lines Inc.
#> 6 EV        ExpressJet Airlines Inc.
#> # … with 10 more rows
```

- `aeropuertos` entrega información de cada aeropuerto, identificado por su `código` :

```
aeropuertos
#> # A tibble: 1,458 x 8
#>   codigo_aeropuer… nombre latitud longitud altura zona_horaria horario_verano
#>   <chr>            <chr>    <dbl>    <dbl>  <dbl>        <dbl> <chr>
#> 1 04G              Lansd…    41.1    -80.6   1044           -5 A
#> 2 06A              Moton…    32.5    -85.7    264           -6 A
#> 3 06C              Schau…    42.0    -88.1    801           -6 A
#> 4 06N              Randa…    41.4    -74.4    523           -5 A
#> 5 09J              Jekyl…    31.1    -81.4     11           -5 A
#> 6 0A9              Eliza…    36.4    -82.2   1593           -5 A
#> # … with 1,452 more rows, and 1 more variable: zona_horaria_iana <chr>
```

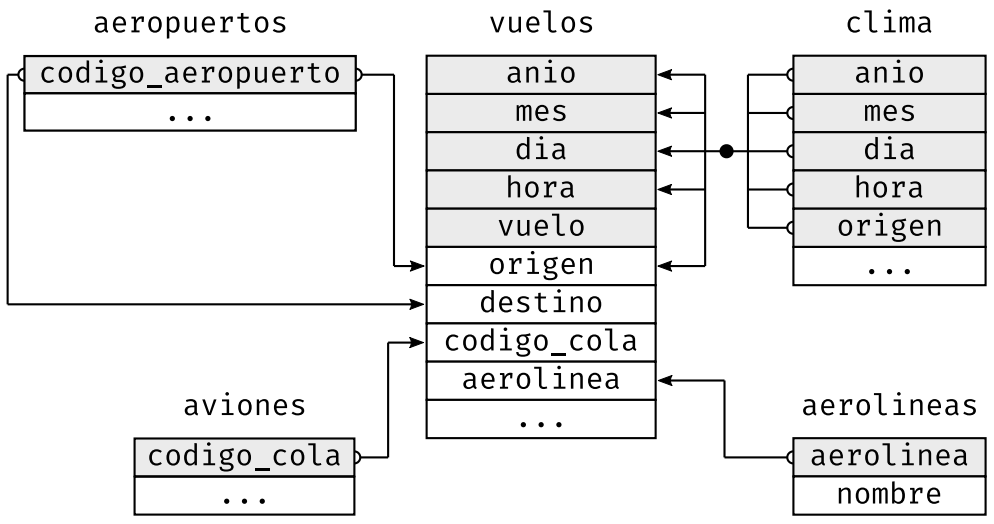
- `aviones` entrega información de cada avión, identificado por su `codigo_cola` :

```
aviones
#> # A tibble: 3,322 x 9
#>   codigo_cola  anio tipo  fabricante modelo motores asientos velocidad
#>   <chr>       <int> <chr> <chr>      <chr>    <int>    <int>     <int>
#> 1 N10156       2004 Ala … EMBRAER    EMB-1…       2       55        NA
#> 2 N102UW       1998 Ala … AIRBUS IN… A320-…       2      182        NA
#> 3 N103US       1999 Ala … AIRBUS IN… A320-…       2      182        NA
#> 4 N104UW       1999 Ala … AIRBUS IN… A320-…       2      182        NA
#> 5 N10575       2002 Ala … EMBRAER    EMB-1…       2       55        NA
#> 6 N105UW       1999 Ala … AIRBUS IN… A320-…       2      182        NA
#> # … with 3,316 more rows, and 1 more variable: tipo_motor <chr>
```

- `clima` entrega información del clima en cada aeropuerto de Nueva York para cada hora:

```
clima
#> # A tibble: 26,115 x 15
#>   origen  anio   mes   dia  hora temperatura punto_rocio humedad
#>   <chr>  <int> <int> <int> <int>       <dbl>       <dbl>   <dbl>
#> 1 EWR     2013     1     1     1        39.0        26.1    59.4
#> 2 EWR     2013     1     1     2        39.0        27.0    61.6
#> 3 EWR     2013     1     1     3        39.0        28.0    64.4
#> 4 EWR     2013     1     1     4        39.9        28.0    62.2
#> 5 EWR     2013     1     1     5        39.0        28.0    64.4
#> 6 EWR     2013     1     1     6        37.9        28.0    67.2
#> # … with 26,109 more rows, and 7 more variables: direccion_viento <dbl>,
#> #   velocidad_viento <dbl>, velocidad_rafaga <dbl>, precipitacion <dbl>,
#> #   presion <dbl>, visibilidad <dbl>, fecha_hora <dttm>
```

Una forma de mostrar las relaciones entre las diferentes tablas es mediante un diagrama:



Este diagrama es un poco abrumador, ¡pero es simple comparado con algunos que verás en el exterior! La clave para entender estos diagramas es recordar que cada relación siempre involucra un par de tablas. No necesitas entender todo el diagrama, necesitas entender la cadena de relaciones entre las tablas que te interesan.

En estos datos:

- `vuelos` se conecta con `aviones` a través de la variable `codigo_cola`.
- `vuelos` se conecta con `aerolineas` a través de la variable `codigo_carrier`.
- `vuelos` se conecta con `aeropuertos` de dos formas: a través de las variables `origen` y `destino`.
- `vuelos` se conecta con `clima` a través de las variables `origen` (la ubicación), `anio`, `mes`, `dia` y `hora` (el horario).

13.2.1 **Ejercicios**

1. Imagina que necesitas dibujar (aproximadamente) la ruta que cada avión vuela desde su origen hasta el destino. ¿Qué variables necesitarías? ¿Qué tablas necesitarías combinar?
2. Olvidamos dibujar la relación entre `clima` y `aeropuertos`. ¿Cuál es la relación y cómo debería aparecer en el diagrama?
3. `clima` únicamente contiene información de los aeropuertos de origen (Nueva York). Si incluyera registros para todos los aeropuertos de EEUU, ¿qué relación tendría con `vuelos`?
4. Sabemos que hay días "especiales" en el año y pocas personas suelen volar en ellos. ¿Cómo se representarían en un data frame? ¿Cuáles serían las claves primarias de esa tabla? ¿Cómo se conectaría con las tablas existentes?

13.3 **Claves**

Las variables usadas para conectar cada par de variables se llaman *claves* (del inglés *key*). Una clave es una variable (o un conjunto de variables) que identifican de manera única una observación. En casos simples, una sola variable es suficiente para identificar una observación. Por ejemplo, cada avión está identificado de forma única por su `codigo_cola`. En otros casos, se pueden necesitar múltiples variables. Por ejemplo, para identificar una observación en `clima` se necesitan cinco variables: `anio`, `mes`, `dia`, `hora` y `origen`.

Existen dos tipos de claves:

- Una *clave primaria* identifica únicamente una observación en su propia tabla. Por ejemplo, `aviones$codigo_cola` es una clave primaria, ya que identifica de manera única cada avión en la tabla `aviones`.
- Una *clave foránea* únicamente identifica una observación en otra tabla. Por ejemplo, `vuelos$codigo_cola` es una clave foránea, ya que aparece en la tabla `vuelos`, en la que une cada vuelo con un único avión.

Una variable puede ser clave primaria *y* clave foránea a la vez. Por ejemplo, `origen` es parte de la clave primaria `clima` y también una clave foránea de `aeropuertos`.

Una vez que identificas las claves primarias en tus tablas, es una buena práctica verificar que identifican de forma única cada observación. Una forma de hacerlo es usar `count()` con las claves primarias y buscar las entradas con `n` mayor a uno:

```
aviones %>%
  count(codigo_cola) %>%
  filter(n > 1)
#> # A tibble: 0 x 2
#> # … with 2 variables: codigo_cola <chr>, n <int>

clima %>%
  count(anio, mes, dia, hora, origen) %>%
  filter(n > 1)
#> # A tibble: 3 x 6
#>    anio   mes   dia  hora origen     n
#>   <int> <int> <int> <int> <chr>  <int>
#> 1  2013    11     3     1 EWR        2
#> 2  2013    11     3     1 JFK        2
#> 3  2013    11     3     1 LGA        2
```

A veces una tabla puede no tener una clave primaria explícita: cada fila es una observación, pero no existe una combinación de variables que la identifique de forma confiable. Por ejemplo, ¿cuál es la clave primaria en la tabla `vuelos` ? Quizás pienses que podría ser la fecha más el vuelo o el código de cola, pero ninguna de esas variables es única:

```
vuelos %>%
  count(anio, mes, dia, vuelo) %>%
  filter(n > 1)
#> # A tibble: 29,768 x 5
#>    anio   mes   dia vuelo     n
#>   <int> <int> <int> <int> <int>
#> 1  2013     1     1     1     2
#> 2  2013     1     1     3     2
#> 3  2013     1     1     4     2
#> 4  2013     1     1    11     3
#> 5  2013     1     1    15     2
#> 6  2013     1     1    21     2
#> # … with 29,762 more rows

vuelos %>%
  count(anio, mes, dia, codigo_cola) %>%
  filter(n > 1)
#> # A tibble: 64,928 x 5
#>    anio   mes   dia codigo_cola     n
#>   <int> <int> <int> <chr>       <int>
#> 1  2013     1     1 N0EGMQ          2
#> 2  2013     1     1 N11189          2
#> 3  2013     1     1 N11536          2
#> 4  2013     1     1 N11544          3
#> 5  2013     1     1 N11551          2
#> 6  2013     1     1 N12540          2
#> # … with 64,922 more rows
```

Al comenzar a trabajar con estos datos, ingenuamente asumimos que cada número de vuelo sería usado solo una vez al día: eso haría mucho más simple comunicar problemas con un vuelo específico. ¡Desafortunadamente este no es el caso! Si una tabla no tiene una clave primaria, a veces es útil incluir una con `mutate()` y `row_number()` (*número de fila*). Eso simplifica hacer coincidir observaciones una vez que haz hecho algunos filtros y quieres volver a verificar con los datos originales. Esto se llama **clave subrogada**.

Una clave primaria y su correspondiente clave foránea en otra tabla forman una **relación**. Las relaciones son típicamente uno-a-muchos. Por ejemplo, cada vuelo tiene un avión, pero cada avión tiene muchos vuelos. En otros datos, ocasionalmente verás relaciones uno-a-uno. Puedes pensar esto como un caso especial de uno-a-muchos. Puedes modelar relaciones muchos-a-muchos como relaciones de la forma

muchos-a-uno y uno-a-muchos. Por ejemplo, en estos datos existe una relación muchos-a-muchos entre aerolíneas y aeropuertos: cada aerolínea vuela a muchos aeropuertos, cada aeropuerto recibe a muchas aerolíneas.

13.3.1 Ejercicios

1. Agrega una clave subrogada a `vuelos`.

2. Identifica las claves en los siguientes conjuntos de datos

3. `datos::bateadores`

4. `datos::nombres`

5. `datos::atmosfera`

6. `datos::vehiculos`

7. `datos::diamantes`

(Puede que necesites leer un poco de documentación.)

1. Dibuja un diagrama que ilustre las conexiones entre las tablas `bateadores`, `personas` y `salarios` incluidas en el paquete **datos**. Dibuja otro diagrama que muestre la relación entre `personas`, `dirigentes` y `premios_dirigentes`.

   ¿Cómo caracterizarías las relación entre `bateadores`, `lanzadores` y `jardineros`?

13.4 Uniones de transformación

La primera herramienta que miraremos para combinar pares de variables es la **unión de transformación** (*mutating join*). Una unión de transformación te permite combinar variables a partir de dos tablas. Primero busca coincidencias de observaciones de acuerdo a sus claves y luego copia las variables de una tabla en la otra.

Tal como `mutate()`, las funciones de unión agregan variables hacia la derecha, por lo que si tienes muchas variables inicialmente, las nuevas variables no se imprimirán. Para estos ejemplos, crearemos un conjunto de datos más angosto para que sea más fácil ver qué es lo que está ocurriendo:

```
vuelos2 <- vuelos %>%
  select(anio:dia, hora, origen, destino, codigo_cola, aerolinea)
vuelos2
#> # A tibble: 336,776 x 8
#>    anio   mes   dia  hora origen destino codigo_cola aerolinea
#>   <int> <int> <int> <dbl> <chr>  <chr>   <chr>       <chr>
#> 1  2013     1     1     5 EWR    IAH     N14228      UA
#> 2  2013     1     1     5 LGA    IAH     N24211      UA
#> 3  2013     1     1     5 JFK    MIA     N619AA      AA
#> 4  2013     1     1     5 JFK    BQN     N804JB      B6
#> 5  2013     1     1     6 LGA    ATL     N668DN      DL
#> 6  2013     1     1     5 EWR    ORD     N39463      UA
#> # … with 336,770 more rows
```

(Recuerda que en RStudio puedes también usar `View()` para evitar este problema.)

Imagina que quieres incluir el nombre completo de la aerolínea en `vuelos2`. Puedes combinar los datos de `aerolinas` y `vuelos2` con `left_join()` (*union_izquierda*):

```
vuelos2 %>%
  select(-origen, -destino) %>%
  left_join(aerolineas, by = "aerolinea")
#> # A tibble: 336,776 x 7
#>    anio   mes   dia  hora codigo_cola aerolinea nombre
#>   <int> <int> <int> <dbl> <chr>       <chr>     <chr>
#> 1  2013     1     1     5 N14228      UA        United Air Lines Inc.
#> 2  2013     1     1     5 N24211      UA        United Air Lines Inc.
#> 3  2013     1     1     5 N619AA      AA        American Airlines Inc.
#> 4  2013     1     1     5 N804JB      B6        JetBlue Airways
#> 5  2013     1     1     6 N668DN      DL        Delta Air Lines Inc.
#> 6  2013     1     1     5 N39463      UA        United Air Lines Inc.
#> # … with 336,770 more rows
```

El resultado de unir `aerolineas` y `vuelos2` es la inclusión de una variable adicional: `nombre`. Esta es la razón de que llamemos unión de transformación a este tipo de unión. En este caso, podrías obtener el mismo resultado usando `mutate()` junto a las operaciones de filtro de R base:

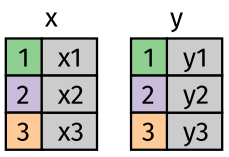
```
vuelos2 %>%
  select(-origen, -destino) %>%
  mutate(nombre = aerolineas$nombre[match(aerolinea, aerolineas$aerolinea)])
#> # A tibble: 336,776 x 7
#>    anio   mes   dia  hora codigo_cola aerolinea nombre
#>   <int> <int> <int> <dbl> <chr>       <chr>     <chr>
#> 1  2013     1     1     5 N14228      UA        United Air Lines Inc.
#> 2  2013     1     1     5 N24211      UA        United Air Lines Inc.
#> 3  2013     1     1     5 N619AA      AA        American Airlines Inc.
#> 4  2013     1     1     5 N804JB      B6        JetBlue Airways
#> 5  2013     1     1     6 N668DN      DL        Delta Air Lines Inc.
#> 6  2013     1     1     5 N39463      UA        United Air Lines Inc.
#> # … with 336,770 more rows
```

Sin embargo, esto último es difícil de generalizar cuando necesitas hacer coincidir múltiples variables y requiere hacer una lectura detenida para entender lo que se quiere hacer.

En las siguientes secciones explicaremos en detalle cómo funcionan las uniones de transformación. Comenzarás aprendiendo una representación visual útil de las uniones. Luego usaremos eso para explicar las cuatro uniones de transformación: la unión interior y las tres uniones exteriores. Cuando trabajas con datos reales, las claves no siempre identifican a las observaciones de forma única. Es por eso que a continuación hablaremos de lo que ocurre cuando no existe una coincidencia única. Finalmente, aprenderás cómo decirle a **dplyr** qué variables son las claves para una unión determinada.

13.4.1 **Entendiendo las uniones**

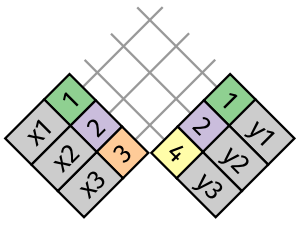
Para ayudarte a entender las uniones, usaremos una representación gráfica:




```
x <- tribble(
  ~key, ~val_x,
  1, "x1",
  2, "x2",
  3, "x3"
)
y <- tribble(
  ~key, ~val_y,
  1, "y1",
  2, "y2",
  4, "y3"
)
```

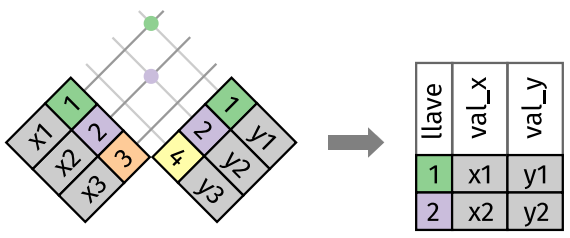
La columna coloreada representa la variable “clave”: estas se usan para unir filas entre las tablas. La columa gris representa la columna “valor” que se usa en todo el proceso. En estos ejemplos te mostraremos una única clave, pero la idea es generalizable de manera directa a múltiples claves y múltiples valores.

Una unión es una forma de conectar cada fila en `x` con cero, una o más filas en `y`. El siguiente diagrama muestra cada coincidencia potencial como una intersección de pares de líneas.



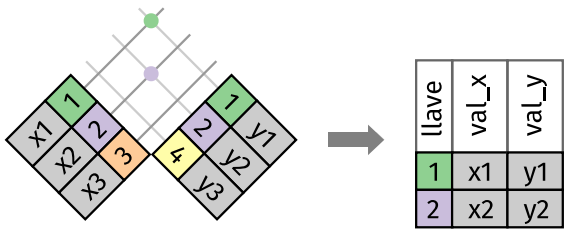
(Si observas detenidamente, te darás cuenta de que hemos cambiado el orden de las columnas clave y valor en `x`. Esto es para enfatizar que las uniones encuentran coincidencias basadas en las claves; el valor simplemente se traslada durante el proceso.)

En la unión que mostramos, las coincidencias se indican con puntos. El número de puntos es igual al número de coincidencias y al número de filas en la salida.



13.4.2 **Unión interior**

La forma más simple de unión es la *unión interior* (del inglés *inner join*). Una unión interior une pares de observaciones siempre que sus claves sean iguales:



(Para ser precisos, esto corresponde a una *unión de igualdad* (o *equijoin*) interior, debido a que las claves se unen usando el operador de igualdad. Dado que muchas uniones son uniones de igualdad, por lo general omitimos esa especificación.)

El output de una unión interior es un nuevo data frame que contiene la clave, los valores de x y los valores de y. Usamos `by` (*según*) para indicar a dplyr qué variable es la clave:

```
x %>%
  inner_join(y, by = "key")
#> # A tibble: 2 x 3
#>     key val_x val_y
#>   <dbl> <chr> <chr>
#> 1     1 x1    y1
#> 2     2 x2    y2
```

Copy

La propiedad más importante de una unión interior es que las filas no coincidentes no se incluyen en el resultado. Esto significa que generalmente las uniones interiores no son apropiadas para su uso en el análisis de datos dado que es muy fácil perder observaciones.

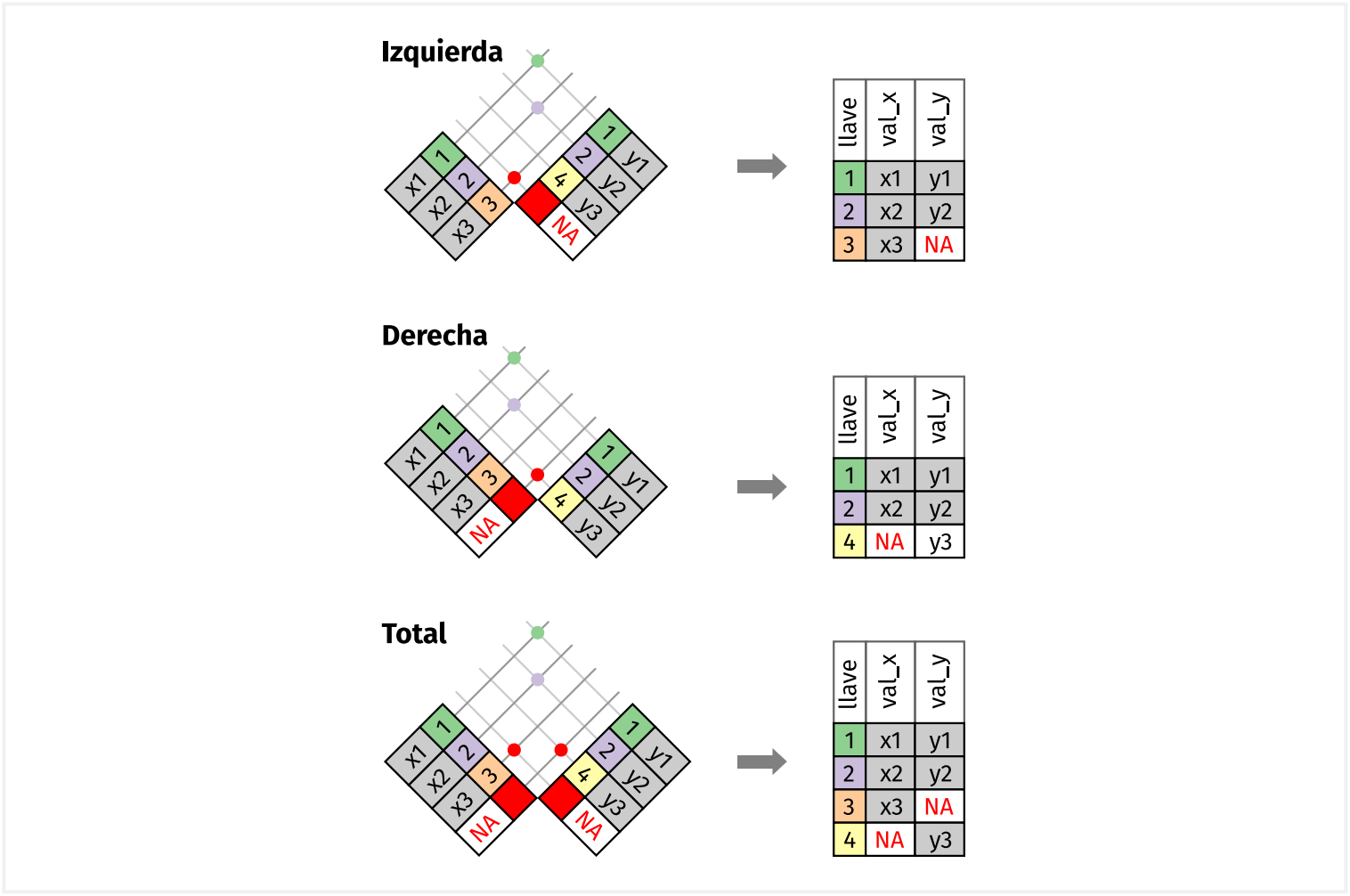
13.4.3 **Uniones exteriores**

Una unión interior mantiene las observaciones que aparecen en ambas tablas. Una **unión exterior** mantiene las observaciones que aparecen en al menos una de las tablas. Existen tres tipos de uniones exteriores:

- Una **unión izquierda** (*left join*) mantiene todas las observaciones en `x`.
- Una **unión derecha** (*right join*) mantiene todas las observaciones en `y`.
- Una **unión completa** (*full join*) mantiene todas las observaciones en `x` e `y`.

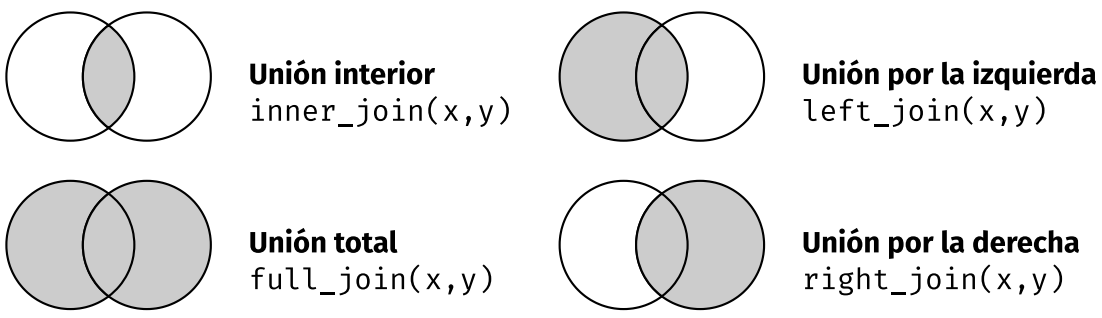
Estas uniones funcionan agregando una observación "virtual" adicional a cada tabla. Esta observación tiene una clave que siempre coincide (de no haber otras claves coincidentes) y un valor que se llena con `NA`.

Gráficamente corresponde a lo siguiente:



La unión que más frecuentemente se usa es la unión izquierda: úsala cuando necesites buscar datos adicionales en otra tabla, dado que preserva las observaciones originales incluso cuando no hay coincidencias. La unión izquierda debiera ser tu unión por defecto, a menos que tengas un motivo importante para preferir una de las otras.

Otra forma de ilustrar diferentes tipos de uniones es mediante un diagrama de Venn:

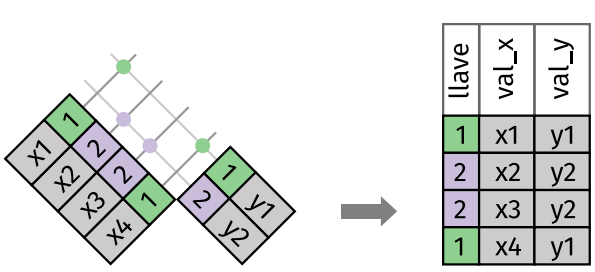


Sin embargo, esta no es una buena representación. Puede ayudar a recordar qué uniones preservan las observaciones en qué tabla pero esto tiene una limitante importante: un diagrama de Venn no puede mostrar qué ocurre con las claves que no identifican de manera única una observación.

13.4.4 **Claves duplicadas**

Hasta ahora todos los diagramas han asumido que las claves son únicas. Pero ese no siempre es así. Esta sección explica qué ocurre en esos casos. Existen dos posibilidades:

1. Una tabla tiene claves duplicadas. Esto es útil cuando quieres agregar información adicional dado que típicamente existe una relación uno a muchos.

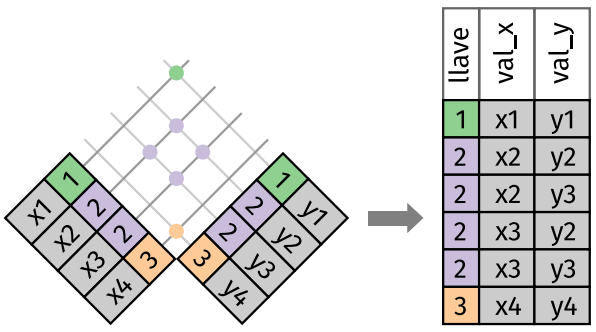


Nota que hemos puesto la columna clave en una posición ligeramente distinta en el output. Esto refleja que la clave es una clave primaria en `y` y una clave foránea en `x`.

```
x <- tribble(
 ~key, ~val_x,
 1, "x1",
 2, "x2",
 2, "x3",
 1, "x4"
)
y <- tribble(
 ~key, ~val_y,
 1, "y1",
 2, "y2"
)
left_join(x, y, by = "key")
#> # A tibble: 4 x 3
#>     key val_x val_y
#>   <dbl> <chr> <chr>
#> 1     1 x1    y1
#> 2     2 x2    y2
#> 3     2 x3    y2
#> 4     1 x4    y1
```

Copy

1. Ambas tablas tienen claves duplicadas. Esto es usualmente un error debido a que en ninguna de las tablas las claves identifican de manera única una observación. Cuando unes claves duplicadas, se obtienen todas las posibles combinaciones, es decir, el producto cartesiano:




```
 x <- tribble(
~key, ~val_x,
1, "x1",
2, "x2",
2, "x3",
3, "x4"
 )
 y <- tribble(
~key, ~val_y,
1, "y1",
2, "y2",
2, "y3",
3, "y4"
 )
 left_join(x, y, by = "key")
#> # A tibble: 6 x 3
#>     key val_x val_y
#>   <dbl> <chr> <chr>
#> 1     1 x1    y1
#> 2     2 x2    y2
#> 3     2 x2    y3
#> 4     2 x3    y2
#> 5     2 x3    y3
#> 6     3 x4    y4
```

Copy

13.4.5 Definiendo las columnas clave

Hasta ahora, los pares de tablas siempre se han unido de acuerdo a una única variable y esa variable tiene el mismo nombre en ambas tablas. Esta restricción se expresa de la forma `by = "key"`. Puedes usar otros valores de `by` para conectar las tablas de otras maneras:

- Por defecto, `by = NULL`, usa todas las variables que aparecen en ambas tablas, lo que se conoce como unión **natural**. Por ejemplo, las tablas vuelos y clima coinciden en sus variables comunes: `anio`, `mes`, `dia`, `hora` y `origen`.

```
vuelos2 %>%
 left_join(clima)
#> Joining, by = c("anio", "mes", "dia", "hora", "origen")
#> # A tibble: 336,776 x 18
#>    anio   mes   dia  hora origen destino codigo_cola aerolinea temperatura
#>   <int> <int> <int> <dbl> <chr>  <chr>   <chr>       <chr>           <dbl>
#> 1  2013     1     1     5 EWR    IAH     N14228      UA               39.0
#> 2  2013     1     1     5 LGA    IAH     N24211      UA               39.9
#> 3  2013     1     1     5 JFK    MIA     N619AA      AA               39.0
#> 4  2013     1     1     5 JFK    BQN     N804JB      B6               39.0
#> 5  2013     1     1     6 LGA    ATL     N668DN      DL               39.9
#> 6  2013     1     1     5 EWR    ORD     N39463      UA               39.0
#> # … with 336,770 more rows, and 9 more variables: punto_rocio <dbl>,
#> #   humedad <dbl>, direccion_viento <dbl>, velocidad_viento <dbl>,
#> #   velocidad_rafaga <dbl>, precipitacion <dbl>, presion <dbl>,
#> #   visibilidad <dbl>, fecha_hora <dttm>
```

- Un vector de caracteres, `by = "x"`. Esto es similar a una unión natural, pero usa algunas de las variables comunes. Por ejemplo, `vuelos` y `aviones` tienen la variable `anio`, pero esta significa cosas distintas en cada tabla por lo que queremos unir por `codigo_cola`.

```
vuelos2 %>%
 left_join(aviones, by = "codigo_cola")
#> # A tibble: 336,776 x 16
#>   anio.x   mes   dia  hora origen destino codigo_cola aerolinea anio.y tipo
#>    <int> <int> <int> <dbl> <chr>  <chr>   <chr>       <chr>      <int> <chr>
#> 1   2013     1     1     5 EWR    IAH     N14228      UA          1999 Ala …
#> 2   2013     1     1     5 LGA    IAH     N24211      UA          1998 Ala …
#> 3   2013     1     1     5 JFK    MIA     N619AA      AA          1990 Ala …
#> 4   2013     1     1     5 JFK    BQN     N804JB      B6          2012 Ala …
#> 5   2013     1     1     6 LGA    ATL     N668DN      DL          1991 Ala …
#> 6   2013     1     1     5 EWR    ORD     N39463      UA          2012 Ala …
#> # … with 336,770 more rows, and 6 more variables: fabricante <chr>,
#> #   modelo <chr>, motores <int>, asientos <int>, velocidad <int>,
#> #   tipo_motor <chr>
```

Nota que la variable `anio` (que aparece en los dos data frames de entrada, pero que no es igual en ambos casos) se desambigua con un sufijo en el output.

- Un vector de caracteres con nombres: `by = c("a" = "b")`. Esto va a unir la variable `a` en la tabla `x` con la variabla `b` en la tabla `y`. Las variables de `x` se usarán en el output.

Por ejemplo, si queremos dibujar un mapa necesitamos combinar los datos de vuelos con los datos de aeropuertos, que contienen la ubicación de cada uno (`latitud` y `longitud`). Cada vuelo tiene un `aeropuerto` de origen y destino, por lo que necesitamos especficar cuál queremos unir:

```
vuelos2 %>%
 left_join(aeropuertos, c("origen" = "codigo_aeropuerto"))
#> # A tibble: 336,776 x 15
#>    anio   mes   dia  hora origen destino codigo_cola aerolinea nombre latitud
#>   <int> <int> <int> <dbl> <chr>  <chr>   <chr>       <chr>     <chr>    <dbl>
#> 1  2013     1     1     5 EWR    IAH     N14228      UA        Newar…    40.7
#> 2  2013     1     1     5 LGA    IAH     N24211      UA        La Gu…    40.8
#> 3  2013     1     1     5 JFK    MIA     N619AA      AA        John …    40.6
#> 4  2013     1     1     5 JFK    BQN     N804JB      B6        John …    40.6
#> 5  2013     1     1     6 LGA    ATL     N668DN      DL        La Gu…    40.8
#> 6  2013     1     1     5 EWR    ORD     N39463      UA        Newar…    40.7
#> # … with 336,770 more rows, and 5 more variables: longitud <dbl>, altura <dbl>,
#> #   zona_horaria <dbl>, horario_verano <chr>, zona_horaria_iana <chr>

vuelos2 %>%
 left_join(aeropuertos, c("destino" = "codigo_aeropuerto"))
#> # A tibble: 336,776 x 15
#>    anio   mes   dia  hora origen destino codigo_cola aerolinea nombre latitud
#>   <int> <int> <int> <dbl> <chr>  <chr>   <chr>       <chr>     <chr>    <dbl>
#> 1  2013     1     1     5 EWR    IAH     N14228      UA        Georg…    30.0
#> 2  2013     1     1     5 LGA    IAH     N24211      UA        Georg…    30.0
#> 3  2013     1     1     5 JFK    MIA     N619AA      AA        Miami…    25.8
#> 4  2013     1     1     5 JFK    BQN     N804JB      B6        <NA>      NA
#> 5  2013     1     1     6 LGA    ATL     N668DN      DL        Harts…    33.6
#> 6  2013     1     1     5 EWR    ORD     N39463      UA        Chica…    42.0
#> # … with 336,770 more rows, and 5 more variables: longitud <dbl>, altura <dbl>,
#> #   zona_horaria <dbl>, horario_verano <chr>, zona_horaria_iana <chr>
```

13.4.6 Ejercicios

1. Calcula el atraso promedio por destino y luego une los datos en `aeropuertos` para que puedas mostrar la distribución espacial de los atrasos. Esta es una forma fácil de dibujar un mapa de los Estados Unidos:

```
aeropuertos %>%
semi_join(vuelos, c("codigo_aeropuerto" = "destino")) %>%
ggplot(aes(longitud, latitud)) +
borders("state") +
geom_point() +
coord_quickmap()
```

(No te preocupes si no entiendes quá hace `semi_join()`; lo aprenderás a continuación.)

Quizás quieras usar `size` o `colour` para editar los puntos y mostrar el atraso promedio de cada aeropuerto.

2. Agrega la ubicación de origen *y* destino (por ejemplo, `latitud` y `longitud`) a `vuelos`.

3. ¿Existe una relación entre la antiguedad de un avión y sus atrasos?

4. ¿Qué condiciones climáticas hacen más probables los atrasos?

5. ¿Qué sucedió el día 13 de junio de 2013? Muestra el patrón espacial de los atrasos. Luego, usa un buscador para encontrar referencias cruzadas con el clima.

13.4.7 Otras implementaciones

`base::merge()` puede realizar los cuatro tipos de uniones de transformación:

dplyr	merge
`inner_join(x, y)`	`merge(x, y)`
`left_join(x, y)`	`merge(x, y, all.x = TRUE)`
`right_join(x, y)`	`merge(x, y, all.y = TRUE)`,
`full_join(x, y)`	`merge(x, y, all.x = TRUE, all.y = TRUE)`

La ventaja de los verbos específicos de **dplyr** es que muestran de manera clara la intención del código: la diferencia entre las uniones es realmente importante pero se esconde en los argumentos de `merge()`. Las uniones de **dplyr** son considerablemente más rápidas y no generan problemas con el orden de las filas

SQL es una inspiración para las convenciones de **dplyr**, por lo que su traducción es directa:

dplyr	SQL
`inner_join(x, y, by = "z")`	`SELECT * FROM x INNER JOIN y USING (z)`
`left_join(x, y, by = "z")`	`SELECT * FROM x LEFT OUTER JOIN y USING (z)`
`right_join(x, y, by = "z")`	`SELECT * FROM x RIGHT OUTER JOIN y USING (z)`
`full_join(x, y, by = "z")`	`SELECT * FROM x FULL OUTER JOIN y USING (z)`

Nota que “INNER” y “OUTER” son opcionales, por lo que a menudo se omiten.

Unir distintas variables entre tablas, por ejemplo `inner_join(x, y, by = c("a" = "b"))`, usa una sintaxis ligeramente distinta en SQL: `SELECT * FROM x INNER JOIN y ON x.a = y.b`. Como la sintaxis sugiere, SQL soporta un rango más amplio de tipos de uniones que **dplyr**, ya que puedes conectar tablas usando restricciones distintas a las de igualdad (a veces llamadas de no-igualdad).

13.5 Uniones de filtro

Las uniones de filtro unen observaciones de la misma forma que las uniones de transformación pero afectan a las observaciones, no a las variables. Existen dos tipos:

- `semi_join(x, y)` **mantiene** todas las observaciones en `x` con coincidencias en `y`.
- `anti_join(x, y)` **descarta** todas las observaciones en `x` con coincidencias en `y`.

Las semi uniones son útiles para unir tablas resumen previamente filtradas con las filas originales. Por ejemplo, imagina que encontraste los diez destinos más populares:

```
destinos_populares <- vuelos %>%
  count(destino, sort = TRUE) %>%
  head(10)
destinos_populares
#> # A tibble: 10 x 2
#>   destino     n
#>   <chr>   <int>
#> 1 ORD     17283
#> 2 ATL     17215
#> 3 LAX     16174
#> 4 BOS     15508
#> 5 MCO     14082
#> 6 CLT     14064
#> # … with 4 more rows
```

Ahora quieres encontrar cada vuelo que fue a alguno de esos destinos. Puedes construir un filtro:

```
vuelos %>%
  filter(destino %in% destinos_populares$destino)
#> # A tibble: 141,145 x 19
#>    anio   mes   dia horario_salida salida_programa… atraso_salida
#>   <int> <int> <int>          <int>            <int>         <dbl>
#> 1  2013     1     1            542              540             2
#> 2  2013     1     1            554              600            -6
#> 3  2013     1     1            554              558            -4
#> 4  2013     1     1            555              600            -5
#> 5  2013     1     1            557              600            -3
#> 6  2013     1     1            558              600            -2
#> # … with 141,139 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>
```

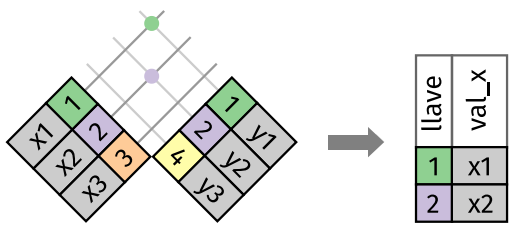
Pero es difícil extender este enfoque a varias variables. Por ejemplo, imagina que encontraste los diez días con los atrasos promedio más altos. ¿Cómo construirías un filtro que use `anio`, `mes` y `dia` para buscar coincidencias con `vuelos`?

Puedes, en cambio, usar `semi_join()`, que conecta dos tablas de manera similar a una unión de transformación, pero en lugar de agregar nuevas columnas, mantiene las filas en `x` que tienen coincidencias en `y`:

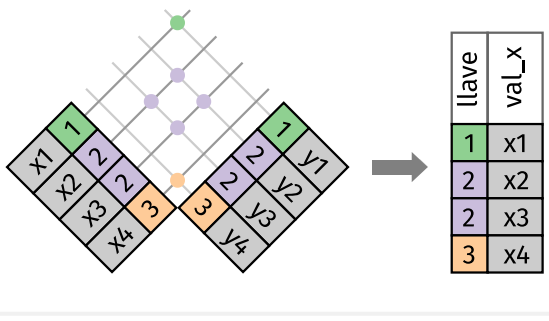
```
vuelos %>%
  semi_join(destinos_populares)
#> Joining, by = "destino"
#> # A tibble: 141,145 x 19
#>    anio   mes   dia horario_salida salida_programa… atraso_salida
#>   <int> <int> <int>          <int>            <int>         <dbl>
#> 1  2013     1     1            542              540             2
#> 2  2013     1     1            554              600            -6
#> 3  2013     1     1            554              558            -4
#> 4  2013     1     1            555              600            -5
#> 5  2013     1     1            557              600            -3
#> 6  2013     1     1            558              600            -2
#> # … with 141,139 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>
```

Copy

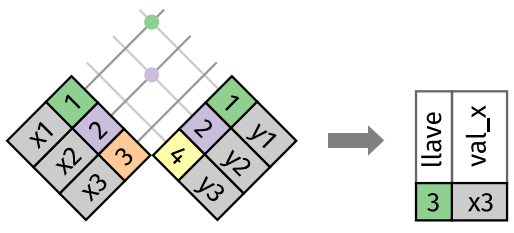
Gráficamente, un `semi_join()` se ve de la siguiente forma:



Solo la existencia de coincidencias es importante; da igual qué observaciones son coincidentes. Esto significa que las uniones de filtro nunca duplican filas como lo hacen las uniones de transformación:



La operación inversa de `semi_join()` es `anti_join()`. `anti_join()` mantiene las filas que *no* tienen coincidencias:



Las anti uniones son útiles para encontrar desajustes. Por ejemplo, al conectar `aviones` y `vuelos`, podría interesarte saber que existen muchos `vuelos` que no tienen coincidencias en `aviones`:

```
vuelos %>%
  anti_join(aviones, by = "codigo_cola") %>%
  count(codigo_cola, sort = TRUE)
#> # A tibble: 722 x 2
#>   codigo_cola     n
#>   <chr>       <int>
#> 1 <NA>         2512
#> 2 N725MQ        575
#> 3 N722MQ        513
#> 4 N723MQ        507
#> 5 N713MQ        483
#> 6 N735MQ        396
#> # … with 716 more rows
```

Copy

13.5.1 Ejercicios

1. ¿Qué significa que a un vuelo le falte `codigo_cola` ? ¿Qué tienen en común los códigos de cola que no tienen registros coincidentes en `aviones` ? (Pista: una variable explica ~90% de los problemas.)

2. Filtra los vuelos para mostrar únicamente los aviones que han realizado al menos cien viajes.

3. Combina `datos::vehiculos` y `datos::comunes` para encontrar los registros de los modelos más comunes.

4. Encuentra las 48 horas (en el transcurso del año) que tengan los peores atrasos. Haz una referencia cruzada con la tabla `clima` . ¿Puedes observar patrones?

5. ¿Qué te indica `anti_join(vuelos, aeropuertos, by = c("destino" = "codigo_aeropuerto"))` ? ¿Qué te indica `anti_join(aeropuertos, vuelos, by = c("codigo_aeropuerto" = "destino"))` ?

6. Puedes esperar que exista una relación implícita entre aviones y aerolíneas, dado que cada avión es operado por una única aerolínea. Confirma o descarta esta hipótesis usando las herramientas que aprendiste más arriba.

13.6 Problemas con las uniones

Los datos con los que has estado trabajando en este capítulo han sido limpiados de modo que tengas la menor cantidad de problemas posibles. Tus datos difícilmente estarán tan ordenados, por lo que hay algunas consideraciones y pasos a tener en cuenta para que las uniones funcionen adecuadamente sobre tus propios datos.

1. Comienza identificando las variables que forman las claves primarias en cada tabla. Usualmente debieras hacerlo considerando tus conocimientos de los datos, no observando empíricamente las combinaciones de variables que resultan en un identificador único. Si te centras en las variables sin pensar en sus significados, puedes tener la (mala) suerte de encontrar una combinación única en tus datos pero que no sea válida en general.

   Por ejemplo, la altura y la longitud identifican de manera única cada aeropuerto, ¡pero no son buenos identificadores!

```
  aeropuertos %>% count(altura, longitud) %>% filter(n > 1)
#> # A tibble: 0 x 3
#> # … with 3 variables: altura <dbl>, longitud <dbl>, n <int>
```

Copy

2. Verifica que ninguna de las variables en la clave primaria esté perdida. ¡Si hay un valor faltante no podrá identificar una observación!

3. Verifica que las claves foráneas coincidan con las claves primarias en otra tabla. La mejor forma de hacerlo es mediante un `anti_join()` . Es común que las claves no coincidan debido a errores en la entrada de datos. Arreglar este problema requiere mucho trabajo.

   Si tienes claves perdidas, debes tener cuidado en el uso de unión interior versus unión exterior y considerar cuidadosamente si quieres descartar las filas que no tengan coincidencias.

Ten en cuenta que verificar el número de filas antes y después de unir no es suficiente para asegurar que la unión funcionó de forma exitosa. Si tienes una unión interior con claves duplicadas en ambas tablas, puedes tener la mala suerte de que el número de filas descartadas sea igual al número de filas duplicadas.

13.7 Operaciones de conjuntos

El tipo final de verbo para dos tablas son las operaciones de conjunto. Si bien lo usamos de manera poco frecuente, en ocasiones es útil cuando quieres dividir un filtro complejo en partes maś simples. Todas estas operaciones funcionan con una fila completa, comparando los valores de cada variable. Esto espera que los input `x` e `y` tengan las mismas variables y trata las observaciones como conjuntos:

- `intersect(x, y)` : devuelve las observaciones comunes en `x` e `y` .
- `union(x, y)` : devuelve las observaciones únicas en `x` e `y` .
- `setdiff(x, y)` : devuelve las observaciones en `x` pero no en `y` .

Dados los siguientes datos simples:

```
df1 <- tribble(
  ~x, ~y,
  1, 1,
  2, 1
)
df2 <- tribble(
  ~x, ~y,
  1, 1,
  1, 2
)
```

Las cuatro posibilidades son:

```
intersect(df1, df2)
#> # A tibble: 1 x 2
#>       x     y
#>   <dbl> <dbl>
#> 1     1     1

# Nota que obtenemos 3 filas, no 4
union(df1, df2)
#> # A tibble: 3 x 2
#>       x     y
#>   <dbl> <dbl>
#> 1     1     1
#> 2     2     1
#> 3     1     2

setdiff(df1, df2)
#> # A tibble: 1 x 2
#>       x     y
#>   <dbl> <dbl>
#> 1     2     1

setdiff(df2, df1)
#> # A tibble: 1 x 2
#>       x     y
#>   <dbl> <dbl>
#> 1     1     2
```

« 12 Datos ordenados

14 Cadenas de caracteres »

"" was written by .

This book was built by the bookdown R package.

14 Cadenas de caracteres

14.1 Introducción

Este capítulo te introduce en la manipulación de cadenas de caracteres en R. Si bien aprenderás los aspectos básicos acerca de cómo funcionan y cómo crearlas a mano, el foco del capítulo estará puesto en las expresiones regulares (o *regex*). Como las cadenas de caracteres suelen contener datos no estructurados o semiestructurados, las expresiones regulares resultan útiles porque permiten describir patrones en ellas a través de un lenguaje conciso. Cuando mires por primera vez una expresión regular te parecerá que un gato caminó sobre tu teclado, pero a medida que vayas ampliando tu conocimiento pronto te empezarán a hacer sentido.

14.1.1 Prerequisitos

Este capítulo se enfocará en el paquete para manipulación de cadenas llamado **stringr** (del inglés *string*, cadena), que es parte del Tidyverse.

```
library(tidyverse)
library(datos)
```

Copy

14.2 Cadenas: elementos básicos

Puedes crear una cadena utilizando comillas simples o dobles. A diferencia de otros lenguajes, no hay diferencias en su comportamiento. Nuestra recomendación es siempre utilizar `"`, a menos que quieras crear una cadena que contenga múltiples `"`.

```
string1 <- "Esta es una cadena de caracteres"
string2 <- 'Si quiero incluir "comillas" dentro de la cadena, uso comillas simples'
```

Copy

Si olvidas cerrar las comillas, verás un `+` en la consola, que es el signo de continuación para indicar que el código no está completo:

```
> "Esta es una cadena de caracteres sin comillas de cierre
+
+
+ AYUDA, ESTOY ATASCADO
```

Copy

Si esto te ocurre, ¡presiona la tecla *Escape* e inténtalo de nuevo!

Para incluir comillas simples o dobles de manera literal en una cadena puedes utilizar `\` para “escaparlas” (“escapar” viene de la tecla *escape*):

```
comilla_doble <- "\"" # o '"'
comilla_simple <- '\'' # o "'"
```

Copy

Esto quiere decir que si quieres incluir una barra invertida, necesitas duplicarla: `"\\"`.

Ten cuidado con el hecho de que la representación impresa de una cadena no es equivalente a la cadena misma, ya que la representación muestra las barras utilizadas para “escapar” caracteres, es decir, para sean interpretados en su sentido literal, no como caracteres especiales. Para ver el contenido en bruto de una cadena utiliza `writeLines()`:

On this page

View source
Edit this page

```
x <- c("\"", "\\")
x
#> [1] "\"" "\\"
writeLines(x)
#> "
#> \
```

Existe una serie de otros caracteres especiales. Los más comunes son `"\n"`, para salto de línea, y `"\t"`, para tabulación. Puedes ver la lista completa pidiendo ayuda acerca de `"`: `?'"'` o `?"'"`. A veces también verás cadenas del tipo `"\u00b5"`, que es la manera de escribir caracteres que no están en inglés para que funcionen en todas las plataformas:

```
x <- "\u00b5"
x
#> [1] "µ"
```

Usualmente se guardan múltiples cadenas en un vector de caracteres. Puedes crearlo usando `c()`:

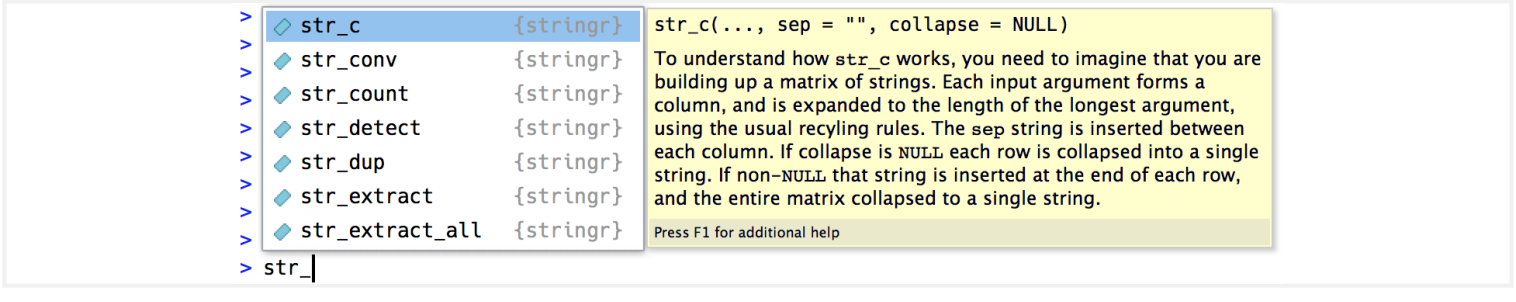
```
c("uno", "dos", "tres")
#> [1] "uno"  "dos"  "tres"
```

14.2.1 Largo de cadena

R base tiene muchas funciones para trabajar con cadenas de caracteres, pero las evitaremos porque pueden ser incosistentes, lo que hace que sean difíciles de recordar. En su lugar, utilizaremos funciones del paquete **stringr**. Estas tienen nombres más intuitivos y todas empiezan con `str_`. Por ejemplo, `str_length()` te dice el número de caracteres de una cadena (*length* en inglés es *largo*):

```
str_length(c("a", "R para Ciencia de Datos", NA))
#> [1]  1 23 NA
```

El prefijo común `str_` es particularmente útil si utilizas RStudio, ya que al escribir `str_` se activa el autocompletado, lo que te permite ver todas las funciones de **stringr**:



14.2.2 Combinar cadenas

Para combinar dos o más cadenas utiliza `str_c()`:

```
str_c("x", "y")
#> [1] "xy"
str_c("x", "y", "z")
#> [1] "xyz"
```

Usa el argumento `sep` para controlar cómo separlas:

```
str_c("x", "y", sep = ", ")
#> [1] "x, y"
```

Al igual que en muchas otras funciones de R, los valores faltantes son contagiosos. Si quieres que se impriman como `"NA"`, utiliza `str_replace_na()` (*replace* = remplazar):

```
x <- c("abc", NA)
str_c("|-", x, "-|")
#> [1] "|-abc-|" NA
str_c("|-", str_replace_na(x), "-|")
#> [1] "|-abc-|" "|-NA-|"
```

Como se observa arriba, `str_c()` es una función vectorizada que automáticamente recicla los vectores más cortos hasta alcanzar la extensión del más largo:

```
str_c("prefijo-", c("a", "b", "c"), "-sufijo")
#> [1] "prefijo-a-sufijo" "prefijo-b-sufijo" "prefijo-c-sufijo"
```

Copy

Los objetos de extensión 0 se descartan de manera silenciosa. Esto es particularmente útil en conjunto con `if` (*si*):

```
nombre <- "Hadley"
hora_del_dia <- "mañana"
cumpleanios <- FALSE

str_c(
  "Que tengas una buena ", hora_del_dia, ", ", nombre,
  if (cumpleanios) " y ¡FELIZ CUMPLEAÑOS!",
  "."
)
#> [1] "Que tengas una buena mañana, Hadley."
```

Copy

Para colapsar un vector de cadenas en una sola, utiliza `collapse`:

```
str_c(c("x", "y", "z"), collapse = ", ")
#> [1] "x, y, z"
```

Copy

14.2.3 Dividir cadenas

Puedes extraer partes de una cadena utilizando `str_sub()`. Al igual que la cadena, `str_sub()` tiene como argumentos `start` (*inicio*) y `end` (*fin*), que indican la posición (inclusiva) del subconjunto que se quiere extraer:

```
x <- c("Manzana", "Plátano", "Pera")
str_sub(x, 1, 3)
#> [1] "Man" "Plá" "Per"
# los números negativos cuentan de manera invertida desde el final
str_sub(x, -3, -1)
#> [1] "ana" "ano" "era"
```

Copy

Ten en cuenta que `str_sub()` no fallará si la cadena es muy corta; simplemente devolverá todo lo que sea posible:

```
str_sub("a", 1, 5)
#> [1] "a"
```

Copy

También puedes utilizar `str_sub()` en forma de asignación para modificar una cadena:

```
str_sub(x, 1, 1) <- str_to_lower(str_sub(x, 1, 1))
x
#> [1] "manzana" "plátano" "pera"
```

Copy

14.2.4 *Locales*

Arriba utilizamos `str_to_lower()` para cambiar el texto a minúsculas. También puedes utilizar `str_to_upper()` o `str_to_title()`, si quieres modificar el texto a mayúsculas o formato título, respectivamente. Sin embargo, este tipo de cambios puede ser más complicado de lo parece a primera vista, ya que las reglas no son iguales en todos los idiomas. Puedes selecionar qué tipo de reglas aplicar especificando el entorno local o *locale*:

```
# La lengua turca tiene dos i: una con punto y otra sin punto
# Tienen diferentes reglas para convertirlas en mayúsculas

str_to_upper(c("i", "ı"))
#> [1] "I" "I"
str_to_upper(c("i", "ı"), locale = "tr")
#> [1] "İ" "I"
```

El entorno local o *locale* se especifica con un código de idioma ISO 639, que es una abreviación de dos letras. Si todavía no conoces el código para tu idioma, en Wikipedia puedes encontrar una buena lista. Si dejas el *locale* en blanco, se aplicará el que estés utilizando actualmente, que es provisto por tu sistema operativo.

Otra operación importante que es afectada por el *locale* es ordenar. Las funciones `order()` y `sort()` de R base ordenan las cadenas usando el *locale* actual. Si quieres un comportamiento consistente a través de diferentes computadoras, sería preferible usar `str_sort()` y `str_order()`, que aceptan un argumento adicional para definir el `locale`:

```
x <- c("arándano", "espinaca", "banana")

str_sort(x, locale = "es")  # Español
#> [1] "arándano" "banana"   "espinaca"

str_sort(x, locale = "haw") # Hawaiano
#> [1] "arándano" "espinaca" "banana"
```

14.2.5 Ejercicios

1. En ejemplos de código en los que no se utiliza **stringr**, verás usualmente `paste()` y `paste0()` (*paste* = pegar). ¿Cuál es la diferencia entre estas dos funciones? ¿A qué función de **stringr** son equivalentes? ¿Cómo difieren estas dos funciones respecto de su manejo de los `NA`?

2. Describe con tus propias palabras la diferencia entre los argumentos `sep` y `collapse` de la función `str_c()`.

3. Utiliza `str_length()` y `str_sub()` para extraer el caracter del medio de una cadena. ¿Qué harías si el número de caracteres es par?

4. ¿Qué hace `str_wrap()`? (*wrap* = envolver) ¿Cuándo podrías querer utilizarla?

5. ¿Qué hace `str_trim()`? (*trim* = recortar) ¿Cuál es el opuesto de `str_trim()`?

6. Escribe una función que convierta, por ejemplo, el vector `c("a", "b", "c")` en la cadena `a, b y c`. Piensa con detención qué debería hacer dado un vector de largo 0, 1 o 2.

14.2.6 Buscar coincidencia de patrones con expresiones regulares

Las expresiones regulares son un lenguaje conciso que te permite describir patrones en cadenas de caracteres. Toma un tiempo entenderlas, pero una vez que lo hagas te darás cuenta que son extremadamente útiles.

Para aprender sobre expresiones regulares usaremos `str_view()` y `str_view_all()` (*view* = ver). Estas funciones toman un vector de caracteres y una expresión regular y te muestran cómo coinciden. Partiremos con expresiones regulares simples que gradualmente se irán volviendo más y más complejas. Una vez que domines la coincidencia de patrones, aprenderás cómo aplicar estas ideas con otras funciones de **stringr**.

14.2.7 Coincidencias básicas

Los patrones más simples buscan coincidencias con cadenas exactas:

```
x <- c("manzana", "banana", "pera")
str_view(x, "an")
```

manzana

```
banana
```

```
pera
```

El siguiente paso en complejidad es `.`, que coincide con cualquier caracter (excepto un salto de línea):

```
str_view(x, ".a.")
```

Copy

```
manzana
```

```
banana
```

```
pera
```

Pero si "`.`" coincide con cualquier caracter, ¿cómo buscar una coincidencia con el caracter "`.`"? Necesitas utilizar un "escape" para decirle a la expresión regular que quieres hacerla coincidir de manera exacta, no usar su comportamiento especial. Al igual que en las cadenas, las expresiones regulares usan la barra invertida, `\`, para "escapar" los comportamientos especiales. Por lo tanto, para hacer coincidir un `.`, necesitas la expresión regular `\.`. Lamentablemente, esto crea una problema. Estamos usando cadenas para representar una expresión regular y en ellas `\` también se usa como símbolo de "escape". Por lo tanto, para crear la expresión regular `\.` necesitamos la cadena `"\\."`.

```
# Para crear una expresión regular necesitamos \\
punto <- "\\."

# Pero la expresión en sí misma solo contiene una \
writeLines(punto)
#> \.

# Esto le dice a R que busque el . de manera explícita
str_view(c("abc", "a.c", "bef"), "a\\.c")
```

Copy

```
abc
```

```
a.c
```

```
bef
```

Si `\` se utiliza para escapar un caracter en una expresión regular, ¿cómo coincidir de manera literal una `\`? Bueno, necesitarías escaparla creando la expresión regular `\\`. Para crear esa expresión regular necesitas usar una cadena, que requiere también escapar la `\`. Esto quiere decir que para coincidir literalmente `\` necesitas escribir `"\\\\"` — ¡necesitas cuatro barras invertidas para coincidir una!

```
x <- "a\\b"
writeLines(x)
#> a\b

str_view(x, "\\\\")
```

Copy

```
a\b
```

En este libro escribiremos las expresiones regulares como `\.` y las cadenas que representan a las expresiones regulares como `"\\."`.

14.2.7.1 Ejercicios

1. Explica por qué cada una de estas cadenas no coincide con `\`: `"\"`, `"\\"`, `"\\\"`.

2. ¿Cómo harías coincidir la secuencia `"'\`?

3. ¿Con qué patrones coincidiría la expresión regular `\..\..\..`? ¿Cómo la representarías en una cadena?

14.2.8 Anclas

Por defecto, las expresiones regulares buscarán una coincidencia en cualquier parte de una cadena. Suele ser útil *anclar* una expresión regular para que solo busque coincidencias al inicio o al final. Puedes utilizar:

- `^` para buscar la coincidencia al inicio de la cadena.
- `$` para buscar la coincidencia al final de la cadena.

```
x <- c("arándano", "banana", "pera")
str_view(x, "^a")
```

Copy

arándano

banana

pera

str_view(x, "a$")

arándano

banana

pera

Para recordar cuál es cuál, puedes intentar este recurso mnemotécnico que aprendimos de Evan Misshula: si empiezas con potencia (`^`), terminarás con dinero (`$`).

Para forzar que una expresión regular coincida con una cadena completa, ánclala usando tanto `^` como `$` :

```
x <- c("pie de manzana", "manzana", "queque de manzana")
str_view(x, "manzana")
```

Copy

pie de manzana

manzana

queque de manzana

str_view(x, "^manzana$")

pie de manzana

manzana

queque de manzana

También puedes coincidir el límite entre palabras con `\b` . No utilizamos frecuentemente esta forma en R, pero sí a veces cuando buscamos en RStudio el nombre de una función que también compone el nombre de otras funciones. Por ejemplo, buscaríamos `\bsum\b` para evitar la coincidencia con `summarise` , `summary` , `rowsum` y otras.

14.2.8.1 **Ejercicios**

1. ¿Cómo harías coincidir la cadena `"$^$"` de manera literal?
2. Dado el corpus de palabras comunes en `datos::palabras` , crea una expresión regular que busque palabras que:
    1. Empiecen con “y”.
    2. Terminen con “z”
    3. Tengan una extensión de exactamente tres letras. (¡No hagas trampa usando `str_length()` !)
    4. Tengan ocho letras o más.

    Dado que esta será una lista larga, podrías querer usar el argumento `match` en `str_view()` para mostrar solo las palabras que coincidan o no coincidan.

14.2.9 **Clases de caracteres y alternativas**

Existe una serie de patrones especiales que coinciden con más de un caracter. Ya has visto `.` , que coincide con cualquier caracter excepto un salto de línea. Hay otras cuatro herramientas que son de utilidad:

- `\d` : coincide con cualquier dígito.
- `\s` : coincide con cualquier espacio en blanco (por ejemplo, espacio simple, tabulador, salto de línea).
- `[abc]` : coincide con a, b o c.
- `[^abc]` : coincide con todo menos con a, b o c.

Recuerda que para crear una expresión regular que contenga `\d` o `\s` necesitas escapar la `\` en la cadena, por lo que debes escribir `"\\d"` o `"\\s"` .

Utilizar una clase de caracter que contenga en su interior un solo caracter puede ser una buena alternativa a la barra invertida cuando quieres incluir un solo metacaracter en la expresión regular. Muchas personas encuentran que así es más fácil de leer.

```
# Buscar de forma literal un caracter que usualmente tiene un significado especial en una expresión regular
str_view(c("abc", "a.c", "a*c", "a c"), "a[.]c")
```

Copy

abc

a.c

a*c

a c

str_view(c("abc", "a.c", "a*c", "a c"), ".[*]c")

abc

a.c

a*c

a c

str_view(c("abc", "a.c", "a*c", "a c"), "a[ ]")

abc

a.c

a*c

a c

Esto funciona para la mayoría (pero no para todos) los metacaracteres de las expresiones regulares: `$ . | ? * + ( ) [ {`. Desafortunadamente, existen unos pocos caracteres que tienen un significado especial incluso dentro de una clase de caracteres y deben manejarse con barras invertidas para escaparlos: `] \ ^` y `-`.

Puedes utiizar una *disyunción* para elegir entre uno más patrones alternativos. Por ejemplo, `abc|d..a` concidirá tanto con '"abc"', como con `"duna"`. Ten en cuenta que la precedencia de `|` es baja, por lo que `abc|xyz` coincidirá con `abc` o `xyz`, no con `abcyz` o `abxyz`. Al igual que en expresiones matemáticas, si el valor de `|` se vuelve confuso, utiliza paréntesis para dejar claro qué es lo que quieres:

```
str_view(c("cómo", "como"), "c(ó|o)mo")
```

Copy

cómo

como

14.2.9.1 Ejercicios

1. Crea una expresión regular que encuentre todas las palabras que:
    1. Empiecen con una vocal.
    2. Solo contengan consonantes. (Pista: piensa en cómo buscar coincidencias para "no"-vocales.)
    3. Terminen en `ón`, pero no en `ión`.
    4. Terminen con `ndo` o `ado`.
2. ¿Siempre a una "q" la sigue una "u"?
3. Escribe una expresión regular que permita buscar un verbo que haya sido escrito usando voseo en segunda persona plural
   (por ejemplo, *queréis* en vez de *quieren*).
4. Crea una expresión regular que coincida con la forma en que habitualmente se escriben los números de teléfono en tu país.
5. En inglés existe una regla que dice que la letra i va siempre antes de la e, excepto cuando está después de una c". Verifica empíricamente esta regla utilizando las palabras contenidas en `stringr::words`.

14.2.10 Repetición

El siguiente paso en términos de poder implica controlar cuántas veces queremos que se encuentre un patrón:

- `?`: 0 o 1

- `+`: 1 o más
- `*`: 0 o más

```
x <- "1888 es el año más largo en números romanos: MDCCCLXXXVIII"
str_view(x, "CC?")
```

Copy

```
1888 es el año más largo en números romanos: MDCCCLXXXVIII
```

str_view(x, "CC+")

```
1888 es el año más largo en números romanos: MDCCCLXXXVIII
```

str_view(x, 'C[LX]+')

```
1888 es el año más largo en números romanos: MDCCCLXXXVIII
```

Ten en cuenta que la precedencia de este operador es alta, por lo que puedes escribir: `cantái?s` para encontrar tanto voseo americano como de la variante peninsular del español (es decir, *cantás* y *cantáis*). Esto quiere decir que en la mayor parte de los usos se necesitarán paréntesis, como `bana(na)+`.

También puedes especificar el número de coincidencias que quieres encontrar de manera precisa:

- `{n}`: exactamente n
- `{n,}`: n o más
- `{,m}`: no más de m
- `{n,m}`: entre n y m

```
str_view(x, "C{2}")
```

Copy

```
1888 es el año más largo en números romanos: MDCCCLXXXVIII
```

str_view(x, "C{2,}")

```
1888 es el año más largo en números romanos: MDCCCLXXXVIII
```

str_view(x, "C{2,3}")

```
1888 es el año más largo en números romanos: MDCCCLXXXVIII
```

Por defecto, este tipo de coincidencias son “avaras” (*greedy*): tratarán de coincidir con la cadena más larga posible. También puedes hacerlas “perezosas” (*lazy*) para que coincidan con la cadena más corta posible, poniendo un `?` después de ellas. Esta es una característica avanzada de las expresiones regulares, pero es útil saber que existe:

```
str_view(x, 'C{2,3}?')
```

Copy

```
1888 es el año más largo en números romanos: MDCCCLXXXVIII
```

str_view(x, 'C[LX]+?')

```
1888 es el año más largo en números romanos: MDCCCLXXXVIII
```

14.2.10.1 **Ejercicios**

1. Describe los equivalentes de `?`, `+`, `*` en el formato `{m,n}`.
2. Describe en palabras con qué coincidiría cada una de estas expresiones regulares: (lee con atención para ver si estamos utilizando una expresión regular o una cadena que define una expresión regular.)
    1. `^.*$`
    2. `"\\{.+\\}"`
    3. `\d{4}-\d{2}-\d{2}`
    4. `"\\\\{4}"`
3. Crea expresiones regulares para buscar todas las palabras que:
    1. Empiecen con dos consonantes.
    2. Tengan tres o más vocales seguidas.
    3. Tengan tres o más pares de vocal-consonante seguidos.

14.2.11 **Agrupamiento y referencias previas**

Anteriormente aprendiste sobre el uso de paréntesis para desambiguar expresiones complejas. Los paréntesis también sirven para crear un grupo de captura *numerado* (número 1, 2, etc.). Un grupo de captura guarda *la parte de la cadena* que coincide con la parte de la expresión regular entre paréntesis. Puedes referirte al mismo texto tal como fue guardado en un grupo de captura utilizando *referencias previas*, como `\1`, `\2` etc. Por ejemplo, la siguiente expresión regular busca todas las frutas que tengan un par de letras repetido.

```
str_view(frutas, "(..)\\1", match = TRUE)
```

Copy

banana

papaya

ananá

coco

(En breve, también verás cómo esto es útil en conjunto con `str_match()`.)

14.2.11.1 Ejercicios

1. Describe en palabras con qué coinciden estas expresiones:
    1. `(.)\1\1`
    2. `"(.)(.)\\2\\1"`
    3. `(..)\1`
    4. `"(.).\\1.\\1"`
    5. `"(.)(.)(.).*\\3\\2\\1"`
2. Construye una expresión regular que coincida con palabras que:
    1. Empiecen y terminen con el mismo caracter.
    2. Contengan un par de letras repetido (p. ej. “nacional” tiene “na” repetido dos veces.)
    3. Contengan una letra repetida en al menos tres lugares (p. ej. “característica” tiene tres “a”.)

14.3 Herramientas

Ahora que has aprendido los elementos básicos de las expresiones regulares, es tiempo de aprender cómo aplicarlos en problemas reales. En esta sección aprenderás una amplia variedad de funciones de **stringr** que te permitirán:

- Determinar qué cadenas coinciden con un patrón.
- Encontrar la posición de una coincidencia.
- Extraer el contenido de las coincidencias.
- Remplazar coincidencias con nuevos valores.
- Dividir una cadena de acuerdo a una coincidencia.

Una advertencia antes de continuar: debido a que las expresiones regulares son tan poderosas, es fácil intentar resolver todos los problemas con una sola expresión regular. En palabras de Jamie Zawinski:

> Cuando se enfrentan a un problema, algunas personas piensan “Lo sé, usaré expresiones regulares.” Ahora tienen dos problemas.

Como advertencia, revisa esta expresión regular que chequea si una dirección de correo electrónico es válida:

```
(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:
\r\n)?[ \t])*)(?:(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[
?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\
\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:
(?:\r\n)?[ \t])*"(?:(?:\r\n)?[ \t])*)(?:(?:\r\n)?[ \t])*))*)?@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\
\]|\\.|(?:(?:\r\n)?[ \t])*"(?:(?:\r\n)?[ \t])*)(?:(?:\r\n)?[ \t])*))*)
```

Copy

```
@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\r\n)?[
 \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,
;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t]
)*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\
".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*)*:(?:(?:\r\n)?[ \t])*)?
(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".
\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:
\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\[
"()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])
*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])
+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\
.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z
|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(
?:\r\n)?[ \t])*))*)?;\s*)
```

En cierto sentido, este es un ejemplo “patológico” (porque las direcciones de correo electrónico son de verdad sorpresivamente complejas), pero se usa en código real. Mira esta discusión (en inglés) en stackoverflow http://stackoverflow.com/a/201378 para más detalles.

No olvides que estás trabajando en un lenguaje de programación y que tienes otras herramientas a tu disposición. En vez de crear una sola expresión regular compleja, usualmente es más fácil crear una serie de expresiones regulares más simples. Si te atascaste tratando de crear una sola expresión regular que resuelva tu problema, da un paso atrás y piensa cómo podrías dividir el problema en partes más pequeñas. Esto te permitirá ir resolviendo cada desafío antes de moverte al siguiente.

14.3.1 Detectar coincidencias

Para determinar si un vector de caracteres coincide con un patrón de búsqueda, puedes utilizar `str_detect()`. Este devuelve un vector lógico del mismo largo que el *input*:

```r
x <- c("manzana", "plátano", "pera")
str_detect(x, "e")
#> [1] FALSE FALSE  TRUE
```

Recuerda que cuando usas vectores lógicos en un contexto numérico, `FALSE` (*falso*) se convierte en 0 y `TRUE` (*verdadero*) se convierte en 1. Eso hace que `sum()` (*suma*) y `mean()` (*media*) sean funciones útiles si quieres responder preguntas sobre coincidencias a lo largo de un vector más extenso:

```r
# ¿Cuántas palabras comunes empiezan con m?
sum(str_detect(palabras, "^m"))
#> [1] 81
# ¿Qué proporción de palabras comunes terminan con una vocal?
mean(str_detect(palabras, "[aáeéiéoéuú]$"))
#> [1] 0.546
```

Cuando tienes condiciones lógicas complejas, (p. ej., encontrar *a* o *b* pero no *c*, salvo que *d*) suele ser más fácil combinar múltiples llamadas a `str_detect()` con operadores lógicos, que tratar de crear una sola expresión regular. Por ejemplo, hay dos maneras de buscar todas las palabras que no contengan ninguna vocal:

```r
# Encuentra todas las palabras que contengan al menos una vocal, y luego niégalo
sin_vocales_1 <- !str_detect(palabras, "[aáeéiíoóuúüü]")
# Encuentra todas las palabras consistentes solo en consonantes (no vocales)
sin_vocales_2 <- str_detect(palabras, "^[^aáeéiíoóuúüü]+$")
identical(sin_vocales_1, sin_vocales_2)
#> [1] TRUE
```

Los resultados son idénticos; sin embargo, creemos que la primera aproximación es significativamente más fácil de entender. Si tu expresión regular se vuelve extremadamente compleja, trata de dividirla en partes más pequeñas, dale un nombre a cada parte y luego combínalas en operaciones lógicas.

Un uso común de `str_detect()` es para seleccionar elementos que coincidan con un patrón. Puedes hacer eso con subdivisiones lógicas o utilizando `str_subset()`, que es un “envoltorio” (*wrapper*) de esas operaciones.

```
palabras[str_detect(palabras, "x$")]
#> [1] "ex"
str_subset(palabras, "x$")
#> [1] "ex"
```

En todo caso, lo más habitual es que tus cadenas de caracteres sean una columna de un *data frame* y que prefieras utilizar la función `filter()` (*filtrar*):

```
df <- tibble(
  palabra = palabras,
  i = seq_along(palabra)
)
df %>%
  filter(str_detect(palabras, "x$"))
#> # A tibble: 1 x 2
#>   palabra     i
#>   <chr>   <int>
#> 1 ex        338
```

Una varación de `str_detect()` es `str_count()` (*count* = contar): más que un simple sí o no, te indica cuántas coincidencias hay en una cadena:

```
x <- c("manzana", "plátano", "pera")
str_count(x, "a")
#> [1] 3 1 1

# En promedio, ¿cuántas vocales hay por palabra?
mean(str_count(palabras, "[aáeéiíoóuúü]"))
#> [1] 2.72
```

Es natural usar `str_count()` junto con `mutate()`:

```
df %>%
  mutate(
    vocales = str_count(palabra, "[aáeéiíoóuúü]"),
    consonantes = str_count(palabra, "[^aáeéiíoóuúü]")
  )
#> # A tibble: 1,000 x 4
#>   palabra      i vocales consonantes
#>   <chr>    <int>   <int>       <int>
#> 1 a            1       1           0
#> 2 abril        2       2           3
#> 3 acción       3       3           3
#> 4 acciones     4       4           4
#> 5 acerca       5       3           3
#> 6 actitud      6       3           4
#> # … with 994 more rows
```

Ten en cuenta que las coincidencias nunca se superponen. Por ejemplo, en `"abababa"`, ¿cuántas veces se encontrará una coincidencia con el patrón `"aba"`? Las expresiones regulares dicen que dos, no tres:

```
str_count("abababa", "aba")
#> [1] 2
str_view_all("abababa", "aba")
```

abababa

Toma nota sobre el uso de `str_view_all()`. Como aprenderás dentro de poco, muchas funciones de **stringr** vienen en pares: una función trabaja con una sola coincidencia y la otra con todas. La segunda función tendrá el sufijo `_all` (*todas*).

14.3.2 Ejercicios

1. Para cada uno de los siguientes desafíos, intenta buscar una solución utilizando tanto una expresión regular simple como una combinación de múltiples llamadas a `str_detect()`.

1. Encuentra todas las palabras que empiezan o terminan con `y`.
2. Encuentra todas las palabras que empiezan con una vocal y terminan con una consonante.
3. ¿Existen palabras que tengan todas las vocales?

2. ¿Qué palabra tiene el mayor número de vocales? ¿Qué palabra tiene la mayor proporción de vocales? (Pista: ¿cuál es el denominador?)

14.3.3 **Extraer coincidencias**

Para extraer el texto de una coincidencia utiliza `str_extract()`. Para mostrar cómo funciona, necesitaremos un ejemplo más complicado. Para ello, usaremos una selección y adaptación al español de las oraciones disponibles originalmente en `stringr::sentences` y que puedes encontrar en `datos::oraciones`:

```
length(oraciones)
#> [1] 50
head(oraciones)
#> [1] "Las casas están construidas de ladrillos de arcilla roja."
#> [2] "La caja fue arrojada al lado del camión estacionado."
#> [3] "El domingo es la mejor parte de la semana."
#> [4] "Agrega a la cuenta de la tienda hasta el último centavo."
#> [5] "Nueve hombres fueron contratados para excavar las ruinas."
#> [6] "Pega la hoja en el fondo azul oscuro."
```

Copy

Imagina que quieres encontrar todas las oraciones que tengan el nombre de un color. Primero, creamos un vector con nombres de colores y luego lo convertimos en una sola expresión regular:

```
colores <- c("rojo", "amarillo", "verde", "azul", "marrón")
coincidencia_color <- str_c(colores, collapse = "|")
coincidencia_color
#> [1] "rojo|amarillo|verde|azul|marrón"
```

Copy

Ahora, podemos seleccionar las oraciones que contienen un color y extraer luego el color para saber de cuál se trata:

```
tiene_color <- str_subset(oraciones, coincidencia_color)
coincidencia <- str_extract(tiene_color, coincidencia_color)
head(coincidencia)
#> [1] "azul"   "azul"   "rojo"   "azul"   "azul"   "marrón"
```

Copy

Ten en cuenta que `str_extract()` solo extrae la primera coincidencia. Podemos ver eso de manera sencilla seleccionando primero todas las oraciones que tengan más de una coincidencia:

```
mas <- oraciones[str_count(oraciones, coincidencia_color) > 1]
str_view_all(mas, coincidencia_color)
```

Copy

`Instalaron azulejos verdes en la cocina.`

`Si arrojo la taza azul al suelo se romperá.`

`Las hojas se vuelven de color marrón y amarillo en el otoño.`

`La luz verde en la caja marrón parpadeaba.`

str_extract(mas, coincidencia_color) #> [1] "azul" "rojo" "marrón" "verde"

Este es un patrón de coincidencia común para las funciones de **stringr**, ya que trabajar con una sola coincidencia te permite utilizar estructuras de datos más simples. Para obtener todas las coincidencias, utiliza `str_extract_all()`. Esta función devuelve una lista:

```
str_extract_all(mas, coincidencia_color)
#> [[1]]
#> [1] "azul"  "verde"
#>
#> [[2]]
#> [1] "rojo" "azul"
#>
#> [[3]]
#> [1] "marrón"   "amarillo"
#>
#> [[4]]
#> [1] "verde"  "marrón"
```

Aprenderás más sobre listas en el capítulo sobre vectores y en el sobre iteración.

Si utilizas `simplify = TRUE` (es decir, *simplificar* = VERDADERO), `str_extract_all()` devolverá una matriz con las coincidencias más cortas expandidas hasta el largo de las más extensas:

```
str_extract_all(mas, coincidencia_color, simplify = TRUE)
#>      [,1]     [,2]
#> [1,] "azul"   "verde"
#> [2,] "rojo"   "azul"
#> [3,] "marrón" "amarillo"
#> [4,] "verde"  "marrón"

x <- c("a", "a b", "a b c")
str_extract_all(x, "[a-z]", simplify = TRUE)
#>      [,1] [,2] [,3]
#> [1,] "a"  ""   ""
#> [2,] "a"  "b"  ""
#> [3,] "a"  "b"  "c"
```

14.3.3.1 Ejercicios

1. Te habrás dado cuenta que en el ejemplo anterior la expresión regular que utilizamos también devolvió como resultado “arrojo” y “azulejos”, que no son nombres de colores. Modifica la expresión regular para resolver ese problema.

2. De `datos::oraciones` extrae:

    1. La primera palabra de cada oración.
    2. Todas las palabras que terminen en `ción`.
    3. Todos los plurales.

14.3.4 Coincidencias agrupadas

Antes en este capítulo hablamos sobre el uso de paréntesis para aclarar la *precedencia* y las *referencias previas* al buscar coincidencias. También puedes utilizar los paréntesis para extraer una coincidencia compleja. Por ejemplo, imagina que quieres extraer los sustantivos de una oración. Como heurística, buscaremos cualquier palabra que venga después de un artículo (*el*, *la*, *un*, *una*, etc.). Definir qué es una palabra en una expresión regular es un poco complicado, así que aquí utilizaremos una aproximación simple: una secuencia de al menos un caracter que no sea un espacio.

```
sustantivo <- "(el|la|los|las|lo|un|una|unos|unas) ([^ ]+)"

tiene_sustantivo <- oraciones %>%
  str_subset(sustantivo) %>%
  head(10)
tiene_sustantivo %>%
  str_extract(sustantivo)
#>  [1] "los de"      "el camión"   "la mejor"    "la cuenta"   "las ruinas."
#>  [6] "la hoja"     "la cocina."  "la taza"     "el tanque."  "el calor"
```

`str_extract()` nos devuelve la coincidencia completa; `str_match()` nos entrega cada componente. En vez de un vector de caracteres, devuelve una matriz con una columna para la coincidencia completa y una columna para cada grupo:

```
tiene_sustantivo %>%
  str_match(sustantivo)
#>       [,1]          [,2]  [,3]
#>  [1,] "los de"      "los" "de"
#>  [2,] "el camión"   "el"  "camión"
#>  [3,] "la mejor"    "la"  "mejor"
#>  [4,] "la cuenta"   "la"  "cuenta"
#>  [5,] "las ruinas." "las" "ruinas."
#>  [6,] "la hoja"     "la"  "hoja"
#>  [7,] "la cocina."  "la"  "cocina."
#>  [8,] "la taza"     "la"  "taza"
#>  [9,] "el tanque."  "el"  "tanque."
#> [10,] "el calor"    "el"  "calor"
```

(Como era de esperarse, nuestra heurística para detectar sustantivos es pobre, ya que también selecciona adjetivos como “mejor” y preposiciones como “de”).

Si tus datos están en un tibble, suele ser más fácil utilizar `tidyr::extract()`. Funciona como `str_match()` pero requiere ponerle un nombre a las coincidencias, las que luego son puestas en columnas nuevas:

```
tibble(oracion = oraciones) %>%
  tidyr::extract(
    oracion, c("articulo", "sustantivo"), "(el|la|los|las|un|una|unos|unas) ([^ ]+)",
    remove = FALSE
  )
#> # A tibble: 50 x 3
#>   oracion                                            articulo sustantivo
#>   <chr>                                              <chr>    <chr>
#> 1 Las casas están construidas de ladrillos de arcilla roja. los      de
#> 2 La caja fue arrojada al lado del camión estacionado.      el       camión
#> 3 El domingo es la mejor parte de la semana.                la       mejor
#> 4 Agrega a la cuenta de la tienda hasta el último centavo.  la       cuenta
#> 5 Nueve hombres fueron contratados para excavar las ruinas. las      ruinas.
#> 6 Pega la hoja en el fondo azul oscuro.                     la       hoja
#> # … with 44 more rows
```

Al igual que con `str_extract()`, si quieres todas las coincidencias para cada cadena, tienes que utilizar `str_match_all()`.

14.3.4.1 Ejercicios

1. Busca en `datos::oraciones` todas las palabras que vengan después de un “número”, como “un(o|a)”, “dos”, “tres”, etc. Extrae tanto el número como la palabra.

2. En español a veces se utiliza el guión para unir adjetivos, establecer relaciones entre conceptos o para unir gentilicios (p. ej., *teórico-práctico*, *precio-calidad*, *franco-porteña*). ¿Cómo podrías encontrar esas palabras y separar lo que viene antes y después del guión?

14.3.5 Remplazar coincidencias

`str_replace()` y `str_replace_all()` te permiten remplazar coincidencias en una nueva cadena. Su uso más simple es para remplazar un patrón con una cadena fija:

```
x <- c("manzana", "pera", "banana")
str_replace(x, "[aeiou]", "-")
#> [1] "m-nzana" "p-ra"    "b-nana"
str_replace_all(x, "[aeiou]", "-")
#> [1] "m-nz-n-" "p-r-"    "b-n-n-"
```

Con `str_replace_all()` puedes realizar múltiples remplazos a través de un vector cuyos elementos tiene nombre (*named vector*):

```
x <- c("1 casa", "2 autos", "3 personas")
str_replace_all(x, c("1" = "una", "2" = "dos", "3" = "tres"))
#> [1] "una casa"      "dos autos"     "tres personas"
```

Copy

En vez de hacer remplazos con una cadena fija, puedes utilizar *referencias previas* para insertar componentes de la coincidencia. En el siguiente código invertimos el orden de la segunda y la tercera palabra:

```
oraciones %>%
  str_replace("([^ ]+) ([^ ]+) ([^ ]+)", "\\1 \\3 \\2") %>%
  head(5)
#> [1] "Las están casas construidas de ladrillos de arcilla roja."
#> [2] "La fue caja arrojada al lado del camión estacionado."
#> [3] "El es domingo la mejor parte de la semana."
#> [4] "Agrega la a cuenta de la tienda hasta el último centavo."
#> [5] "Nueve fueron hombres contratados para excavar las ruinas."
```

Copy

14.3.5.1 **Ejercicios**

1. Remplaza en una cadena todas las barras por barras invertidas.
2. Implementa una versón simple de `str_to_lower()` (*a minúsculas*) usando `replace_all()`.
3. Cambia la primera y la última letra en `palabras`. ¿Cuáles de esas cadenas siguen siendo palabras?

14.3.6 **Divisiones**

Usa `str_split()` para dividir una cadena en partes. Por ejemplo, podemos dividir `oraciones` en palabras:

```
oraciones %>%
  head(5) %>%
  str_split(" ")
#> [[1]]
#> [1] "Las"         "casas"       "están"       "construidas" "de"
#> [6] "ladrillos"   "de"          "arcilla"     "roja."
#>
#> [[2]]
#> [1] "La"           "caja"         "fue"          "arrojada"     "al"
#> [6] "lado"         "del"          "camión"       "estacionado."
#>
#> [[3]]
#> [1] "El"      "domingo" "es"      "la"      "mejor"   "parte"   "de"
#> [8] "la"      "semana."
#>
#> [[4]]
#>  [1] "Agrega"   "a"        "la"       "cuenta"   "de"       "la"
#>  [7] "tienda"   "hasta"    "el"       "último"   "centavo."
#>
#> [[5]]
#> [1] "Nueve"       "hombres"     "fueron"      "contratados" "para"
#> [6] "excavar"     "las"         "ruinas."
```

Copy

Como cada componente podría tener un número diferente de elementos, esto devuelve una lista. Si estás trabajando con vectores de extensión 1, lo más fácil es extraer el primer elemento de la lista:

```
"a|b|c|d" %>%
  str_split("\\|") %>%
  .[[1]]
#> [1] "a" "b" "c" "d"
```

Copy

Otra opción es, al igual que con otras funciones de **stringr** que devuelven una lista, utilizar `simplify = TRUE` para obtener una matriz:

```
oraciones %>%
  head(5) %>%
  str_split(" ", simplify = TRUE)
#>      [,1]     [,2]      [,3]     [,4]          [,5]    [,6]        [,7]
#> [1,] "Las"    "casas"   "están"  "construidas" "de"    "ladrillos" "de"
#> [2,] "La"     "caja"    "fue"    "arrojada"    "al"    "lado"      "del"
#> [3,] "El"     "domingo" "es"     "la"          "mejor" "parte"     "de"
#> [4,] "Agrega" "a"       "la"     "cuenta"      "de"    "la"        "tienda"
#> [5,] "Nueve"  "hombres" "fueron" "contratados" "para"  "excavar"   "las"
#>      [,8]      [,9]           [,10]    [,11]
#> [1,] "arcilla" "roja."        ""       ""
#> [2,] "camión"  "estacionado." ""       ""
#> [3,] "la"      "semana."      ""       ""
#> [4,] "hasta"   "el"           "último" "centavo."
#> [5,] "ruinas." ""             ""       ""
```

Copy

También puedes indicar un número máximo de elementos:

```
campos <- c("Nombre: Hadley", "País: NZ", "Edad: 35")
campos %>% str_split(": ", n = 2, simplify = TRUE)
#>      [,1]     [,2]
#> [1,] "Nombre" "Hadley"
#> [2,] "País"   "NZ"
#> [3,] "Edad"   "35"
```

Copy

En vez de dividir una cadena según patrones, puedes hacerlo según caracter, línea, oración o palabra. Para ello, puedes utilizar la función `boundary()` (*límite*). En el siguiente ejemplo la división se hace por palabra (*word*):

```
x <- "Esta es una oración. Esta es otra oración."
str_view_all(x, boundary("word"))
```

Copy

`Esta` `es` `una` `oración`. `Esta` `es` `otra` `oración`.

str_split(x, " ")[[1]] #> [1] "Esta" "es" "una" "oración." "Esta" "es" "otra" #> [8] "oración." str_split(x, boundary("word"))[[1]] #> [1] "Esta" "es" "una" "oración" "Esta" "es" "otra" #> [8] "oración"

14.3.6.1 Ejercicios

1. Divide una cadena como `"manzanas, peras y bananas"` en elementos individuales.
2. ¿Por qué es mejor dividir utilizando `boundary("word")` en vez de `" "`?
3. ¿Qué pasa si dividimos con una cadena vacía (`""`)? Experimenta y luego lee la documentación

14.3.7 Buscar coincidencias

`str_locate()` y `str_locate_all()` te indican la posición inicial y final de una coincidencia. Son particularmente útiles cuando ninguna otra función hace exactamente lo que quieres. Puedes utilizar `str_locate()` para encontrar los patrones de coincidencia y `str_sub()` para extraerlos y/o modificarlos.

14.4 Otro tipo de patrones

Cuando utilizas un patrón que es una cadena, este automáticamente es encapsulado en la función `regex()` (*regex* es la forma abreviada de *regular expression*, es decir, expresión regular):

```
# La manera regular en que escribimos el patrón
str_view(frutas, "nana")
# Es un atajo de
str_view(frutas, regex("nana"))
```

Copy

Puedes utilizar los otros argumentos de `regex()` para controlar los detalles de la coincidencia:

- `ignore_case = TRUE` permite que la búsqueda coincida tanto con caracteres en mayúscula como en minúscula. Este argumento siempre utiliza los parámetros de tu *locale*.

```
bananas <- c("banana", "Banana", "BANANA")
str_view(bananas, "banana")
```

Copy

banana

Banana

BANANA

str_view(bananas, regex("banana", ignore_case = TRUE))

banana

Banana

BANANA

- `multiline = TRUE` permite que `^` y `$` coincidan con el inicio y fin de cada línea, en vez del inicio y fin de la cadena completa.

```
x <- "Línea 1\nLínea 2\nLínea 3"
str_extract_all(x, "^Línea")[[1]]
#> [1] "Línea"
str_extract_all(x, regex("^Línea", multiline = TRUE))[[1]]
#> [1] "Línea" "Línea" "Línea"
```

Copy

- `comments = TRUE` te permite utilizar comentarios y espacios en blanco para hacer más entendibles las expresiones regulares complejas. Los espacios son ignorados, al igual que todo lo que está después de `#`. Para coincidir un espacio de manera literal, tendrías que "escaparlo: `"\\ "`.

```
telefono <- regex("
  \\(?     # paréntesis inicial opcional
  (\\d{3}) # código de área
  [) -]?   # paréntesis, espacio o guión inicial opcional
  (\\d{3}) # otros tres números
  [ -]?    # espacio o guión opcional
  (\\d{3}) # otros tres números
  ", comments = TRUE)

str_match("514-791-8141", telefono)
#>      [,1]          [,2]  [,3]  [,4]
#> [1,] "514-791-814" "514" "791" "814"
```

Copy

- `dotall = TRUE` permite que `.` coincida con todo, incluidos los saltos de línea (`\n`).

Existen otras tres funciones que puedes utilizar en vez de `regex()`:

- `fixed()`: busca una coincidencia exacta de la secuencia de bytes especificada. Ignora todas las expresiones regulares especiales y opera a un nivel muy bajo. Esto te permite evitar formas de "escapado" complejas y puede ser mucho más rápida que las expresiones regulares. La comparación utilizando `microbenchmark` muestra que `fixed()` es casi dos veces más rápida.

```
#install.packages(microbenchmark)
microbenchmark::microbenchmark(
  fixed = str_detect(oraciones, fixed("la")),
  regex = str_detect(oraciones, "la"),
  times = 20
)
#> Unit: microseconds
#>   expr    min      lq     mean  median     uq     max neval
#>  fixed 35.818 37.1995 55.03315 42.0075 44.063 317.361    20
#>  regex 44.412 45.2985 48.06575 46.2065 46.731  83.034    20
```

Copy

IMPORTANTE: ten precaución al utilizar `fixed()` con datos que no estén en inglés. Puede causar problemas porque muchas veces existen múltiples formas de representar un mismo caracter. Por ejemplo, hay dos formas de difinir "á": como un solo caracter o como una "a" con un acento:

```
a1 <- "\u00e1"
a2 <- "a\u0301"
c(a1, a2)
#> [1] "á" "á"
a1 == a2
#> [1] FALSE
```

Ambas se *renderean* de manera idéntica, pero como están definidas de manera distinta, `fixed()` no encuentra una coincidencia. En su lugar, puedes utilizar `coll()`, que definiremos a continuación, ya que respeta las reglas humanas de comparación de caracteres:

```
str_detect(a1, fixed(a2))
#> [1] FALSE
str_detect(a1, coll(a2))
#> [1] TRUE
```

- `coll()`: compara cadenas usando reglas de secuenciación (*collation*) estándar. Esto es útil para buscar coincidencias que sean insensibles a mayúsculas y minúsculas. Ten en cuenta que `coll()` incluye un parámetro para el *locale*, Lamentablemente, se utilizan diferentes reglas en diferentes partes del mundo.

```
# Esto quiere decir que también tienes que prestar atención a esas
# diferencias al buscar coincidencias insensibles a mayúsculas y
# minúsculas
i <- c("I", "İ", "i", "ı")
i
#> [1] "I" "İ" "i" "ı"

str_subset(i, coll("i", ignore_case = TRUE))
#> [1] "I" "i"
str_subset(i, coll("i", ignore_case = TRUE, locale = "tr"))
#> [1] "İ" "i"
```

Tanto `fixed()` como `regex()` tienen argumentos para ignorar la diferencia entre mayúsculas y minúsculas (`ignore_case`); sin embargo, no te permiten elegir tu *locale*: siempre utilizan el que está definido por defecto. Puedes ver cuál se está usando con el siguiente código. Pronto hablaremos más sobre el paquete **stringi**.

```
stringi::stri_locale_info()
#> $Language
#> [1] "en"
#>
#> $Country
#> [1] "US"
#>
#> $Variant
#> [1] ""
#>
#> $Name
#> [1] "en_US"
```

Una desventaja de `coll()` es la velocidad. Debido a que las reglas para reconocer qué caracteres son iguales suelen ser complicadas, `coll()` es relativamente más lenta al compararla con `regex()` y `fixed()`.

- Como viste con `str_split()`, puedes utilizar `boundary()` para coincidir límites. También puedes utilizarla con otras funciones:

```
x <- "Esta es una oración."
str_view_all(x, boundary("word"))
```

`Esta` `es` `una` `oración`.

str_extract_all(x, boundary("word")) #> [[1]] #> [1] "Esta" "es" "una" "oración"

14.4.1 Ejercicios

1. ¿Cómo buscarías todas las cadenas que contienen `\` con `regex()` vs. con `fixed()` ?

2. ¿Cuáles son las cinco palabras más comunes en `oraciones` ?

14.5 Otros usos de las expresiones regulares.

Existen dos funciones útiles en R base que también utilizan expresiones regulares:

- `apropos()` busca todos los objetos disponibles en el ambiente global (*global environment*). Esto es útil si no recuerdas bien el nombre de una función.

```
apropos("replace")
#> [1] "%+replace%"       "replace"          "replace_na"       "setReplaceMethod"
#> [5] "str_replace"      "str_replace_all"  "str_replace_na"   "theme_replace"
```

- `dir()` entrega una lista con todos los archivos en un directorio. El argumento `pattern` recibe una expresión regular y retorna solo los nombres de archivos que coinciden con ese patrón. Por ejemplo, puedes encontrar todos los archivos de R Markdown en el directorio actual con:

```
head(dir(pattern = "\\.Rmd$"))
#> [1] "01-intro.Rmd"            "02-explore.Rmd"
#> [3] "03-visualize.Rmd"        "04-workflow-basics.Rmd"
#> [5] "05-transform.Rmd"        "06-workflow-scripts.Rmd"
```

(Si te resulta más cómodo trabajar con “globs”, es decir, especificar los nombres de archivo utilizando comodines, como en `*.Rmd` , puedes convertirlos a expresiones regulares con la función `glob2rx()`)

14.6 stringi

stringr está construido sobre la base del paquete **stringi**. **stringr** es útil cuando estás aprendiendo, ya que presenta un set mínimo de funciones, que han sido elegidas cuidadosamente para manejar las funciones de manipulación de cadenas más comunes. **stringi**, por su parte, está diseñado para ser comprehensivo. Contiene casi todas las funciones que podrías necesitar: **stringi** tiene 250 funciones, frente a las 49 de **stringr**.

Si en algún momento te encuentras en dificultades para hacer algo en **stringr** , vale la pena darle una mirada a **stringi**. Ambos paquetes funcionan de manera muy similar, por lo que deberías poder traducir tu conocimiento sobre **stringr** de manera natural. La principal diferencia es el prefijo: `str_` vs. `stri_` .

14.6.1 Ejercicios

1. Busca la función de **stringi** que:

    1. Cuenta el número de palabras.

    2. Busca cadenas duplicadas.

    3. Genera texto aleatorio.

2. ¿Cómo puedes controlar qué lengua usa `stri_sort()` para ordenar?

« 13 Datos relacionales

15 Factores »

15 Factores

15.1 Introducción

En R, los factores se usan para trabajar con variables categóricas, es decir, variables que tienen un conjunto fijo y conocido de valores posibles. También son útiles cuando quieres mostrar vectores de caracteres en un orden no alfabético.

Históricamente, los factores eran más sencillos de trabajar que los caracteres. Como resultado, muchas de las funciones de R base automáticamente convierten los caracteres a factores. Esto significa que, a menudo, los factores aparecen en lugares donde no son realmente útiles. Afortunadamente, no tienes que preocuparte de eso en el tidyverse y puedes concentrarte en situaciones en las que los factores son genuinamente útiles.

15.1.1 Prerrequisitos

Para trabajar con factores, vamos a usar el paquete **forcats**, que es parte del **tidyverse**. Este paquete provee herramientas para lidiar con variables **cat**egóricas (¡y es un anagrama de *factores* en inglés!) y ofrece un amplio rango de ayudas para trabajar con factores.

```
library(tidyverse)
library(datos)
```

Copy

15.1.2 Aprendiendo más

Si quieres aprender más sobre factores, te recomendamos leer el artículo de Amelia McNamara y Nicholas Horton, *Wrangling categorical data in R* (el nombre significa *Domando/Manejando Datos Categóricos en R*). Este artículo cuenta parte de la historia discutida en *stringsAsFactors: An unauthorized biography* (del inglés *cadenasComoFactores: Una Biografía No Autorizada*) y *stringsAsFactors = <sigh>* (del inglés *cadenasComoFactores = <suspiro>*), y compara las propuestas *tidy* para los datos categóricos demostrados en este libro, en comparación a los métodos de R base. Una versión temprana de este artículo ayudó a motivar y definir el alcance del paquete **forcats**. ¡Gracias Amelia y Nick!

15.2 Creando factores

Imagina que tienes una variable que registra meses:

```
x1 <- c("Dic", "Abr", "Ene", "Mar")
```

Copy

Usar una cadena de caracteres (o *string*, en inglés) para guardar esta variable tiene dos problemas:

1. Solo hay doce meses posibles y no hay nada que te resguarde de errores de tipeo:

```
x2 <- c("Dic", "Abr", "Eme", "Mar")
```

Copy

2. No se ordena de una forma útil:

```
sort(x1)
#> [1] "Abr" "Dic" "Ene" "Mar"
```

Copy

Puedes solucionar ambos problemas con un factor. Para crearlo, debes empezar definiendo una lista con los **niveles** válidos:

```
niveles_meses <- c(
  "Ene", "Feb", "Mar", "Abr", "May", "Jun",
  "Jul", "Ago", "Sep", "Oct", "Nov", "Dic"
)
```

Copy

On this page

View source
Edit this page

Ahora puedes crear un factor:

```
y1 <- factor(x1, levels = niveles_meses)
y1
#> [1] Dic Abr Ene Mar
#> Levels: Ene Feb Mar Abr May Jun Jul Ago Sep Oct Nov Dic
sort(y1)
#> [1] Ene Mar Abr Dic
#> Levels: Ene Feb Mar Abr May Jun Jul Ago Sep Oct Nov Dic
```

Cualquier valor no fijado en el conjunto será convertido a `NA` de forma silenciosa:

```
y2 <- factor(x2, levels = niveles_meses)
y2
#> [1] Dic  Abr  <NA> Mar
#> Levels: Ene Feb Mar Abr May Jun Jul Ago Sep Oct Nov Dic
```

Si quieres una advertencia, puedes usar `readr::parse_factor()` (*segmentar un factor*, en inglés):

```
y2 <- parse_factor(x2, levels = niveles_meses)
#> Warning: 1 parsing failure.
#> row col           expected actual
#>   3  -- value in level set    Eme
```

Si omites los niveles, se van a definir a partir de los datos en orden alfabético:

```
factor(x1)
#> [1] Dic Abr Ene Mar
#> Levels: Abr Dic Ene Mar
```

A veces es preferible que el orden de los niveles se corresponda con su primera aparición en los datos. Puedes hacer esto cuando creas el factor, al definir los niveles con `unique(x)` (*único*) o después con `fct_inorder()` (*factores en orden*):

```
f1 <- factor(x1, levels = unique(x1))
f1
#> [1] Dic Abr Ene Mar
#> Levels: Dic Abr Ene Mar

f2 <- x1 %>% factor() %>% fct_inorder()
f2
#> [1] Dic Abr Ene Mar
#> Levels: Dic Abr Ene Mar
```

Si alguna vez necesitas acceso directo al conjunto de niveles válidos, puedes hacerlo con `levels()` (*niveles*):

```
levels(f2)
#> [1] "Dic" "Abr" "Ene" "Mar"
```

15.3 Encuesta Social General

Por el resto del capítulo, nos vamos a concentrar en `datos::encuesta` . Esta es la versión traducida al español de un conjunto de datos de ejemplo de la General Social Survey, una encuesta realizada en Estados Unidos desde hace mucho tiempo, conducida por la organización de investigación independiente llamada NORC, en la Universidad de Chicago. La encuesta tiene miles de preguntas, así que en el conjunto de datos hemos seleccionado aquellas que ilustran algunos de los desafíos comunes que encontrarás al trabajar con factores.

```
encuesta
#> # A tibble: 21,483 x 9
#>    anio estado_civil   edad raza  ingreso partido religion denominacion horas_tv
#>   <int> <fct>         <int> <fct> <fct>   <fct>   <fct>    <fct>           <int>
#> 1  2000 Nunca se ha …    26 Blan… 8000 -… Ind, p… Protest… Bautistas d…       12
#> 2  2000 Divorciado       48 Blan… 8000 -… No fue… Protest… Bautista, n…       NA
#> 3  2000 Viudo            67 Blan… No apl… Indepe… Protest… No denomina…        2
#> 4  2000 Nunca se ha …    39 Blan… No apl… Ind, p… Cristia… No aplica           4
#> 5  2000 Divorciado       25 Blan… No apl… No fue… Ninguna  No aplica           1
#> 6  2000 Casado           25 Blan… 20000 … Fuerte… Protest… Bautistas d…       NA
#> # … with 21,477 more rows
```

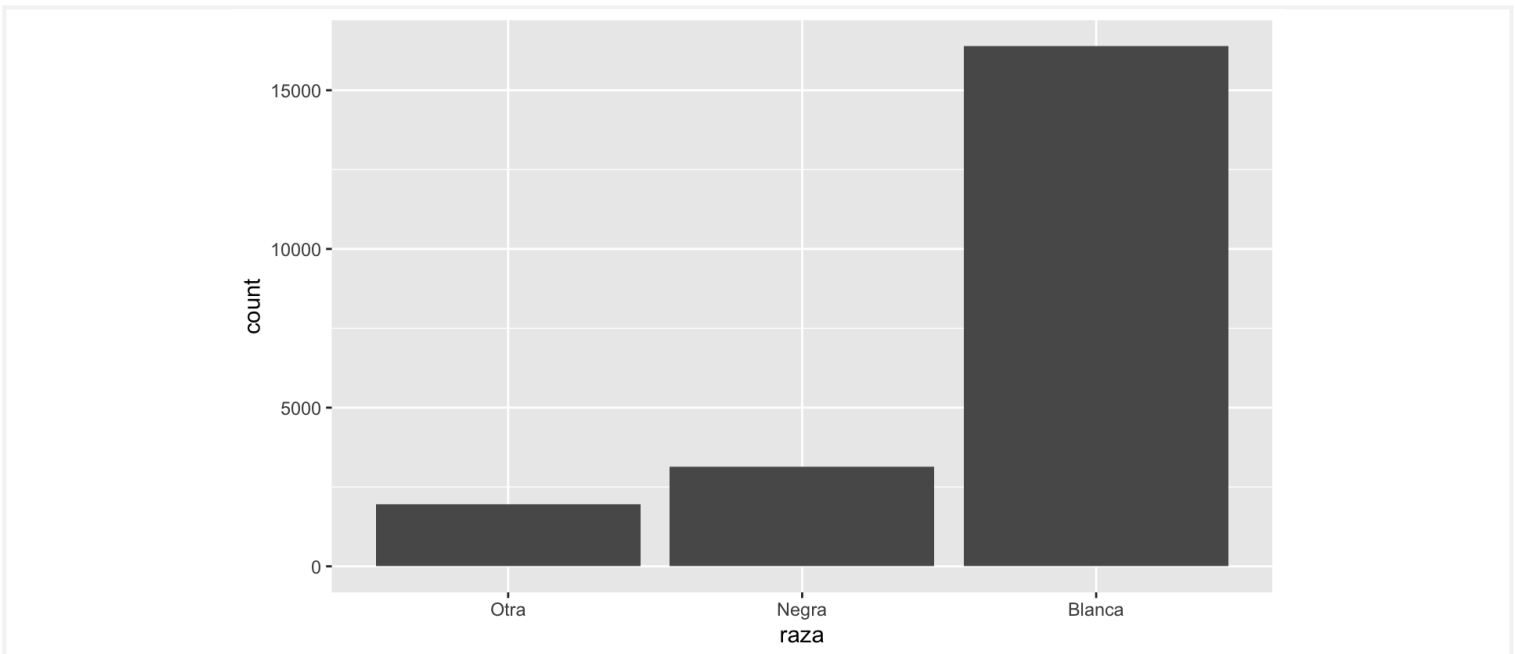
(Recuerda que como este conjunto de datos está provisto por un paquete, puedes obtener más información de las variables con `?encuesta` .)

Cuando los factores están almacenados en un *tibble* no puedes ver sus niveles tan fácilmente. Una forma de verlos es con `count()` (*contar*):

```
encuesta %>%
  count(raza)
#> # A tibble: 3 x 2
#>   raza       n
#> * <fct>  <int>
#> 1 Otra    1959
#> 2 Negra   3129
#> 3 Blanca 16395
```

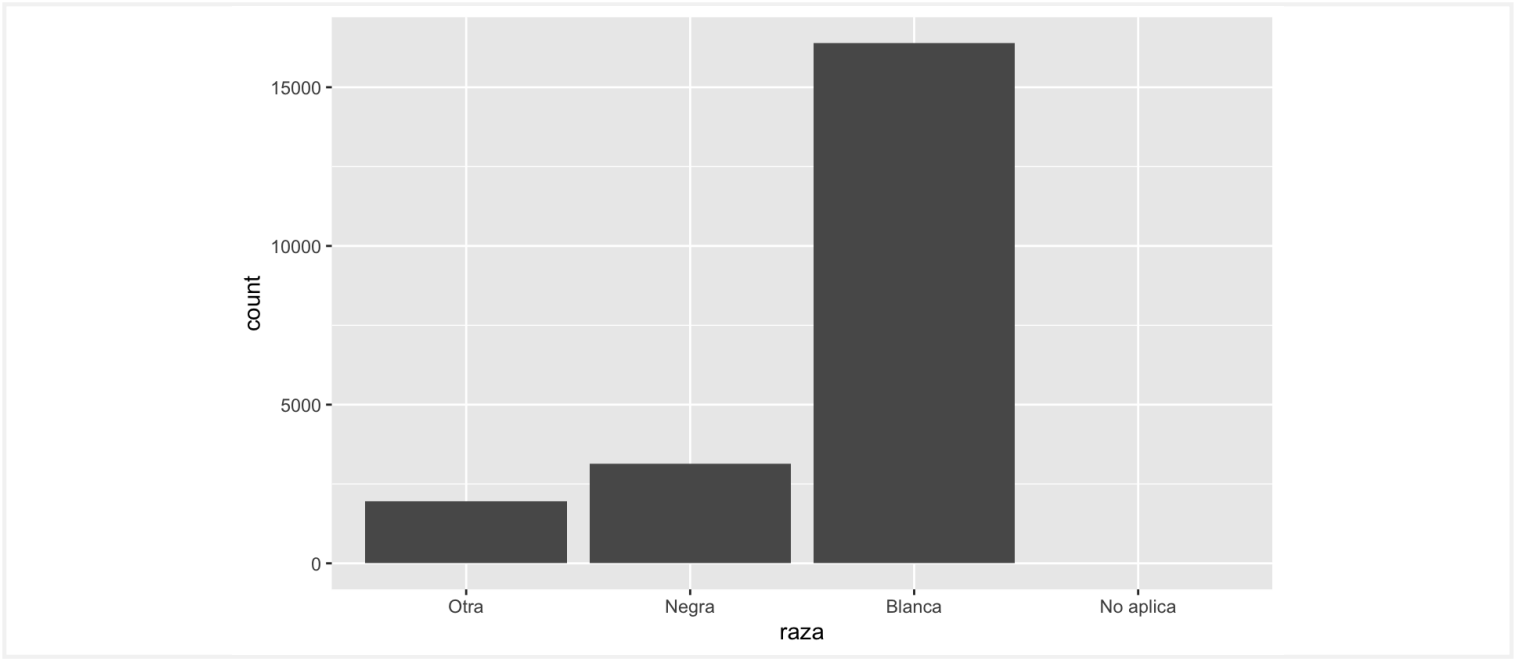
O con un gráfico de barras:

```
ggplot(encuesta, aes(raza)) +
  geom_bar()
```



Por defecto, **ggplot2** retira los niveles que no tienen valores. Puedes forzarlos para que se visualicen con:

```
ggplot(encuesta, aes(raza)) +
  geom_bar() +
  scale_x_discrete(drop = FALSE)
```



Estos niveles representan valores válidos que simplemente no ocurren en este conjunto de datos. Desafortunadamente, **dplyr** no tiene una opción de `drop` (*descartar*, *eliminar*), pero la tendrá en el futuro.

Cuando se trabaja con factores, las dos operaciones más comunes son cambiar el orden de los niveles y cambiar sus valores. Estas operaciones se describen en las siguientes secciones.

15.3.1 Ejercicios

1. Explora la distribución de `ingreso` . ¿Qué hace que el gráfico de barras por defecto sea tan difícil de comprender? ¿Cómo podrías mejorarlo?

2. ¿Cuál es la `religion` más común en esta encuesta? ¿Cuál es el `partido` más común?

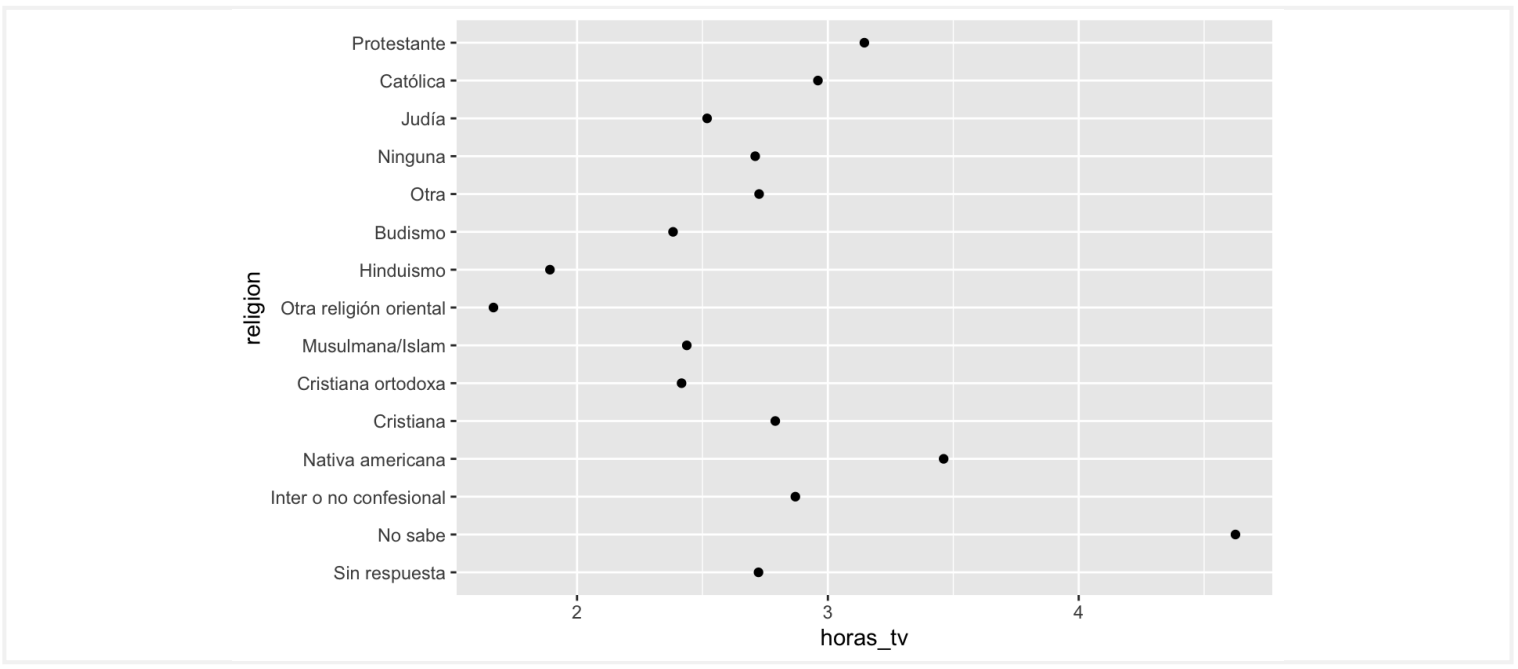
3. ¿A qué `religion` se aplica cada `denominacion` ? ¿Cómo puedes descubrirlo con una tabla? ¿Cómo lo puedes descubrir con una visualización?

15.4 Modificar el orden de los factores

A menudo resulta útil cambiar el orden de los niveles de factores en una visualización. Por ejemplo, imagina que quieres explorar el número promedio de horas consumidas mirando televisión por día, para cada religión:

```
resumen_religion <- encuesta %>%
  group_by(religion) %>%
  summarise(
    edad = mean(edad, na.rm = TRUE),
    horas_tv = mean(horas_tv, na.rm = TRUE),
    n = n()
  )

ggplot(resumen_religion, aes(horas_tv, religion)) + geom_point()
```

Copy

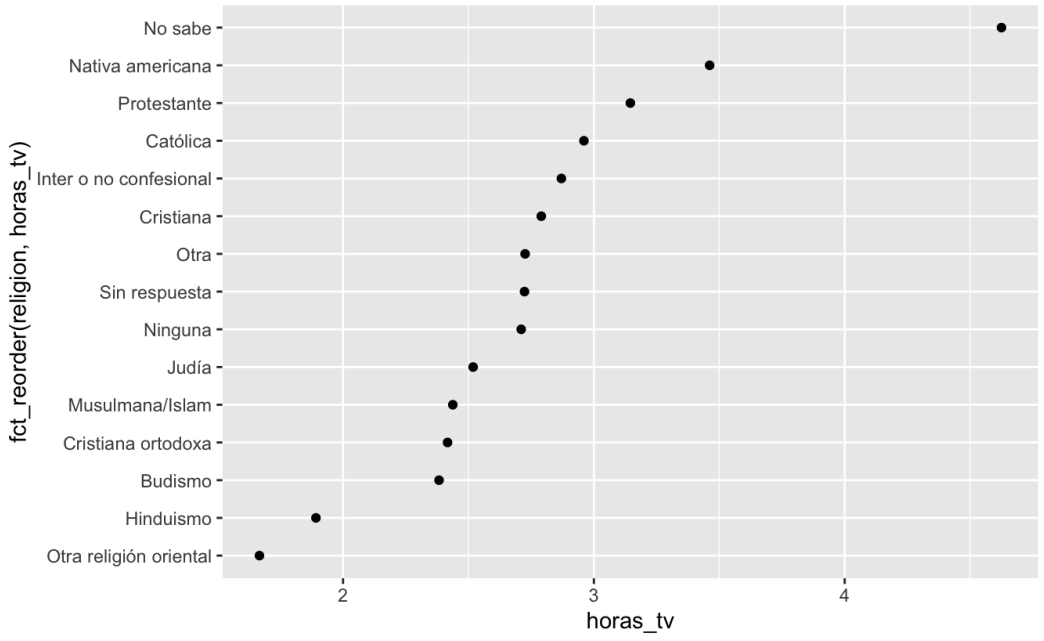


Este gráfico resulta dificil de interpretar porque no hay un patrón general. Podemos mejorarlo al ordenar los niveles de `religion` usando `fct_reorder()` (*reordenar factores*). `fct_reorder()` requiere tres argumentos:

- `f`, el factor cuyos niveles quieres modificar.
- `x`, un vector numérico que quieres usar para reordenar los niveles.
- Opcionalmente, `fun`, una función que se usa si hay múltiples valores de `x` para cada valor de `f`. El valor por defecto es `median` (*mediana*).

```
ggplot(resumen_religion, aes(horas_tv, fct_reorder(religion, horas_tv))) +
  geom_point()
```

Copy



Reordenar la columna religión (`religion`) hace que sea más sencillo ver que las personas en la categoría "No sabe" ven más televisión, mientras que "Hinduismo" y "Otra religión oriental" ven mucho menos.

Cuando haces transformaciones más complicadas, recomendamos que las remuevas de `aes()` hacia un paso de transformación separado usando `mutate()`. Por ejemplo, puedes reescribir el gráfico anterior de la siguiente forma:

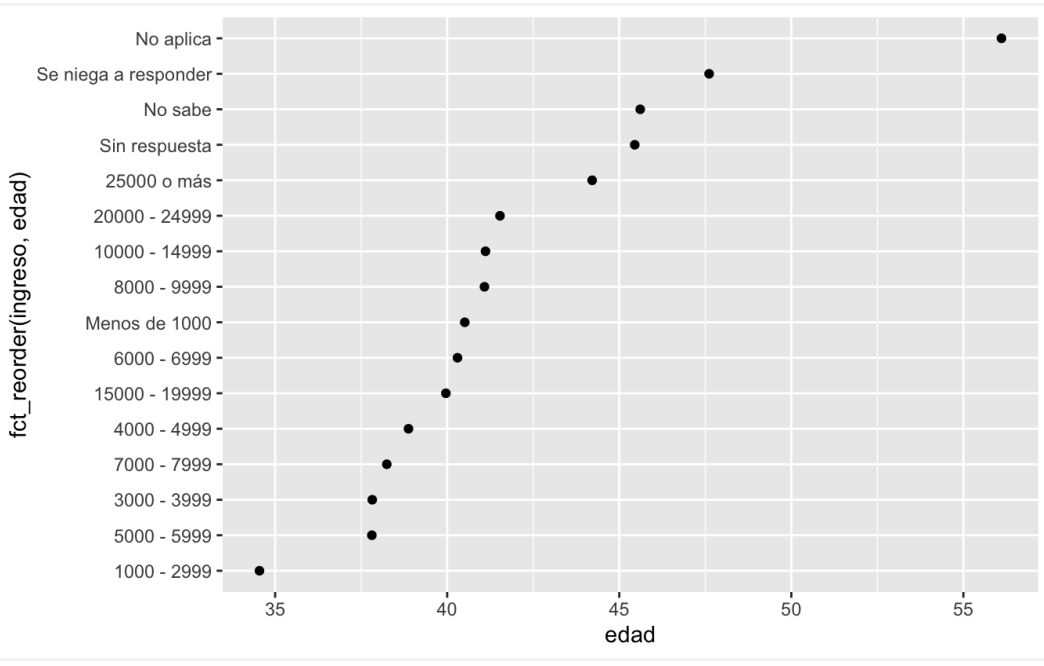
```
resumen_religion %>%
  mutate(religion = fct_reorder(religion, horas_tv)) %>%
  ggplot(aes(horas_tv, religion)) +
  geom_point()
```

Copy

¿Qué sucede si creamos un gráfico para observar cómo varía la edad promedio para cada ingreso reportado?

```
resumen_ingreso <- encuesta %>%
  group_by(ingreso) %>%
  summarise(
    edad = mean(edad, na.rm = TRUE),
    horas_tv = mean(horas_tv, na.rm = TRUE),
    n = n()
  )

ggplot(resumen_ingreso, aes(edad, fct_reorder(ingreso, edad))) + geom_point()
```

Copy

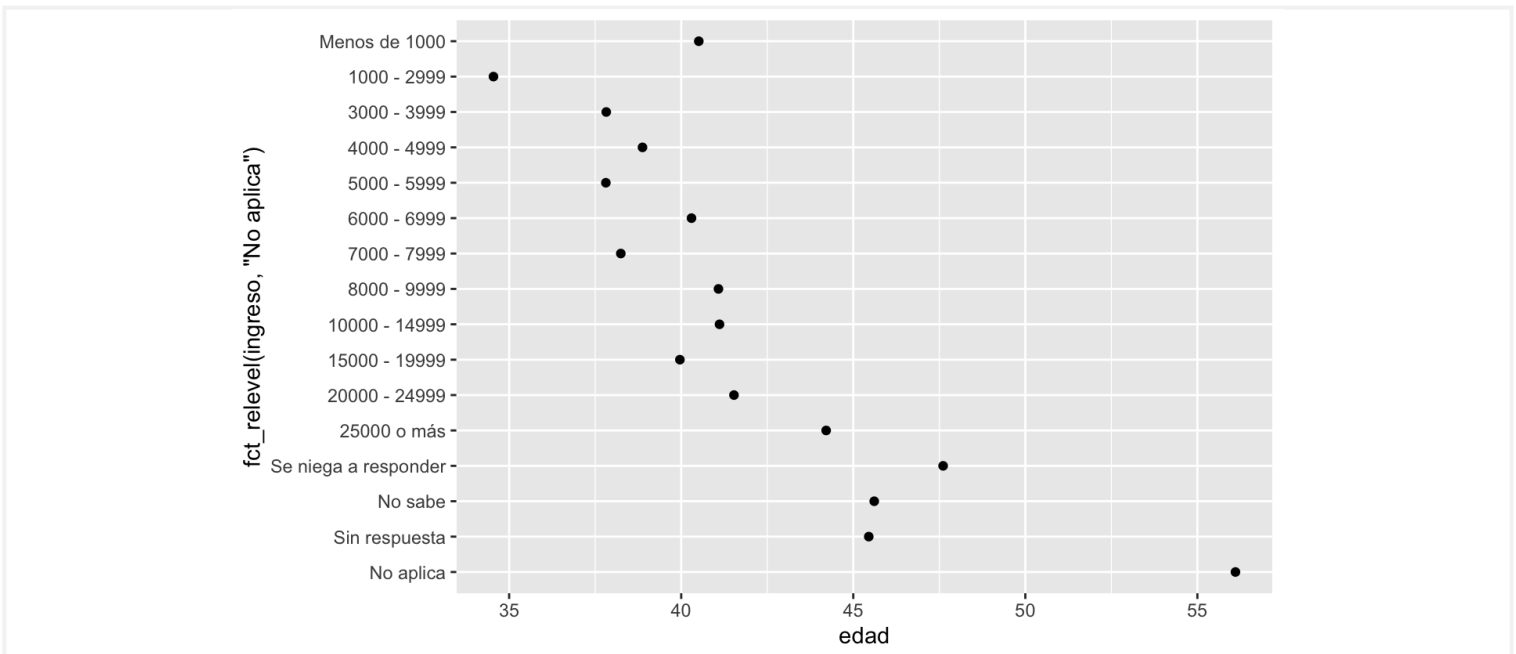


En ete caso, ¡reordenar los niveles arbitrariamente no es una buena idea! Eso es porque `ingreso` ya tiene un orden basado en un principio determinado, con el cual no deberíamos meternos. Reserva `fct_reorder()` para factores cuyos niveles están ordenados arbitrariamente.

Sin embargo, sí tiene sentido mover "No Aplica" al frente, junto a los otros niveles especiales. Puedes usar `fct_relevel()` (*cambiar niveles*). Esta función recibe como argumento un factor, `f` y luego cualquier número de niveles que quieras mover al principio de la línea.

```
ggplot(resumen_ingreso, aes(edad, fct_relevel(ingreso, "No aplica"))) +
  geom_point()
```

Copy

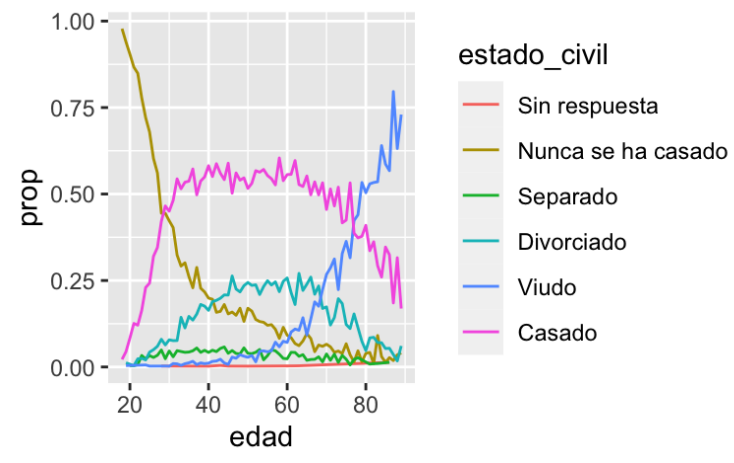


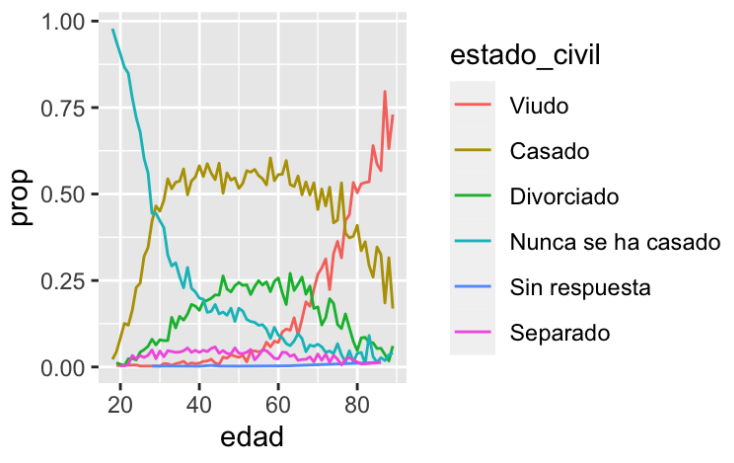
¿Por qué crees que la edad promedio para "No aplica" es tan alta?

Es otro el tipo de reordenamiento que resulta útil cuando estás coloreando las líneas de un gráfico. `fct_reorder2()` reordena el factor mediante los valores `y` asociados con los valores `x` más grandes. Esto hace que el gráfico sea más sencillo de leer, porque los colores de líneas se ajustan con la leyenda.

```
por_edad <- encuesta %>%
  filter(!is.na(edad)) %>%
  count(edad, estado_civil) %>%
  group_by(edad) %>%
  mutate(prop = n / sum(n))

ggplot(por_edad, aes(edad, prop, colour = estado_civil)) +
  geom_line(na.rm = TRUE)

ggplot(por_edad, aes(edad, prop, colour = fct_reorder2(estado_civil, edad, prop))) +
  geom_line() +
  labs(colour = "estado_civil")
```

Copy

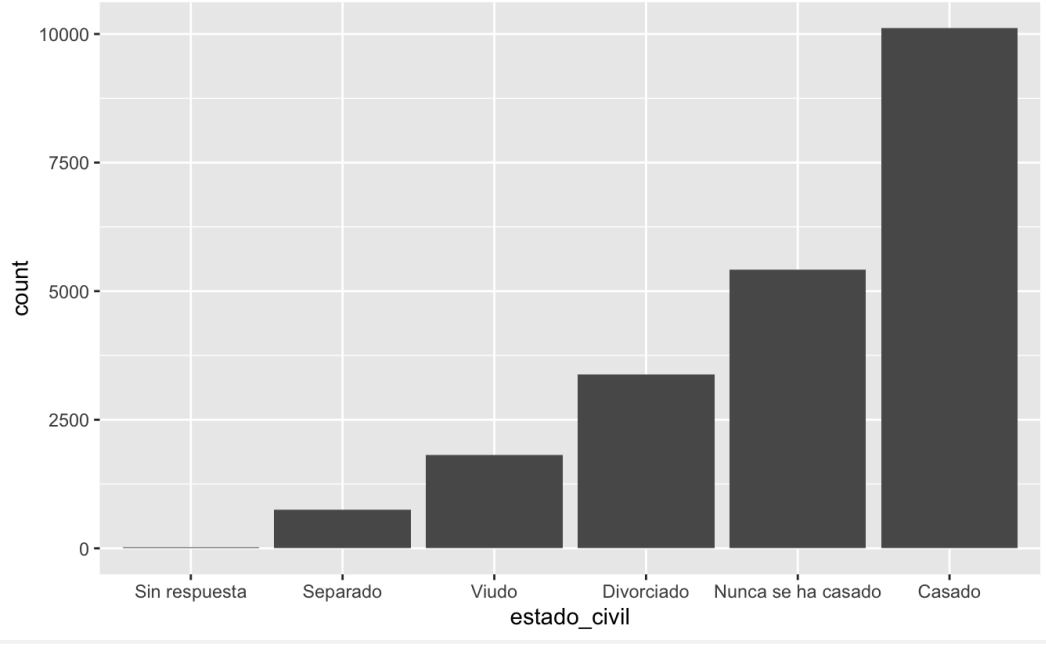




Finalmente, para los gráficos de barra puedes usar `fct_infreq()` (*frecuencia incremental de factores*) para ordenar los niveles incrementalmente según su frecuencia: este es el ordenamiento más sencillo porque no requiere de variables adicionales. Puedes querer combinarlo con `fct_rev()` (*invertir factores*).

```
encuesta %>%
  mutate(estado_civil = estado_civil %>% fct_infreq() %>% fct_rev()) %>%
  ggplot(aes(estado_civil)) +
  geom_bar()
```

Copy



15.4.1 **Ejercicios**

1. Hay algunos números sospechosamente grandes en `horas_tv`. ¿Es la media un buen resumen?

2. Identifica para cada factor en `encuesta` si el orden de los niveles es arbitrario o responde a algún principio.

3. ¿Por qué mover "No aplica" al inicio de los niveles lo llevó al final del gráfico?

15.5 **Modificar los niveles de los factores**

Más poderoso que cambiar el orden de los niveles es cambiar sus valores. Esto te permite clarificar etiquetas para publicación y colapsar niveles para visualizaciones de alto nivel. La herramienta más general y más poderosa es `fct_recode()` (*recodificar factores*). Esta función te permite recodificar o cambiar el valor de cada nivel. Por ejemplo, toma la columna `encuesta$partido`:

```
encuesta %>% count(partido)
#> # A tibble: 10 x 2
#>   partido                       n
#> * <fct>                     <int>
#> 1 Sin respuesta               154
#> 2 No sabe                       1
#> 3 Otro partido                393
#> 4 Fuertemente republicano    2314
#> 5 No fuertemente republicano 3032
#> 6 Ind, pro rep               1791
#> # … with 4 more rows
```

Copy

Los niveles son concisos e inconsistentes. Modifiquémoslos un poco para que sean más largos y para poder usar una construcción paralela.

```
encuesta %>%
  mutate(partido = fct_recode(partido,
    "Republicano duro" = "Fuertemente republicano",
    "Republicano moderado" = "No fuertemente republicano",
    "Independiente pro republicano" = "Ind, pro rep",
    "Independiente pro demócrata" = "Ind, pro dem",
    "Demócrata moderado" = "No fuertemente demócrata",
    "Demócrata duro" = "Fuertemente demócrata"
  )) %>%
  count(partido)
#> # A tibble: 10 x 2
#>   partido                          n
#> * <fct>                        <int>
#> 1 Sin respuesta                  154
#> 2 No sabe                          1
#> 3 Otro partido                   393
#> 4 Republicano duro              2314
#> 5 Republicano moderado          3032
#> 6 Independiente pro republicano 1791
#> # … with 4 more rows
```

`fct_recode()` no modificará los niveles que no han sido mencionados explícitamente y te advertirá si accidentalmente te refieres a un nivel que no existe.

Para combinar grupos, puedes asignar múltiples niveles viejos al mismo nivel nuevo:

```
encuesta %>%
  mutate(partido = fct_recode(partido,
    "Republicano duro" = "Fuertemente republicano",
    "Republicano moderado" = "No fuertemente republicano",
    "Independiente pro republicano" = "Ind, pro rep",
    "Independiente pro demócrata" = "Ind, pro dem",
    "Demócrata moderado" = "No fuertemente demócrata",
    "Demócrata duro" = "Fuertemente demócrata",
    "Otro" = "Sin respuesta",
    "Otro" = "No sabe",
    "Otro" = "Otro partido"
  )) %>%
  count(partido)
#> # A tibble: 8 x 2
#>   partido                          n
#> * <fct>                        <int>
#> 1 Otro                           548
#> 2 Republicano duro              2314
#> 3 Republicano moderado          3032
#> 4 Independiente pro republicano 1791
#> 5 Independiente                 4119
#> 6 Independiente pro demócrata   2499
#> # … with 2 more rows
```

Debes usar esta técnica con cuidado: si agrupas categorías que son realmente diferentes, obtendrás resultados confusos y/o engañosos.

Si quieres colapsar muchos niveles, `fct_collapse()` (*colapsar factores*) es una variante muy útil de `fct_recode()`. Para cada nueva variable puedes proveer un vector de niveles viejos:

```
encuesta %>%
  mutate(partido = fct_collapse(partido,
    otro = c("Sin respuesta", "No sabe", "Otro partido"),
    republicano = c("Fuertemente republicano", "No fuertemente republicano"),
    independiente = c("Ind, pro rep", "Independiente", "Ind, pro dem"),
    demócrata = c("No fuertemente demócrata", "Fuertemente demócrata")
  )) %>%
  count(partido)
#> # A tibble: 4 x 2
#>   partido           n
#> * <fct>         <int>
#> 1 otro            548
#> 2 republicano    5346
#> 3 independiente  8409
#> 4 demócrata      7180
```

A veces, simplemente quieres agrupar todos los grupos pequeños para simplificar un gráfico o tabla. Ese es un trabajo para `fct_lump()` (*agrupar factores*):

```
encuesta %>%
  mutate(religion = fct_lump(religion, other_level = "Otra")) %>%
  count(religion)
#> # A tibble: 2 x 2
#>   religion        n
#> * <fct>       <int>
#> 1 Protestante 10846
#> 2 Otra        10637
```

El comportamiento por defecto es agrupar los grupos pequeños de forma progresiva, asegurando que la agregación continúa siendo el grupo más pequeño. En este caso, esto no resulta demasiado útil: es cierto que la mayoría de los estadounidenses en esta encuesta son protestantes, pero probablemente hemos colapsado en exceso.

En cambio, podemos usar el parámetro `n` para especificar cuántos grupos (excluyendo *otros*) queremos colapsar:

```
encuesta %>%
  mutate(religion = fct_lump(religion, n = 10, other_level = "Otra")) %>%
  count(religion, sort = TRUE) %>%
  print(n = Inf)
#> # A tibble: 10 x 2
#>    religion                  n
#>    <fct>                 <int>
#>  1 Protestante           10846
#>  2 Católica               5124
#>  3 Ninguna                3523
#>  4 Cristiana               689
#>  5 Otra                    458
#>  6 Judía                   388
#>  7 Budismo                 147
#>  8 Inter o no confesional  109
#>  9 Musulmana/Islam         104
#> 10 Cristiana ortodoxa       95
```

15.5.1 Ejercicios

1. ¿Cómo han cambiado en el tiempo las proporciones de personas que se identifican como demócratas, republicanas e independientes?

2. ¿Cómo podrías colapsar `ingreso` en un grupo más pequeño de categorías?

« 14 Cadenas de caracteres

16 Fechas y horas »

"" was written by .

This book was built by the bookdown R package.

16 Fechas y horas

16.1 Introducción

Este capítulo te mostrará cómo trabajar con fechas y horas en R. A primera vista, esto parece sencillo. Las usas en todo momento en tu vida regular y no parecen causar demasiada confusión. Sin embargo, cuanto más aprendes de fechas y horas, más complicadas se vuelven. Para prepararnos, intenta estas preguntas sencillas:

- ¿Todos los años tienen 365 días?
- ¿Todos los días tienen 24 horas?
- ¿Cada minuto tiene 60 segundos?

Estamos seguros que sabes que no todos los años tienen 365 días, ¿pero acaso conoces la regla entera para determinar si un año es bisiesto? (Tiene tres partes, de hecho). Puedes recordar que muchas partes del mundo usan horarios de verano, así que algunos días tienen 23 horas y otros tienen 25. Puede ser que no supieras que algunos minutos tienen 61 segundos, porque de vez en cuando se agregan segundos adicionales ya que la rotación de la tierra se hace cada vez más lenta.

Las fechas y las horas son complicadas porque tienen que reconciliar dos fenómenos físicos (la rotación de la Tierra y su órbita alrededor del sol), con todo un conjunto de fenómenos geopolíticos que incluyen a los meses, los husos horarios y los horarios de verano. Este capítulo no te enseñará cada detalle sobre fechas y horas, pero te dará un sólido fundamento de habilidades prácticas que te ayudarán con los desafíos más comunes de análisis de datos.

16.1.1 Requisitos previos

Este capítulo se centra en el paquete **lubridate**, que simplifica el trabajo con fechas y horas en R. **lubridate** no es parte de los paquetes centrales de **tidyverse** porque solo se necesita al trabajar con fechas/horas. A su vez, necesitaremos los datos sobre `vuelos` contenidos en el paquete **datos**.

```
library(tidyverse)
library(lubridate)
library(datos)
```

Copy

16.2 Creando fechas/horas

Hay tres tipos de datos de fechas/horas que se refieren a un instante en el tiempo:

- Una fecha o `date`. Un *tibble* lo imprime como `<date>`.
- Una hora o `time` dentro de un día. Los *tibbles* lo imprimen como `<time>`.
- Una fecha-hora o `date-time` es una fecha con una hora adjunta: identifica de forma única un instante en el tiempo (típicamente al segundo más cercano). Los *tibbles* imprimen esto como `<dttm>`. En otras partes de R se les llama POSIXct, pero no creemos que sea un nombre muy útil.

En este capítulo solo nos concentraremos en fechas (*dates*) y fechas-horas (*date-times*), ya que R no tiene una clase nativa para almacenar horas. Si necesitas una, puedes usar el paquete **hms**.

Siempre deberías usar el tipo de datos más sencillo que se ajuste a tus necesidades. Esto significa que si puedes usar `date` en lugar de `date-time`, deberías hacerlo. Las fechas-horas son sustancialmente más complicadas porque necesitas gestionar los husos horarios, a los que volveremos al final del capítulo.

Para obtener la fecha o fecha-hora actual, puedes usar `today()` (*hoy*) o `now()` (*ahora*):

```
today()
#> [1] "2021-01-18"
now()
#> [1] "2021-01-18 17:45:10 UTC"
```

Hay tres modos en los que puedes crear una fecha/hora:

- Desde una cadena de caracteres (o *string*, en inglés).
- Desde componentes de fecha-hora individuales.
- Desde un objeto fecha-hora existente.

Estos funcionan de la siguiente manera.

16.2.1 **Desde cadenas de caracteres**

Los datos de fecha/hora a menudo vienen como cadenas de caracteres. Ya has visto una forma de segmentarlas como `date-times` en el capítulo sobre importación de datos. Otra forma es usar las ayudas provistas por **lubridate**. Estas trabajan automáticamente el formato una vez que especificas el orden de los componentes. Para usarlas, identifica el orden en el que el año, mes y día aparecen en tus fechas, y luego ordena "y" (del inglés *year*), "m" (mes) y "d" (día) en el mismo orden. Esto te dará el nombre de la función **lubridate** que segmetará tu fecha. Por ejemplo:

```
ymd("2017-01-31")
#> [1] "2017-01-31"
mdy("Enero 31, 2017")
#> Warning: All formats failed to parse. No formats found.
#> [1] NA
dmy("31-Ene-2017")
#> Warning: All formats failed to parse. No formats found.
#> [1] NA
```

Estas funciones también reciben números sin comillas. Esta es la forma más concisa de crear un único objeto fecha-hora, tal como podrías necesitarla cuando filtras datos temporales. `ymd()` (*año-mes-día*) es corta y no ambigüa:

```
ymd(20170131)
#> [1] "2017-01-31"
```

`ymd()` y sus funciones amigas crean fechas (`date`). Para generar una fecha-hora, agrega un guión bajo y al menos un "h", "m" y "s" al nombre de la función de segmentación:

```
ymd_hms("2017-01-31 20:11:59")
#> [1] "2017-01-31 20:11:59 UTC"
mdy_hm("01/31/2017 08:01")
#> [1] "2017-01-31 08:01:00 UTC"
```

También puedes forzar la creación de una fecha-hora desde una fecha, al proveer un huso horario:

```
ymd(20170131, tz = "UTC")
#> [1] "2017-01-31 UTC"
```

16.2.2 **Desde componentes individuales**

En lugar de una cadena de caracteres simple, a veces tienes los componentes individuales de una fecha-hora repartidos en múltiples columnas. Esto es lo que tenemos en los datos de vuelos:

```
vuelos %>%
  select(anio, mes, dia, hora, minuto)
#> # A tibble: 336,776 x 5
#>    anio   mes   dia  hora minuto
#>   <int> <int> <int> <dbl>  <dbl>
#> 1  2013     1     1     5     15
#> 2  2013     1     1     5     29
#> 3  2013     1     1     5     40
#> 4  2013     1     1     5     45
#> 5  2013     1     1     6      0
#> 6  2013     1     1     5     58
#> # … with 336,770 more rows
```

Para crear una fecha-hora desde este tipo de input, usa `make_date()` (*crear fecha*) para las fechas, o `make_datetime()` (*crear fecha-hora*) para las fechas-horas:

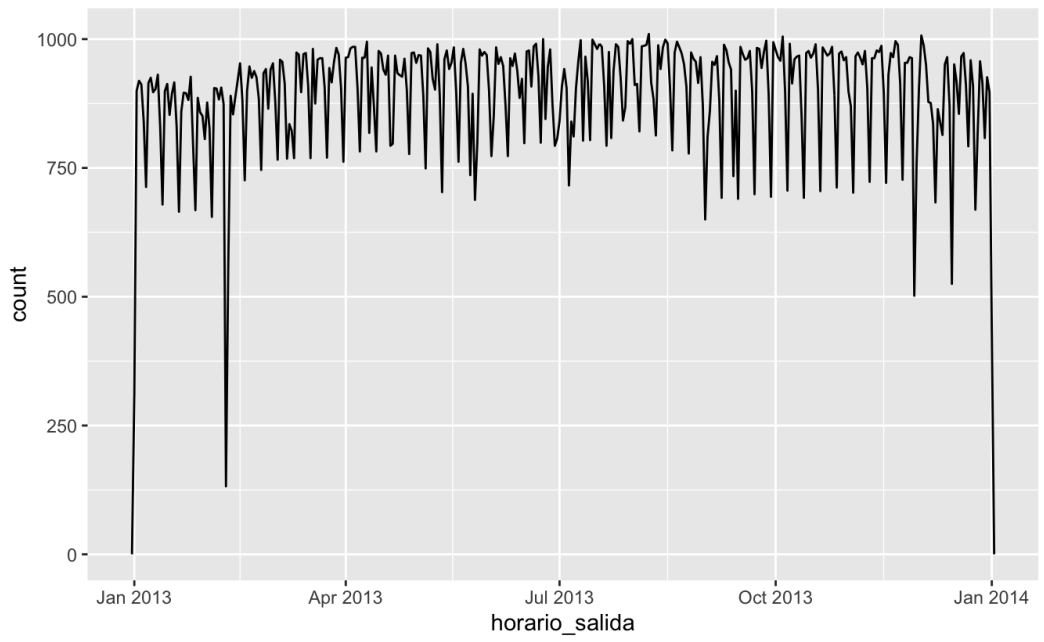
```
vuelos %>%
  select(anio, mes, dia, hora, minuto) %>%
  mutate(salida = make_datetime(anio, mes, dia, hora, minuto))
#> # A tibble: 336,776 x 6
#>    anio   mes   dia  hora minuto salida
#>   <int> <int> <int> <dbl>  <dbl> <dttm>
#> 1  2013     1     1     5     15 2013-01-01 05:15:00
#> 2  2013     1     1     5     29 2013-01-01 05:29:00
#> 3  2013     1     1     5     40 2013-01-01 05:40:00
#> 4  2013     1     1     5     45 2013-01-01 05:45:00
#> 5  2013     1     1     6      0 2013-01-01 06:00:00
#> 6  2013     1     1     5     58 2013-01-01 05:58:00
#> # … with 336,770 more rows
```

Hagamos esto mismo para cada una de las cuatro columnas de tiempo en `vuelos`. Las horas están representadas en un formato ligeramente más extraño, así que usaremos el módulo aritmético para extraer los componentes de horas y minutos. Una vez que hayamos creado las variables fecha-hora, nos centraremos en las variables que usaremos por el resto del capítulo.

```
hacer_fechahora_100 <- function(anio, mes, dia, tiempo) {
  make_datetime(anio, mes, dia, tiempo %/% 100, tiempo %% 100)
}

vuelos_dt <- vuelos %>%
  filter(!is.na(horario_salida), !is.na(horario_llegada)) %>%
  mutate(
    horario_salida = hacer_fechahora_100(anio, mes, dia, horario_salida),
    horario_llegada = hacer_fechahora_100(anio, mes, dia, horario_llegada),
    salida_programada = hacer_fechahora_100(anio, mes, dia, salida_programada),
    llegada_programada = hacer_fechahora_100(anio, mes, dia, llegada_programada)
  ) %>%
  select(origen, destino, starts_with("atraso"), starts_with("horario"), ends_with("programada"),
tiempo_vuelo)

vuelos_dt
#> # A tibble: 328,063 x 9
#>   origen destino atraso_salida atraso_llegada horario_salida
#>   <chr>  <chr>           <dbl>          <dbl> <dttm>
#> 1 EWR    IAH                 2             11 2013-01-01 05:17:00
#> 2 LGA    IAH                 4             20 2013-01-01 05:33:00
#> 3 JFK    MIA                 2             33 2013-01-01 05:42:00
#> 4 JFK    BQN                -1            -18 2013-01-01 05:44:00
#> 5 LGA    ATL                -6            -25 2013-01-01 05:54:00
#> 6 EWR    ORD                -4             12 2013-01-01 05:54:00
#> # … with 328,057 more rows, and 4 more variables: horario_llegada <dttm>,
#> #   salida_programada <dttm>, llegada_programada <dttm>, tiempo_vuelo <dbl>
```

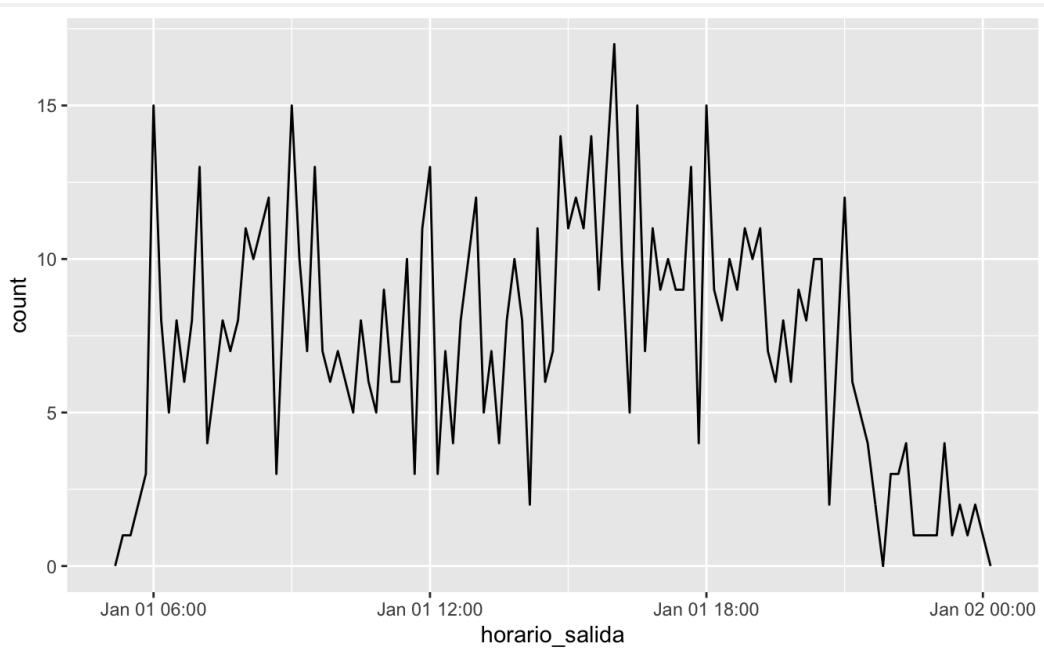
Con estos datos, podemos visualizar la distribución de las horas de salida a lo largo del año:

```
vuelos_dt %>%
  ggplot(aes(horario_salida)) +
  geom_freqpoly(binwidth = 86400) # 86400 segundos = 1 día
```



O para un solo día:

```
vuelos_dt %>%
  filter(horario_salida < ymd(20130102)) %>%
  ggplot(aes(horario_salida)) +
  geom_freqpoly(binwidth = 600) # 600 segundos = 10 minutos
```



Ten en cuenta que cuando usas fechas-hora en un contexto numérico (como en un histograma), 1 significa un segundo, por lo tanto, un `binwidth` (*ancho del contenedor*) de 86400 significa un día. Para las fechas, 1 significa un día.

16.2.3 Desde otros tipos

Puedes querer cambiar entre una fecha-hora y una fecha. Ese es el trabajo de `as_datetime()` (*como fecha-hora*) y `as_date()` (*como fecha*):

```
as_datetime(today())
#> [1] "2021-01-18 UTC"
as_date(now())
#> [1] "2021-01-18"
```

A veces tendrás fechas/horas como desfases numéricos de la "Época Unix" similares a 1970-01-01. Si el desfase es en segundos, usa `as_datetime()` ; si es en días, usa `as_date()` .

```
as_datetime(60 * 60 * 10)
#> [1] "1970-01-01 10:00:00 UTC"
as_date(365 * 10 + 2)
#> [1] "1980-01-01"
```

16.2.4 Ejercicios

1. ¿Qué sucede si analizas una cadena de caracteres que contiene fechas inválidas?

```
ymd(c("2010-10-10", "bananas"))
```

2. ¿Qué hace el argumento `tzone` (*time zone* = *huso horario*) para `today()` ? ¿Por qué es importante?

3. Utiliza la función de **lubridate** apropiada para analizar las siguientes fechas:

```
d1 <- "Enero 1, 2010"
d2 <- "2015-Mar-07"
d3 <- "06-Jun-2017"
d4 <- c("Agosto 19 (2015)", "Julio 1 (2015)")
d5 <- "12/30/14" # Diciembre 30, 2014
```

16.3 Componentes de fecha-hora

Ahora que ya conoces cómo tener datos de fechas y horas en las estructuras de datos de R, vamos a explorar qué puedes hacer con ellos. Esta sección se concentrará en las funciones de acceso (*accessor functions*) que te permiten obtener y configurar componentes individuales. La siguiente sección se centrará en cómo funciona el trabajo con fechas-horas.

16.3.1 Obteniendo los componentes

Puedes obtener las partes individuales de una fecha con las funciones de acceso `year()` (*año*), `month()` (*mes*), `mday()` (*día del mes*), `yday()` (*día del año*), `wday()` (*día de la semana*), `hour()` (*hora*), `minute()` (*minuto*), y `second()` (*segundo*).

```
fechahora <- ymd_hms("2016-07-08 12:34:56")

year(fechahora)
#> [1] 2016
month(fechahora)
#> [1] 7
mday(fechahora)
#> [1] 8

yday(fechahora)
#> [1] 190
wday(fechahora)
#> [1] 6
```

Para `month()` y `wday()` puedes configurar `label = TRUE` para retornar el nombre abreviado del mes o del día de la semana. Usa `abbr = FALSE` para retornar el nombre completo.

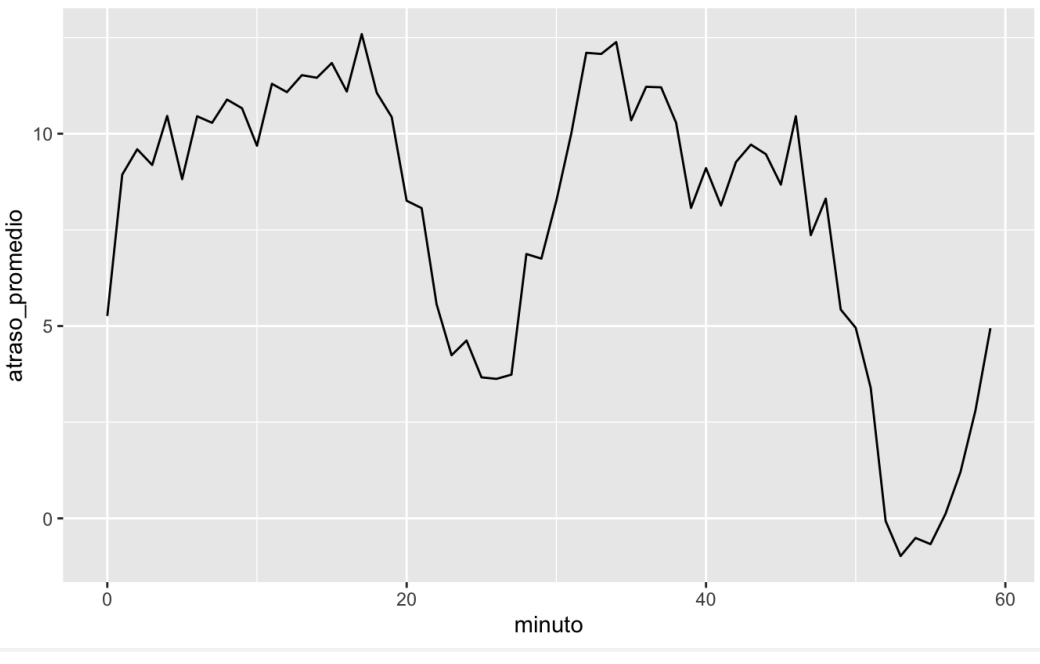
```
month(fechahora, label = TRUE)
#> [1] Jul
#> 12 Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < ... < Dec
wday(fechahora, label = TRUE, abbr = FALSE)
#> [1] Friday
#> 7 Levels: Sunday < Monday < Tuesday < Wednesday < Thursday < ... < Saturday
```

Podemos usar `wday()` para ver que son más los vuelos que salen durante la semana que durante el fin de semana:

```
vuelos_dt %>%
  mutate(dia_semana = wday(horario_salida, label = TRUE)) %>%
  ggplot(aes(x = dia_semana)) +
  geom_bar()
```

Hay un patrón interesante si miramos la demora promedio por minuto dentro de la hora. ¡Parece que los vuelos que salen en los minutos 20-30 y 50-60 tienen mucho menos demora que en el resto de la hora!

```
vuelos_dt %>%
  mutate(minuto = minute(horario_salida)) %>%
  group_by(minuto) %>%
  summarise(
    atraso_promedio = mean(atraso_llegada, na.rm = TRUE),
    n = n()
  ) %>%
ggplot(aes(minuto, atraso_promedio)) +
geom_line()
```



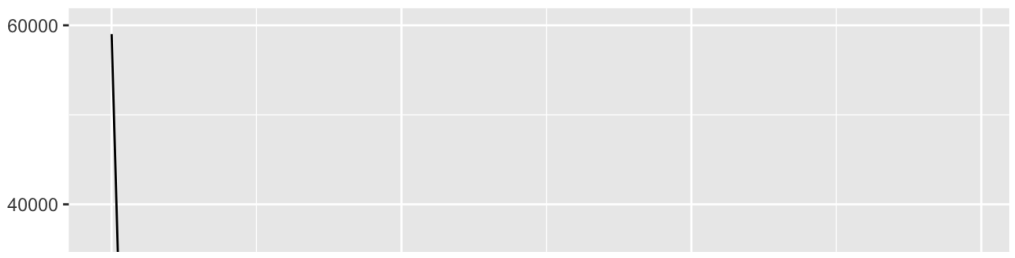
Es interesante que si miramos el horario *programado* de salida, no vemos un patrón tan prominente:

```
salida_programada <- vuelos_dt %>%
  mutate(minuto = minute(salida_programada)) %>%
  group_by(minuto) %>%
  summarise(
    atraso_promedio = mean(atraso_llegada, na.rm = TRUE),
    n = n()
  )

ggplot(salida_programada, aes(minuto, atraso_promedio)) +
  geom_line()
```

Entonces, ¿por qué vemos ese patrón con los horarios reales de salida? Bueno, como muchos datos recolectados por los humanos, hay un sesgo importante hacia los vuelos que salen en horas “agradables”. ¡Mantente siempre alerta respecto a este tipo de patrón cada vez que trabajes con datos que involucran juicio humano!

```
ggplot(salida_programada, aes(minuto, n)) +
  geom_line()
```

Copy



16.3.2 **Redondeo**

Un método alternativo para graficar los componentes individuales es redondear la fecha a una unidad de tiempo cercana, con `floor_date()` (*fecha hacia abajo*), `round_date()` (*redondear fecha*), y `ceiling_date()` (*fecha hacia arriba*). Cada función toma un vector de fechas a ajustar y luego el nombre de la unidad redondeada (con *round*), redondeada hacia abajo (con *floor*) o hacia arriba (con *ceiling*). Esto, por ejemplo, nos permite graficar el número de vuelos por semana:

```
vuelos_dt %>%
  count(semana = floor_date(horario_salida, "week")) %>%
  ggplot(aes(semana, n)) +
  geom_line()
```

Copy

Calcular la diferencia entre una fecha redondeada y una sin redondear puede ser particularmente útil.

16.3.3 **Configurando componentes**

También puedes usar las funciones de acceso para darle un valor a los componentes de las fechas/horas:

```
(fechahora <- ymd_hms("2016-07-08 12:34:56"))
#> [1] "2016-07-08 12:34:56 UTC"

year(fechahora) <- 2020
fechahora
#> [1] "2020-07-08 12:34:56 UTC"
month(fechahora) <- 01
fechahora
#> [1] "2020-01-08 12:34:56 UTC"
hour(fechahora) <- hour(fechahora) + 1
fechahora
#> [1] "2020-01-08 13:34:56 UTC"
```

Alternativamente, en lugar de modificar en un solo lugar, puedes crear una nueva fecha-hora con `update()` (*actualizar*). Esto también te permite configurar múltiples valores al mismo tiempo.

```
update(fechahora, year = 2020, month = 2, mday = 2, hour = 2)
#> [1] "2020-02-02 02:34:56 UTC"
```

Si los valores son demasiado grandes, darán la vuelta:

```
ymd("2015-02-01") %>% update(mday = 30)
#> [1] "2015-03-02"
ymd("2015-02-01") %>% update(hour = 400)
#> [1] "2015-02-17 16:00:00 UTC"
```

Puedes utilizar `update()` para mostrar la distribución de los vuelos a lo largo del día para cada día del año:

```
vuelos_dt %>%
  mutate(horario_salida = update(horario_salida, yday = 1)) %>%
  ggplot(aes(horario_salida)) +
  geom_freqpoly(binwidth = 300)
```

Fijar los componentes más grandes de una fecha con una constante es una técnica que te permite explorar patrones en los componentes más pequeños.

16.3.4 Ejercicios

1. ¿Cómo cambia la distribución de las horas de los vuelos dentro de un día a lo largo del año?
2. Compara `horario_salida`, `salida_programada` y `atraso_salida`. ¿Son consistentes? Explica tus hallazgos.
3. Compara `tiempo_vuelo` con la duración entre la salida y la llegada. Explica tus hallazgos. (Pista: considera la ubicación del aeropuerto).
4. ¿Cómo cambia la demora promedio durante el curso de un día? ¿Deberías usar `horario_salida` o `salida_programada`? ¿Por qué?
5. ¿En qué día de la semana deberías salir si quieres minimizar las posibilidades de una demora?
6. ¿Qué hace que la distribución de `diamantes$quilate` y `vuelos$salida_programada` sea similar?
7. Confirma nuestra hipótesis de que las salidas programadas en los minutos 20-30 y 50-60 están casuadas por los vuelos programados que salen más temprano. Pista: crea una variable binaria que te diga si un vuelo tuvo o no demora.

16.4 Lapsos de tiempo

Ahora, aprenderás cómo trabaja la aritmética con fechas, incluyendo la sustracción, adición y división. En el camino, aprenderás sobre tres importantes clases que representan períodos de tiempo:

- **durations** (*duraciones*), que representa un número exacto de segundos.
- **periods** (*períodos*), que representan unidades humanas como semanas o meses.

- **intervals** (*intervalos*), que representan un punto de inicio y uno de finalización.

16.4.1 Duraciones

Cuando restas dos fechas en R obtienes un objeto de diferencia temporal (en inglés, *difftimes*):

```
# ¿Qué edad tiene Hadley?
edad_h <- today() - ymd(19791014)
edad_h
#> Time difference of 15072 days
```

Un objeto de clase *difftime* registra un lapso de tiempo de segundos, minutos, horas, días o semanas. Esta ambiguedad hace que los *difftimes* sean un poco complicados de trabajar, por lo que **lubridate** provee una alternativa que siempre usa segundos: la **duración**.

```
as.duration(edad_h)
#> [1] "1302220800s (~41.26 years)"
```

Las duraciones traen un conveniente grupo de constructores:

```
dseconds(15)
#> [1] "15s"
dminutes(10)
#> [1] "600s (~10 minutes)"
dhours(c(12, 24))
#> [1] "43200s (~12 hours)" "86400s (~1 days)"
ddays(0:5)
#> [1] "0s"                "86400s (~1 days)"  "172800s (~2 days)"
#> [4] "259200s (~3 days)" "345600s (~4 days)" "432000s (~5 days)"
dweeks(3)
#> [1] "1814400s (~3 weeks)"
dyears(1)
#> [1] "31557600s (~1 years)"
```

Las duraciones siempre registran el lapso de tiempo en segundos. Las unidades más grandes se crean al convertir minutos, horas, días, semanas y años a segundos, mediante una conversión estándar (60 segundos en un minuto, 60 minutos en una hora, 24 horas en un día, 7 días en una semana, 365 días en un año).

Puedes agregar y multiplicar duraciones:

```
2 * dyears(1)
#> [1] "63115200s (~2 years)"
dyears(1) + dweeks(12) + dhours(15)
#> [1] "38869200s (~1.23 years)"
```

Puedes sumar y restar duraciones a días:

```
ayer <- today() + ddays(1)
anio_pasado <- today() - dyears(1)
```

Sin embargo, como las duraciones representan un número exacto de segundos, a veces puedes obtener un resultado inesperado:

```
una_pm <- ymd_hms("2016-03-12 13:00:00", tz = "America/New_York")

una_pm
#> [1] "2016-03-12 13:00:00 EST"
una_pm + ddays(1)
#> [1] "2016-03-13 14:00:00 EDT"
```

¡¿Por qué un día después de la 1 pm del 12 de marzo son las 2 pm del 13 de marzo!? Si miras con cuidado la fecha, te darás cuenta de que los husos horarios han cambiado. Debido al horario de verano (EDT es el horario de verano de la Costa Este de EE. UU.), el 12 de marzo solo tiene 23 horas, por lo que si agregamos un día entero de segundos terminamos con una hora diferente.

16.4.2 Períodos

Para resolver este problema, **lubridate** provee **periodos**. Estos son plazos de tiempo que no tienen un largo fijo en segundos, sino que funcionan con tiempos "humanos", como días o meses. Esto les permite trabajar en una forma más intuitiva:

```
una_pm
#> [1] "2016-03-12 13:00:00 EST"
una_pm + days(1)
#> [1] "2016-03-13 13:00:00 EDT"
```

Copy

Al igual que las duraciones, los períodos pueden ser creados mediante un número de funciones constructoras amigables.

```
seconds(15)
#> [1] "15S"
minutes(10)
#> [1] "10M 0S"
hours(c(12, 24))
#> [1] "12H 0M 0S" "24H 0M 0S"
days(7)
#> [1] "7d 0H 0M 0S"
months(1:6)
#> [1] "1m 0d 0H 0M 0S" "2m 0d 0H 0M 0S" "3m 0d 0H 0M 0S" "4m 0d 0H 0M 0S"
#> [5] "5m 0d 0H 0M 0S" "6m 0d 0H 0M 0S"
weeks(3)
#> [1] "21d 0H 0M 0S"
years(1)
#> [1] "1y 0m 0d 0H 0M 0S"
```

Copy

Puedes sumar y multiplicar períodos:

```
10 * (months(6) + days(1))
#> [1] "60m 10d 0H 0M 0S"
days(50) + hours(25) + minutes(2)
#> [1] "50d 25H 2M 0S"
```

Copy

Y, por supuesto, puedes sumarlos a las fechas. Comparados con las duraciones, los períodos son más propensos a hacer lo que esperas que hagan:

```
# Un año bisiesto
ymd("2016-01-01") + dyears(1)
#> [1] "2016-12-31 06:00:00 UTC"
ymd("2016-01-01") + years(1)
#> [1] "2017-01-01"

# Horarios de verano
una_pm + ddays(1)
#> [1] "2016-03-13 14:00:00 EDT"
una_pm + days(1)
#> [1] "2016-03-13 13:00:00 EDT"
```

Copy

Usemos los períodos para arreglar una rareza relacionada a nuestras fechas de vuelos. Algunos aviones parecen arribar a su destino *antes* de salir de la ciudad de Nueva York.

```
vuelos_dt %>%
  filter(horario_llegada < horario_salida)
#> # A tibble: 10,633 x 9
#>   origen destino atraso_salida atraso_llegada horario_salida
#>   <chr>  <chr>           <dbl>          <dbl> <dttm>
#> 1 EWR    BQN                 9             -4 2013-01-01 19:29:00
#> 2 JFK    DFW                59             NA 2013-01-01 19:39:00
#> 3 EWR    TPA                -2              9 2013-01-01 20:58:00
#> 4 EWR    SJU                -6            -12 2013-01-01 21:02:00
#> 5 EWR    SFO                11            -14 2013-01-01 21:08:00
#> 6 LGA    FLL               -10             -2 2013-01-01 21:20:00
#> # … with 10,627 more rows, and 4 more variables: horario_llegada <dttm>,
#> #   salida_programada <dttm>, llegada_programada <dttm>, tiempo_vuelo <dbl>
```

Estos son vuelos nocturnos. Usamos la misma información de fecha para los horarios de salida y llegada, pero estos vuelos llegaron al día siguiente. Podemos arreglarlo al sumar `days(1)` a la fecha de llegada de cada vuelo nocturno.

```
vuelos_dt <- vuelos_dt %>%
  mutate(
    nocturno = horario_llegada < horario_salida,
    horario_llegada = horario_llegada + days(nocturno * 1),
    llegada_programada = llegada_programada + days(nocturno * 1)
  )
```

Ahora todos los vuelos obedecen a las leyes de la física.

```
vuelos_dt %>%
  filter(nocturno, horario_llegada < horario_salida)
#> # A tibble: 0 x 10
#> # … with 10 variables: origen <chr>, destino <chr>, atraso_salida <dbl>,
#> #   atraso_llegada <dbl>, horario_salida <dttm>, horario_llegada <dttm>,
#> #   salida_programada <dttm>, llegada_programada <dttm>, tiempo_vuelo <dbl>,
#> #   nocturno <lgl>
```

16.4.3 Intervalos

Resulta obvio lo que `dyears(1) / ddays(365)` debería retornar: uno, porque las duraciones siempre se representan por un número de segundos y la duración de un año se define como 365 días convertidos a segundos.

¿Qué debería devolver `years(1) / days(1)` ? Bueno, si el año fuera 2015 debería retornar 365, ¡pero si fuera 2016 debería retornar 366! No hay suficiente información para que **lubridate** nos dé una sola respuesta sencilla. Por ello, lo que hace es darnos una estimación con una advertencia:

```
years(1) / days(1)
#> [1] 365.25
```

Si quieres una medida más precisa, tendrás que usar un **intervalo**. Un intervalo es una duración con un punto de partida: eso lo hace preciso, por lo que puedes determinar exactamente cuán largo es:

```
siguiente_anio <- today() + years(1)
(today() %--% siguiente_anio) / ddays(1)
#> [1] 365
```

Para encontrar cuántos períodos caen dentro de un intervalo, tienes que usar la división entera:

```
(today() %--% siguiente_anio) %/% days(1)
#> [1] 365
```

16.4.4 Resumen

¿Cómo eliges entre duraciones, períodos e intervalos? Como siempre, selecciona la estructura de datos más sencilla que resuelva tu problema. Si solo te interesa el tiempo físico, usa una duración; si necesitas agregar tiempos humanos, usa un período; si tienes que deducir cuán largo es un lapso de tiempo en unidades humanas, usa un intervalo.

La figura 16.1 resume las operaciones artiméticas permitidas entre los tipos de datos.

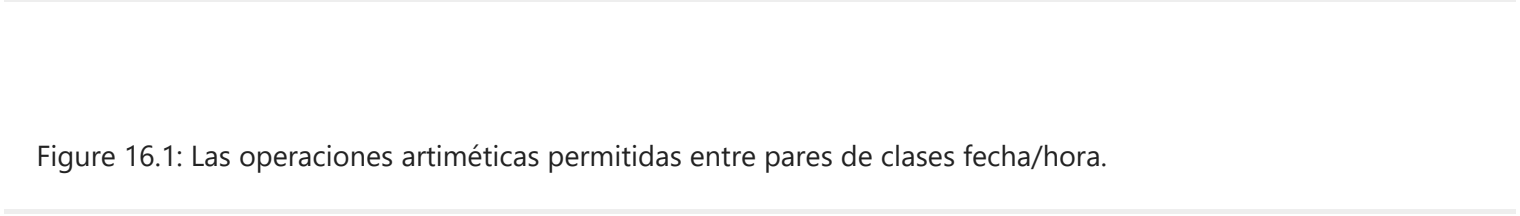


Figure 16.1: Las operaciones artiméticas permitidas entre pares de clases fecha/hora.

16.4.5 Ejercicios

1. ¿Por qué hay `months()` pero no `dmonths()` (*días del mes*)?

2. Explica `days(nocturno * 1)` a alguien que apenas comienza a aprender R. ¿Cómo funciona?

3. Crea un vector de fechas dando el primer día de cada mes de 2015. Crea un vector de fechas dando el primer día de cada mes del año *actual*.

4. Crea una función en la que, dado tu cumpleaños (como una fecha), retorne qué edad tienes en años.

5. ¿Por qué no funciona `(today() %--% (today() + years(1)) / months(1)` ?

16.5 Husos horarios

Los husos horarios son un tema enormemente complicado debido a su interacción con entidades geopolíticas. Afortunadamente, no necesitamos escarbar en todos los detalles, ya que no todos son necesarios para el análisis de datos. Sin embargo, hay algunos desafíos que tendremos que enfrentar.

El primer desafío es que los nombres comunes de los husos horarios tienden a ser ambiguos. Por ejemplo, si eres estadounidense, probablemente te sea familiar la sigla EST, (del inglés de *Tiempo Este Estándar*). Sin embargo, ¡Canadá y Australia también tienen EST! Para evitar la confusión, R usa el estándar internacional IANA para husos horarios. Estos tienen un esquema de nombres que sigue el formato "/", típicamente escrito como "<continente>/<ciudad>", en idioma inglés (hay algunas pocas excepciones porque no todos los países están sobre un continente). Algunos ejemplos: "America/New_York", "Europe/Paris", "Pacific/Auckland", "America/Bogota".

Puede que te preguntes por qué un huso horario usa una ciudad, cuando típicamente piensas en ellos como asociados a un país o a una región dentro de un país. Esto se debe a que la base de datos de IANA tiene que registrar décadas de reglamentos sobre husos horarios. En el curso de las décadas, los países cambian nombres (o desaparecen) de forma bastante frecuente, pero los nombres de las ciudades tienden a mantenerse igual. Otro problema es que los nombres tienen que reflejar no solo el comportamiento actual, sino también la historia completa. Por ejemplo, hay husos horarios tanto para "America/New_York" como para "America/Detroit". Actualmente, ambas ciudades usan el EST, pero entre 1969 y 1972, Michigan (el estado en el que está ubicado Detroit), no empleaba el horario de verano, así que necesita un nombre diferente. ¡Vale la pena leer la base de datos sobre husos horarios (disponible en http://www.iana.org/time-zones) solo para enterarse de algunas de estas historias!

Puedes encontrar cuál es tu huso horario actual para R, usando `Sys.timezone()`:

```
Sys.timezone()
#> [1] "UTC"
```

(Si R no lo sabe, obtendrás un `NA`.)

Y puedes ver la lista completa de todos los husos horarios con `OlsonNames()`:

```
length(OlsonNames())
#> [1] 594
head(OlsonNames())
#> [1] "Africa/Abidjan"     "Africa/Accra"       "Africa/Addis_Ababa"
#> [4] "Africa/Algiers"     "Africa/Asmara"      "Africa/Asmera"
```

En R, el huso horario es un atributo de la fecha-hora (`date-time`) que solo controla la impresión. Por ejemplo, estos tres objetos representan el mismo instante en el tiempo:

```
(x1 <- ymd_hms("2015-06-01 12:00:00", tz = "America/New_York"))
#> [1] "2015-06-01 12:00:00 EDT"
(x2 <- ymd_hms("2015-06-01 18:00:00", tz = "Europe/Copenhagen"))
#> [1] "2015-06-01 18:00:00 CEST"
(x3 <- ymd_hms("2015-06-02 04:00:00", tz = "Pacific/Auckland"))
#> [1] "2015-06-02 04:00:00 NZST"
```

Puedes verificar que son lo mismo al usar una resta:

```
x1 - x2
#> Time difference of 0 secs
x1 - x3
#> Time difference of 0 secs
```

Excepto que se especifique otra cosa, **lubridate** siempre usa UTC. UTC (*Tiempo Universal Coordinado*) es el huso horario estándar empleado por la comunidad científica y es aproximadamente equivalente a su predecesor GMT (siglas en inglés de *Tiempo del Meridiano de Greenwich*). UTC no tiene horario de verano, por lo que resulta una representación conveniente para la computación. Las operaciones que combinan fechas y horas, como `c()` , a menudo descartan el huso horario. En ese caso, las fechas y horas se muestran en tu huso local:

```
x4 <- c(x1, x2, x3)
x4
#> [1] "2015-06-01 12:00:00 EDT" "2015-06-01 12:00:00 EDT"
#> [3] "2015-06-01 12:00:00 EDT"
```

Puedes cambiar el huso horario de dos formas:

- Mantener el instante en el tiempo igual y cambiar solo cómo se representa. Usa esto cuando el instante es correcto, pero quieres una visualización más natural.

  ```
  x4a <- with_tz(x4, tzone = "Australia/Lord_Howe")
  x4a
  #> [1] "2015-06-02 02:30:00 +1030" "2015-06-02 02:30:00 +1030"
  #> [3] "2015-06-02 02:30:00 +1030"
  x4a - x4
  #> Time differences in secs
  #> [1] 0 0 0
  ```

  (Esto también ilustra otro desafío de los husos horarios: ¡no todos los desfasajes son horas como números enteros!)

- Cambia el instante en el tiempo subyacente. Usa esto cuando tienes un instante que ha sido etiquetado con un huso horario incorrecto y necesitas arreglarlo.

  ```
  x4b <- force_tz(x4, tzone = "Australia/Lord_Howe")
  x4b
  #> [1] "2015-06-01 12:00:00 +1030" "2015-06-01 12:00:00 +1030"
  #> [3] "2015-06-01 12:00:00 +1030"
  x4b - x4
  #> Time differences in hours
  #> [1] -14.5 -14.5 -14.5
  ```

« 15 Factores

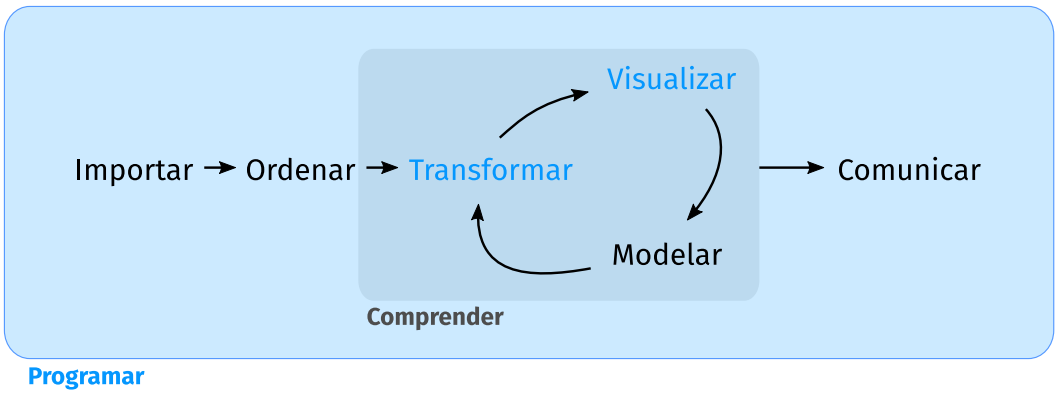
17 Introducción »

"" was written by .

This book was built by the bookdown R package.

17 Introducción

En esta parte del libro mejorarás tus habilidades de programación. La programación es una destreza transversal en todo el trabajo en ciencia de datos: es necesario usar una computadora para hacer ciencia de datos, no puedes hacerla solo con tu cabeza o con lápiz y papel.



Programar produce código y el código es una herramienta de comunicación. Obviamente, el código le dice a la computadora qué es lo que quieres que haga, pero también comunica significado a otros seres humanos. Es importante pensar el código como un medio de comunicación, ya que todo proyecto que realices es esencialmente colaborativo. Aun cuando no estés trabajando con otras personas, definitivamente lo estarás haciendo con tu futuro yo. Escribir código claro es importante para que otras personas (o tú en el futuro) puedan entender por qué encaraste un análisis de la manera que lo hiciste. Esto significa que mejorando cómo programas mejorarás también cómo comunicas. Con el tiempo querrás que tu código resulte no solo más fácil de escribir, sino también más fácil de leer para los demás.

Escribir código es similar en muchas formas a escribir prosa. Un paralelismo que encuentramos particularmente útil es que en ambos casos reescribir es la clave para la claridad. Es poco probable que la primera expresión de tus ideas sea particularmente clara, por lo que es posible que necesites reescribir muchas veces. Después de resolver un problema de análisis de datos, en general vale la pena mirar tu código y pensar si es obvio o no lo que has hecho. Si dedicas un poco de tiempo a reescribir tu código mientras las ideas están frescas, puedes ganar mucho tiempo después, cuando necesites recrear lo que tu código hizo. Esto no significa que debas reescribir todas las funciones: necesitas encontrar un equilibrio entre lo que necesitas lograr ahora y ahorrar tiempo a largo plazo. (Aunque cuanto más reescribas tus funciones, más probable será que tu primer intento sea claro).

En los próximos cuatro capítulos aprenderás habilidades que te permitirán tando abordar nuevos programas como resolver problemas ya existente con mayor claridad y facilidad:

1. En Pipes, navegarás dentro del **pipe**, `%>%`, y aprenderás más sobre cómo trabaja, qué alternativas existen y cuándo no conviene usarlo.

2. Si bien copiar-y-pegar (*copy-and-paste*) es una herramienta poderosa, deberías evitar utilizarla más de dos veces. Repetir el código es peligroso porque puede llevar a errores e inconsistencias. Por eso, en Funciones aprenderás a escribir **funciones** que permitan extraer código repetido para que pueda ser reutilizado con facilidad.

3. A medida que comiences a escribir funciones más potentes, necesitarás una base sólida acerca de las **estructuras de datos** de R que te la proporcionará Vectores. Deberás dominar los cuatro tipos de vectores atómicos clásicos, las tres clases S3 construídas sobre ellos y entender los misterios de la lista y el *data frame*.

4. Las funciones nos permiten evitar la repetición de código; sin embargo, muchas veces necesitas repertir las mismas acciones con diferentes inputs. Puedes hacer esto con herramientas de **iteración**, las que te permitan hacer cosas similares una y otra vez. Estas herramientas incluyen bucles (*loops*) y programación funcional, acerca de las cuales aprenderás en Iteración.

17.1 Aprendiendo más

El objetivo de estos capítulos es enseñarte lo mínimo acerca de programación que necesitas para hacer ciencia de datos, lo que resulta una cantidad considerable de información. Una vez que hayas dominado el material de este libro, creemos firmemente que deberías invertir tiempo en mejorar tus habilidades de programación. Aprender más sobre programación es una inversión a largo plazo: no obtendrás resultados inmediatos, pero a la larga te permitirá resolver nuevos problemas más rápidamente y te permitirá reutilizar el conocimiento que adquiriste en nuevos escenarios.

Para profundizar necesitas estudiar R como un lenguaje de programación, no solo como un ambiente interactivo para ciencia de datos. Escribimos dos libros que te ayudarán con esto, aunque actualmente están disponibles solo en inglés:

- *Hands on Programming with R*, de Garrett Grolemund. Esta es una introducción a R como lenguaje de programación y es un buen lugar para empezar si R es el primer lenguaje que aprendes. Cubre casi el mismo material que estos capítulos, pero con un estilo diferente y con distintos ejemplos de motivación (basados en el casino). Es un complemento muy útil si consideras que estos cuatros capítulos van demasiado rápido.

- *Advanced R* de Hadley Wickham. Este libro se sumerge en los detalles del lenguaje de programación R. Este es un buen punto por donde empezar si ya tienes experiencia en programación. Es también una buena manera de seguir una vez que hayas internalizado las ideas de estos cuatro capítulos. Puedes leerlo online en el sitio https://adv-r.hadley.nz/.

« 16 Fechas y horas

18 Pipes »

18 Pipes

18.1 Introducción

Los pipes son una herramienta poderosa para expresar claramente una secuencia de múltiples operaciones. Hasta aquí, has venido usándolos sin saber cómo funcionan o qué alternativas existen. En este capítulo ya es tiempo de explorarlos en más detalle. En él aprenderás qué alternativas existen, cuándo no deberías utilizarlos y algunas herramientas útiles relacionadas.

18.1.1 Prerequisitos

El pipe, `%>%`, viene del paquete **magrittr** de Stefan Milton Bache. Los paquetes del Tidyverse cargan `%>%` automáticamente, por lo que usualmente no tendrás que cargar **magrittr** de forma explícita. Sin embargo, como acá nos enfocaremos en el uso de pipes y no usaremos otros paquetes del tidyverse, lo cargaremos explícitamente.

```
library(magrittr)
library(datos)
```

Copy

18.2 Alternativas a los pipes

El objetivo de un pipe es ayudarte a escribir código de una manera que sea más fácil de leer y entender. Para ver por qué un pipe es tan útil, vamos a explorar diferentes formas de escribir el mismo código. Usemos código para contar una historia acerca de un pequeño conejito llamado Foo Foo:

> El pequeño conejito Foo Foo Fue saltando por el bosque Recogiendo ratones del campo Y golpeándolos en la cabeza

Este es un poema popular para niños que se acompaña mediante gestos con las manos.

Empezaremos por definir un objeto que represente al pequeño conejito Foo Foo:

```
foo_foo <- pequeño_conejito()
```

Copy

Y usaremos una función para cada verbo clave: `saltar()`, `recoger()`, y `golpear()`. Usando este objeto y estos verbos, existen (al menos) cuatro maneras en las que podemos volver a contar la historia en código:

1. Guardar cada paso intermedio como un nuevo objeto.
2. Sobreescribir el objeto original muchas veces.
3. Componer funciones.
4. Usar un pipe.

Desarrollaremos cada uno de estos enfoques, mostrándote el código y hablando de las ventajas y desventajas.

18.2.1 Pasos intermedios

El enfoque más sencillo es guardar cada paso como un nuevo objeto:

```
foo_foo_1 <- saltar(foo_foo, a_traves = bosque)
foo_foo_2 <- recoger(foo_foo_1, que = ratones_del_campo)
foo_foo_3 <- golpear(foo_foo_2, en = cabeza)
```

Copy

La principal desventaja de esta manera es que te obliga a nombrar cada paso intermedio. Si hay nombres naturales, es una buena idea y deberías hacerlo. Pero muchas veces, como en este ejemplo, no hay nombres naturales, por lo que agregas sufijos numéricos para hacer los nombres únicos. Esto conduce a

dos problemas:

1. El código está abarrotado con nombres poco importantes.

2. Hay que incrementar cuidadosamente el sufijo en cada línea.

Cada vez que escribimos código como este, nos ocurre que inevitablemente usamos el número incorrecto en una línea y luego perdemos 10 minutos rascándonos la cabeza tratándo de darnos cuenta por qué no funciona. Probablemente te preocupa también que esta forma crea muchas copias de tus datos y ocupa demasiada memoria. Sorprendentemente, este no es el caso. Primero, ten en cuenta que preocuparse proactivamente de la memoria no es una manera útil de invertir tu tiempo: preocúpate acerca de ello cuando se convierta en un problema (es decir, cuando te quedes sin memoria), no antes. En segundo lugar, R no es estúpido y compartirá las columnas a lo largo de los dataframes cuando sea posible. Echemos un vistazo a un pipe de manipulación de datos en el que agregamos una nueva columna a `datos::diamantes`:

```
diamantes2 <- diamantes %>%
  dplyr::mutate(precio_por_quilate = precio / quilate)

pryr::object_size(diamantes)
#> Registered S3 method overwritten by 'pryr':
#>   method      from
#>   print.bytes Rcpp
#> 3.46 MB
pryr::object_size(diamantes2)
#> 3.89 MB
pryr::object_size(diamantes, diamantes2)
#> 3.89 MB
```

Copy

`pryr::object_size()` (*tamaño de objeto*) devuelve la cantidad de memoria ocupada por todos los argumentos de un objeto. Los resultados parecen contraintuitivos al principio:

- `diamantes` ocupa 3.46 MB,
- `diamantes2` ocupa 3.89 MB,
- ¡ `diamantes` y `diamantes2` juntos ocupan 3.89 MB!

¿Cómo es que funciona esto? Bien, `diamantes2` tiene 10 columnas en común con `diamantes`: no hay necesidad de duplicar los datos, así que ambos data frames tienen variables en común. Estas variables solo serán copiadas si modificas una de ellas. En el siguiente ejemplo, modificamos un solo valor en `diamantes$quilate`. Esto significa que la variable `quilate` no podrá ser compartida entre los dos data frames, por lo que se debe realizar una copia. El tamaño de cada data frame no cambia, pero el tamaño colectivo se incrementa:

```
diamantes$quilate[1] <- NA
pryr::object_size(diamantes)
#> 3.46 MB
pryr::object_size(diamantes2)
#> 3.89 MB
pryr::object_size(diamantes, diamantes2)
#> 4.32 MB
```

Copy

(Fíjate que aquí usamos `pryr::object_size()`, no la versión de `object.size()` ya precargada. `object.size()` toma un solo objeto, por lo que no puede computar cómo los datos son compartidos a través de múltiples objetos.)

18.2.2 Sobrescribir el original

En vez de crear objetos en cada paso intermedio, podemos sobrescribir el objeto original:

```
foo_foo <- saltar(foo_foo, a_traves = bosque)
foo_foo <- recoger(foo_foo, que = ratones_del_campo)
foo_foo <- golpear(foo_foo, en = cabeza)
```

Copy

Esto es menos tipeo (y menos que pensar), así que es menos probable que cometas errores. Sin embargo, hay dos problemas:

1. Depurar es doloroso: si cometes un error vas a necesitar correr de nuevo todo el código desde el principio.

2. La repetición del objeto a medida que es transformado (¡hemos escrito foo_foo 6 veces!) hace poco transparente lo que está siendo cambiado en cada línea.

18.2.3 **Composición de funciones**

Otro enfoque es abandonar la asignación y encadenar todas las llamadas a las funciones:

```
saltar(
  recoger(
    golpear(foo_foo, por_el = bosque),
    arriba = raton_de_campo
  ),
  en = la_cabeza
)
```

Copy

Aquí la desventaja es que se debe leer de adentro hacia afuera, de derecha a izquierda y que los argumentos terminan separados (problema que se conoce como dagwood sandwich). En resumen, este código es difícil leer para un ser humano.

18.2.4 **Uso de pipe**

Finalmente, podemos usar el pipe:

```
foo_foo %>%
  saltar(a_través = bosque) %>%
  recoger(que = ratones_campo) %>%
  golpear(en = cabeza)
```

Copy

Esta es nuestra forma preferida, ya que se enfoca en los verbos, no en los sustantivos. Puedes leer esta secuencia de composición de funciones como si fuera un conjunto de acciones imperativas. Foo salta, luego recoge, luego golpea. La desventaja, por supuesto, es que necesitas estar familiarizado con el uso de los pipes. Si nunca has visto `%>%` antes, no tendrás idea acerca de lo que realiza el código. Afortunadamente, la mayoría de la gente entiende la idea fácilmente, así que cuando compartes tu código con otros que no están familiarizados con los pipes, puedes enseñárselos fácilmente.

El pipe trabaja realizando una "transformación léxica": detrás de escena, magrittr reensambla el código en el pipe a una forma que funciona sobrescribiendo un objeto intermedio. Cuando se ejecuta un pipe como el de arriba, magrittr hace algo como esto:

```
mi_pipe <- function(.) {
  . <- saltar(., a_traves = bosque)
  . <- recoger(., que = ratones_campo)
  golpear(., en = la_cabeza)
}
mi_pipe(foo_foo)
```

Copy

Esto significa que un pipe no funcionará con dos clases de funciones: 1. Funciones que usan el entorno actual. Por ejemplo, `assign()` (*asignar*) creará una nueva variable con el nombre dado en el entorno actual:

```
assign("x", 10)
x
#> [1] 10

"x" %>% assign(100)
x
#> [1] 10
```

Copy

El uso de la asignación con el pipe no funcionará porque lo asigna a un entorno temporal usado por `%>%`. Si quieres usar la asignación con un pipe, debes explicitar el entorno:

```
env <- environment()
"x" %>% assign(100, envir = env)
x
#> [1] 100
```

Otras funciones con este problema incluyen `get()` y `load()`.

1. Funciones que usan *lazy evaluation* (evaluación diferida o "perezosa"). En R, los argumentos de las funciones son solamente computados cuando la función los usa, no antes de llamar a la función. El pipe computa cada elemento por turno, por lo que no puedes confiar en este comportamiento.

Un caso en el cual esto es un problema es el de `tryCatch()`, que te permite capturar y manejar errores:

```
tryCatch(stop("!"), error = function(e) "Un error")
#> [1] "Un error"

stop("!") %>%
  tryCatch(error = function(e) "Un error")
#> [1] "Un error"
```

Hay una cantidad relativamente grande de funciones con este comportamiento, que incluye a `try()`, `suppressMessages()` y `suppressWarnings()` en R base.

18.3 Cuándo no usar el pipe

El pipe es una herramienta poderosa, pero no es la única herramienta a tu disposición ¡y no soluciona todos los problemas! Los pipes son mayoritariamente usados para reescribir una secuencia lineal bastante corta de operaciones. Creemos que deberías buscar otra herramienta cuando:

- Tus pipes son más largos que (digamos) 10 pasos. En ese caso, crea objetos intermedios con nombres significativos. Esto hará la depuración más fácil, porque puedes chequear con mayor facilidad los resultados intermedios. Además, hace más simple entender tu código, ya que los nombres de las variables pueden ayudar a comunicar la intención.
- Tienes múltiples *inputs* y *outputs*. Si no hay un objeto principal para ser transformado, sino dos o más objetos siendo combinados juntos, no uses el pipe.
- Si estás empezando a pensar acerca de un grafo dirigido con una estructura de dependencia compleja. Los pipes son fundamentalmente lineales y usarlos para expresar relaciones complejas, típicamente llevarán a un código confuso.

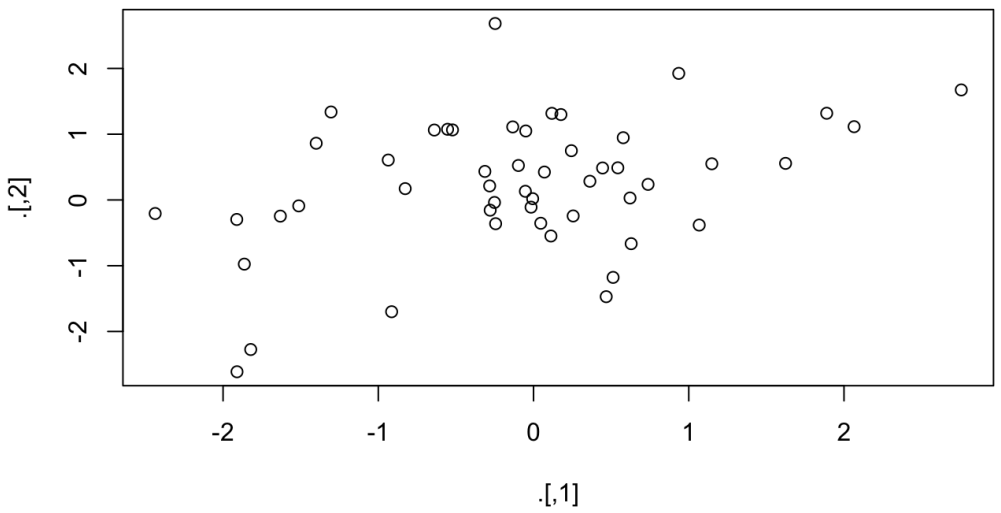
18.4 Otras herramientas de magrittr

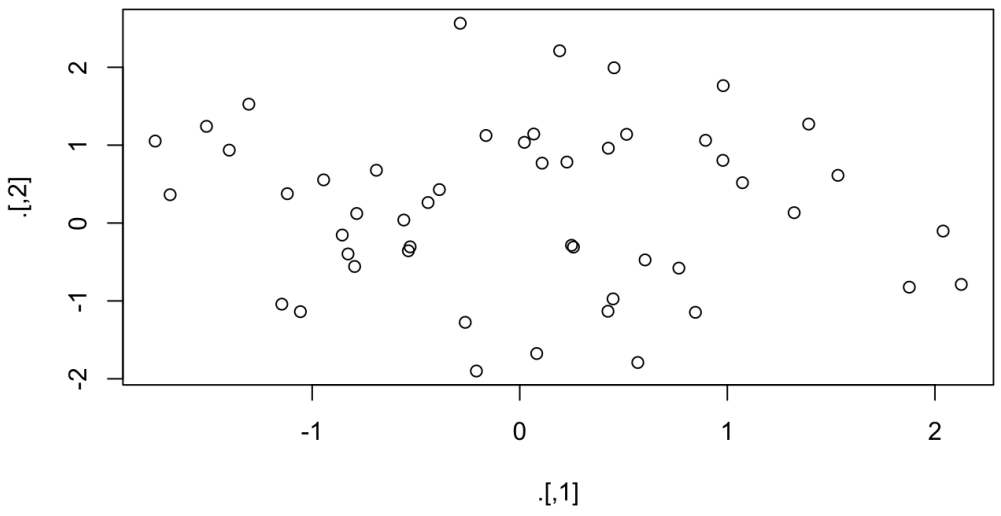
Todos los paquetes del tidyverse automáticamente harán que `%>%` esté disponible, por lo que normalmente no será necesario cargar **magrittr** explícitamente. De todas formas, hay otras herramientas útiles dentro de magrittr que podrías querer utilizar:

- Cuando se trabaja en un pipe más complejo, a veces es útil llamar a una función por sus efectos secundarios. Tal vez quieras imprimir el objeto actual, o graficarlo, o guardarlo en el disco. Muchas veces, estas funciones no devuelven nada, efectivamente terminando el pipe.

Para solucionar este problema, puedes usar el pipe "T". `%T>%` trabaja igual que `%>%`, excepto que devuelve el lado izquierdo en vez del lado derecho. Se lo llama "T" porque literalmente tiene la forma de una T.

```
rnorm(100) %>%
  matrix(ncol = 2) %>%
  plot() %>%
  str()
#>  NULL

rnorm(100) %>%
  matrix(ncol = 2) %T>%
  plot() %>%
  str()
#>  num [1:50, 1:2] -0.387 -0.785 -1.057 -0.796 -1.756 ...
```





- Si estás trabajando con funciones que no tienen una API basada en data frames (esto es, pasas vectores individuales, no un data frame o expresiones que serán evaluadas en el contexto de un data frame), puedes encontrar `%$%` útil. Este operador "explota" las variables en un dataframe para que te puedas referir a ellas de manera explícita. Esto es útil cuando se trabaja con muchas funciones en R base:

```
mtautos %$%
  cor(cilindrada, millas)
#> [1] -0.8475514
```

- Para asignaciones, magrittr provee el operador `%<>%` que te permite reemplazar el código de la siguiente forma:

```
mtautos <- mtautos %>%
  transform(cilindros = cilindros * 2)
```

con

```
mtautos %<>% transform(cilindros = cilindros * 2)
```

No somos partidarios de este operador porque creemos que una asignación es una operación especial que siempre debe ser clara cuando sucede. En nuestra opinión, un poco de duplicación (esto es, repetir el nombre de un objeto dos veces) está bien para lograr hacer la asignación más explícita.

« 17 Introducción

19 Funciones »

19 Funciones

19.1 Introducción

Una de las mejores maneras de lograr tener mayor alcance haciendo ciencia de datos es escribir funciones. Las funciones te permitirán automatizar algunas tareas comunes de una forma más poderosa y general que copiar-y-pegar. Escribir funciones tiene tres grandes ventajas sobre copiar-y-pegar:

1. Puedes dar a la función un nombre evocador que hará tu código más fácil de entender.
2. A medida que cambien los requerimientos, solo necesitarás cambiar tu código en un solo lugar, en vez de en varios lugares.
3. Eliminas las probabilidades de errores accidentales cuando copias y pegas (por ej., al actualizar el nombre de una variable en un lugar, pero no en otro).

Escribir funciones es un viaje de toda una vida. Incluso después de usar R por varios años, seguimos aprendiendo nuevas técnicas y mejores formas de abordar viejos problemas. El objetivo de este capítulo no es enseñarte cada detalle esotérico de las funciones, sino introducirte en este tema con consejos pragmáticos que puedas aplicar inmediatamente.

Además de consejos prácticos para escribir funciones, este capítulo también te entregará consejos de estilo para tu código. Escribir código con buen estilo es como utilizar la puntuación correcta. Puedesmanejartesinella, pero utilizarla hace las cosas más fáciles de leer. Al igual que con los estilos de puntuación, hay muchas posibles variaciones. Si bien aquí presentamos el estilo que nosotros usamos en nuestro código, lo más importante es que seas consistente.

19.1.1 Prerrequisitos

El foco de este capítulo es escribir funciones en R base, por lo que no necesitarás ningún paquete extra.

19.2 ¿Cuándo deberías escribir una función?

Deberías considerar escribir una función cuando has copiado y pegado un bloque de código más de dos veces (es decir, ahora tienes tres copias del mismo). Mira, por ejemplo, el siguiente código. ¿Qué es lo que hace?

```
df <- tibble::tibble(
 a = rnorm(10),
 b = rnorm(10),
 c = rnorm(10),
 d = rnorm(10)
)

df$a <- (df$a - min(df$a, na.rm = TRUE)) /
 (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$b <- (df$b - min(df$b, na.rm = TRUE)) /
 (max(df$b, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$c <- (df$c - min(df$c, na.rm = TRUE)) /
 (max(df$c, na.rm = TRUE) - min(df$c, na.rm = TRUE))
df$d <- (df$d - min(df$d, na.rm = TRUE)) /
 (max(df$d, na.rm = TRUE) - min(df$d, na.rm = TRUE))
```

Es posible que hayas podido descifrar que lo que hace es reescalar cada columna para que tenga un rango de 0 a 1. Pero ¿has encontrado el error? Ocurrió copiando-y-pegando el código para `df$b`: Hemos olvidado de cambiar `a` a `b`. Extraer el código repetido en una función es una buena idea, ya que previene que cometas errores como este.

Para escribir una función, lo primero que necesitas hacer es analizar el código. ¿Cuantos inputs tiene?

```
(df$a - min(df$a, na.rm = TRUE)) /
 (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
```

Copy

Este código tiene un solo input `df$a`. (Si te sorprende que `TRUE` no es un input, puedes explorar el ejercicio de abajo). Para hacer los inputs más claros, es buena idea reescribir el código usando variables temporales con nombres generales. Acá el código requiere un solo vector numérico, por lo que lo llamaremos `x`:

```
x <- df$a
(x - min(x, na.rm = TRUE)) / (max(x, na.rm = TRUE) - min(x, na.rm = TRUE))
#>  [1] 0.2892677 0.7509271 0.0000000 0.6781686 0.8530656 1.0000000 0.1716402
#>  [8] 0.6107464 0.6116181 0.6008793
```

Copy

Hay algo de duplicación en este código. Estamos computando el rango de datos tres veces, así que tiene sentido hacerlo en un solo paso:

```
rng <- range(x, na.rm = TRUE)
(x - rng[1]) / (rng[2] - rng[1])
#>  [1] 0.2892677 0.7509271 0.0000000 0.6781686 0.8530656 1.0000000 0.1716402
#>  [8] 0.6107464 0.6116181 0.6008793
```

Copy

Sacar cálculos intermedios en variables nombradas es una buena práctica porque deja más claro qué es lo que está haciendo el código. Ahora que hemos simplificado el código y chequeado de que aún funciona, podemos convertirlo en una función:

```
rescale01 <- function(x) {
 rng <- range(x, na.rm = TRUE)
 (x - rng[1]) / (rng[2] - rng[1])
}
rescale01(c(0, 5, 10))
#> [1] 0.0 0.5 1.0
```

Copy

Hay tres pasos claves para crear una función nueva:

1. Necesitas elegir un **nombre** para la función. Aquí hemos usado `rescale01`, ya que esta función reescala (*rescale*, en inglés) un vector para que se ubique entre 0 y 1.

2. Listar los inputs, o **argumentos**, de la función dentro de `function`. Aquí solo tenemos un argumento. Si tenemos más, la llamada se vería como `function(x, y, z)`.

3. Situar el código que has creado en el **cuerpo** de una función, un bloque de `{` que sigue inmediatamente a `function(...)`.

Ten en cuenta el proceso general: solo hemos creado la función después de darnos cuenta cómo funciona con un input simple. Es más fácil empezar con código que funciona y luego convertirlo en una función; es más difícil crear la función y luego tratar que funcione.

En este punto es una buena idea chequear tu función con algunos inputs diferentes:

```
rescale01(c(-10, 0, 10))
#> [1] 0.0 0.5 1.0
rescale01(c(1, 2, 3, NA, 5))
#> [1] 0.00 0.25 0.50   NA 1.00
```

Copy

A medida que escribas más y más funciones eventualmente querrás convertir estos tests interactivos informales en tests formales y automatizados. Este proceso se llama *pruebas unitarias* (*unit testing*). Desafortunadamente, este tema está más allá del alcance de este libro, pero puedes aprender sobre él en https://r-pkgs.org/tests.html.

Podemos simplificar el ejemplo original ahora que tenemos una función:

```
df$a <- rescale01(df$a)
df$b <- rescale01(df$b)
df$c <- rescale01(df$c)
df$d <- rescale01(df$d)
```

Comparado al original, este código es fácil de entender y hemos eliminado erorres del tipo copiar-y-pegar. Existe aún un poco de duplicación, ya que estamos relizando lo mismo en diferentes columnas. Aprenderemos cómo eliminar esta duplicación en el capítulo sobre iteración, una vez que hayas aprendido más sobre las estructuras de R en el capítulo sobre vectores.

Otra ventaja de las funcioens es que si nuestros requerimientos cambian, solo necesitamos hacer modificaciones en un solo lugar. Por ejemplo, podríamos descubrir que algunas de nuestras variables incluyen valores infinitos, lo que hará que `rescale01()` falle:

```
x <- c(1:10, Inf)
rescale01(x)
#>  [1]   0   0   0   0   0   0   0   0   0   0 NaN
```

Debido a que hemos extraído el código en una función, solo necesitamos corregirlo en un lugar:

```
rescale01 <- function(x) {
 rng <- range(x, na.rm = TRUE, finite = TRUE)
 (x - rng[1]) / (rng[2] - rng[1])
}
rescale01(x)
#>  [1] 0.0000000 0.1111111 0.2222222 0.3333333 0.4444444 0.5555556 0.6666667
#>  [8] 0.7777778 0.8888889 1.0000000       Inf
```

Esta es una importante parte del principio de “no repetirse a uno mismo” (conocido como DRY: del inglés “**D**o not **R**epeat **Y**ourself”). Cuanta más repetición tengas en tu código, más lugares tendrás que recordar de actualizar cuando las cosas cambien (¡y eso siempre sucede!), y es más probable que crees errores (*bugs*) a lo largo del tiempo.

19.2.1 Ejercicios

1. ¿Por qué `TRUE` no es un parámetro para `rescale01()` ? ¿Qué pasaría si `x` está contenido en un valor único perdido y `na.rm` fuese `FALSE` ?

2. En la segunda variante de `rescale01()` , los valores infinitos se dejan sin cambio. Reescribe `rescale01()` para que `-Inf` sea convertido a 0, e `Inf` a 1.

3. Practica convertir los siguientes fragmentos de código en funciones. Piensa en lo que hace cada función. ¿Cómo la llamarías? ¿Cuántos argumentos necesita? ¿Puedes reescribirla para ser más expresiva o con menos duplicación de código?

   ```r
   mean(is.na(x))

   x / sum(x, na.rm = TRUE)

   sd(x, na.rm = TRUE) / mean(x, na.rm = TRUE)
   ```

4. Escribe tus propias funciones para computar la varianza y la inclinación de un vector numérico. La varianza se define como

$$\mathrm{Var}(x) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2,$$

   donde $\bar{x} = (\sum_i^n x_i)/n$ es la media de la muestra. La inclinación se define como

$$\mathrm{Skew}(x) = \frac{\frac{1}{n-2}\left(\sum_{i=1}^{n}(x_i - \bar{x})^3\right)}{\mathrm{Var}(x)^{3/2}}.$$

5. Escribe `both_na()` (*ambos_na()*), una función que toma dos vectores de la misma longitud y retorna el número de posiciones que tienen `NA` en ambos vectores.

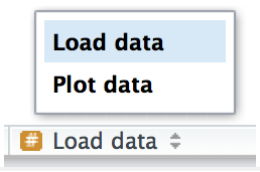
6. ¿Qué hacen las siguientes funciones? ¿Por qué son tan útiles pese a ser tan cortas?

```
Copy
```r
is_directory <- function(x) file.info(x)$isdir
is_readable <- function(x) file.access(x, 4) == 0
```
```

7. Lee la letra completa de “Pequeño Conejito Foo Foo”. Como ves, hay mucha duplicación en la letra de la canción. Extiende el ejemplo inicial de pipes para recrear la canción completa usando funciones para reducir la duplicación.

19.3 Las funciones son para los seres humanos y para las computadoras

Es importante recordar que las funciones no son solo para las computadoras, sino también para los seres humanos. A R no le importa el nombre de tu función ni los comentarios que tiene, pero estos sí serán importantes para los seres humanos que la lean. En esta sección se discutirán algunas cosas que debes tener en mente a la hora de escribir funciones entendibles para otras personas.

El nombre de una función es importante. Idealmente, debería ser corto, pero que evoque claramente lo que la función hace. ¡Eso es difícil! Es mejor que sea claro a que sea corto, considerando que la función de autocompletar de RStudio hace más fácil tipear nombres largos.

Generalmente, los nombres de las funciones deberían ser verbos y los argumentos sustantivos. Hay algunas excepciones: usar un sustantivo está bien si la función computa el valor de un sustantivo muy conocido (por ejemplo, `mean()` — (del inglés *media*) es mejor que `compute_mean()` — (del inglés *computar media*)), o accede a alguna propiedad del objeto (por ejemplo, `coef()` — (abreviatura en inglés de *coeficientes*) es mejor que `get_coefficients()` — (en inglés, *obtener coeficientes*)). Una buena señal de que un sustantivo puede ser una mejor elección es analizar si estás usando un verbo muy amplio como “obtener”, “computar”, “calcular” o “determinar”. Utiliza tu criterio y no tengas miedo de renombrar tu función si encuentras un nombre mejor más tarde.

```
Copy
# Muy corto
f()

# No es un verbo y es poco descriptivo
my_awesome_function()

# Largos, pero descriptivos
imputar_faltantes()
colapsar_anios()
```

Si el nombre de tu función está compuesto por múltiples palabras, te recomendamos usar el formato *serpiente*, o “snake_case”, en el que cada palabra en minúscula está separada por un guión bajo. Otra alternativa popular es es camelCase, o formato camello. No importa realmente cuál elijas, lo importante es que seas consistente: elije uno o el otro y quédate con él. R mismo no es muy consistente, pero no hay nada que puedas hacer al respecto. Asegúrate de no caer en la misma trampa haciendo tu código lo más consistente posible.

```
Copy
# ¡Nunca hagas esto!
col_mins <- function(x, y) {}
rowMaxes <- function(y, x) {}
```

Si tienes una familia de funciones que hacen cosas similares, asegúrate de que tengan nombres y argumentos consistentes. Utiliza un prefijo común para indicar que están conectadas. Eso es mejor que usar un sufijo común, ya que el autocompletado te permite escribir el prefijo y ver todos los otros miembros de la familia.

```
# Bien
input_select()
input_checkbox()
input_text()

# No tan bien
select_input()
checkbox_input()
text_input()
```

Un buen ejemplo de este diseño es el paquete **stringr**: si no recuerdas exactamente qué función necesitas, puedes escribir `str_` y el autocompletado te ayudará a refrescar tu memoria. Siempre que sea posible, evita sobrescribir funciones y variables ya existentes. No siempre es posible hacer esto, ya que hay un montón de nombres buenos que ya han sido utilizados por otros paquetes. De todas maneras, evitar el uso de los nombres más comunes de R base ahorrará confusiones.

```
# ¡No hagas esto!
T <- FALSE
c <- 10
mean <- function(x) sum(x)
```

Usa comentarios, esto es, líneas que comienzan con `#`, para explicar el "porqué" de tu código. En general deberías evitar comentarios que expliquen el "qué" y el "cómo". Si no se entiende qué es lo que hace el código leyéndolo, deberías pensar cómo reescribirlo de manera que sea más claro. ¿Necesitas agregar algunas variables intermedias con nombres útiles? ¿Deberías dividir una función larga en subcomponentes para que pueda ser nombrada? Sin embargo, tu código nunca podrá capturar la razón detrás de tus decisiones: ¿Por qué elegiste este enfoque frente a otras alternativas? ¿Qué otra cosa probaste que no funcionó? Es una gran idea capturar este tipo de pensamientos en un comentario.

Otro uso importante de los comentarios es para dividir tu archivo en partes, de modo que resulte más fácil de leer. Utiliza líneas largas de `-` y `=` para que resulte más fácil detectar los fragmentos.

```
# Cargar los datos -------------------------------------

# Graficar los datos -----------------------------------
```

RStudio proporciona un método abreviado de teclado para crear estos encabezados (Cmd/Ctrl + Shift + R), y los mostrará en el menú desplegable de navegación de código en la parte inferior izquierda del editor:



19.3.1 Ejercicios

1. Lee el código fuente para cada una de las siguientes tres funciones, interpreta qué hacen y luego propone nombres mejores.

```
```r
f1 <- function(string, prefix) {
substr(string, 1, nchar(prefix)) == prefix
}
f2 <- function(x) {
if (length(x) <= 1) return(NULL)
x[-length(x)]
}
f3 <- function(x, y) {
rep(y, length.out = length(x))
}
```
```

2. Toma una función que hayas escrito recientemente y tómate 5 minutos para pensar un mejor nombre para la función y para sus argumentos.

3. Compara y contrasta `rnorm()` y `MASS::mvrnorm()`. ¿Cómo podrías hacerlas más consistentes?

4. Argumenta por qué `norm_r()`, `norm_d()`, etc. sería una mejor opción que `rnorm()`, `dnorm()`. Argumenta lo contrario.

19.4 Ejecución condicional

Una sentencia `if` (*si*) te permite ejecutar un código condicional. Por ejemplo:

```
if (condition) {
 # el código que se ejecuta cuando la condición es verdadera (TRUE)
} else {
 # el código que se ejecuta cuando la condición es falsa (FALSE)
}
```

Copy

Para obtener ayuda acerca de `if` necesitas ponerlo entre acentos graves: `` ?`if` ``. La ayuda no es especialmente útil si aún no tienens tanta experiencia programando. ¡Pero al menos puedes saber cómo llegar a ella!

Aquí se presenta una función simple que utiliza una sentencia `if`. El objetivo de esta función es devolver un vector lógico que describa si cada elemento de un vector tiene nombre (*name*).

```
tiene_nombre <- function(x) {
 nms <- names(x)
 if (is.null(nms)) {
 rep(FALSE, length(x))
 } else {
 !is.na(nms) & nms != ""
 }
}
```

Copy

Esta función aprovecha la regla de retorno estándar: una función devuelve el último valor que calculó. Este es uno de los dos usos de la declaración `if`.

19.4.1 Condiciones

La condición debe evaluar como `TRUE` o `FALSE`. Si es un vector, recibirás un mensaje de advertencia; si es una `NA`, obtendrás un error. Ten cuidado con estos mensajes en tu propio código:

```
if (c(TRUE, FALSE)) {}
#> Warning in if (c(TRUE, FALSE)) {: the condition has length > 1 and only the
#> first element will be used
#> NULL

if (NA) {}
#> Error in if (NA) {: missing value where TRUE/FALSE needed
```

Copy

Puedes usar `||` (o) y `&&` (y) para combinar múltiples expresiones lógicas. Estos operadores hacen "cortocircuito": tan pronto como `||` vea el primer `TRUE` devolverá `TRUE` sin calcular nada más. Tan pronto como && vea el primer `FALSE`, devolverá `FALSE`. Nunca debes usar `|` o `&` en una sentencia `if`: estas son operaciones vectorizadas que se aplican a valores múltiples (es por eso que las usas en `filter()`). Si tienes un vector lógico, puedes utilizar `any()` (*cualquier*) o `all()` (*todo*) para juntarlo en un único valor.

Ten cuidado al comprobar igualdad. `==` está vectorizado, lo que significa que es fácil obtener más de un output. Comprueba si la longitud ya es 1 y junta con `all()` o `any()`, o usa la función no vectorizada `identical()` (*indéntico*). `identical()` es una función muy estricta: siempre devuelve un solo `TRUE` o un solo `FALSE`, y no fuerza tipos de estructuras de datos. Esto significa que debes tener cuidado al comparar enteros y dobles:

```
identical(0L, 0)
#> [1] FALSE
```

Copy

También hay que tener cuidado con los números de punto flotante:

```
x <- sqrt(2) ^ 2
x
#> [1] 2
x == 2
#> [1] FALSE
x - 2
#> [1] 4.440892e-16
```

Copy

En su lugar, utiliza `dplyr::near()` para comparaciones, como se describe en la sección sobre comparaciones.

Y recuerda, ¡ `x == NA` no hace nada útil!

19.4.2 **Condiciones múltiples**

Puedes encadenar múltiples sentencias *if* juntas:

```
if (this) {
 # haz aquello
} else if (that) {
 # haz otra cosa
} else {
 #
}
```

Copy

Pero si terminas con una larga serie de sentencias `if` encadenadas, deberías considerar reescribir el código. Una técnica útil es la función `switch()` . Esta te permite evaluar el código seleccionado según la posición o el nombre.

```
#> function(x, y, op) {
#>  switch(op,
#>  plus = x + y,
#>  minus = x - y,
#>  times = x * y,
#>  divide = x / y,
#>  stop("¡operación desconocida!")
#>  )
#> }
```

Copy

Otra función útil que a menudo puede eliminar largas cadenas de sentencias `if` es `cut()` (*cortar*). Esta es utilizada para convertir en categóricas variables que son continuas.

19.4.3 **Estilo del código**

Tanto `if` como `function` deberían ir (casi) siempre entre llaves (`{}`) y el contenido debería tener una sangría de dos espacios. Esto hace que sea más fácil distinguir la jerarquía dentro de tu código al mirar el margen izquierdo.

La llave de apertura nunca debe ir en su propia línea y siempre debe ir seguida de una línea nueva. Una llave de cierre siempre debe ir en su propia línea, a menos que sea seguida por `else` . Siempre ponle sangría al código que va dentro de las llaves.

```
# Bien
if (y < 0 && debug) {
 message("Y es negativo")
}

if (y == 0) {
 log(x)
} else {
 y ^ x
}

# Mal
if (y < 0 && debug)
message("Y is negative")

if (y == 0) {
 log(x)
}
else {
 y ^ x
}
```

Está bien evitar las llaves si tienes una sentencia `if` muy corta que cabe en una sola línea:

```
y <- 10
x <- if (y < 20) "Too low" else "Too high"
```

Esto se recomienda solo para sentencias `if` muy breves. De lo contrario, la sentencia completa es más fácil de leer:

```
if (y < 20) {
 x <- "Muy bajo"
} else {
 x <- "Muy alto"
}
```

19.4.4 Ejercicios

1. ¿Cuál es la diferencia entre `if` e `ifelse()`? Lee cuidadosamente la ayuda y construye tres ejemplos que ilustren las diferencias clave.

2. Escribe una función de saludo que diga “buenos días”, “buenas tardes” o “buenas noches”, según la hora del día. (Sugerencia: usa un argumento de tiempo que por defecto sea `lubridate::now()`; eso hará que sea más fácil testear tu función).

3. Implementa una función `fizzbuzz` que tenga un solo número como input. Si el número es divisible por tres, devuelve “fizz”. Si es divisible por cinco, devuelve “buzz”. Si es divisible por tres y cinco, devuelve “fizzbuzz”. De lo contrario, devuelve el número. Asegúrate de escribir primero código que funcione antes de crear la función.

4. ¿Cómo podrías usar `cut()` (*cortar()*) para simplificar este conjunto de sentencias if-else anidadas?

```
if (temp <= 0) {
"congelado"
} else if (temp <= 10) {
"helado"
} else if (temp <= 20) {
"fresco"
} else if (temp <= 30) {
"tibio"
} else {
"caluroso"
}
```

```
    ¿Cómo cambiarías la llamada a `cut()` si hubieras usado `<`en lugar de `<=`? ¿Cuál es la ventaja principal de `cut()` para este problema? (Pista: ¿qué sucede si tienes muchos valores en `temp`?)
```

Copy

5. ¿Qué sucede si usas `switch()` con un valor numérico?

6. ¿Qué hace la llamada a `switch()` ? ¿Qué sucede si `x` fuera “e”?

Copy

```
 ```r
switch(x,
a = ,
b = "ab",
c = ,
d = "cd"
)
```

Experimenta, luego lee cuidadosamente la documentación.
```

19.5 Argumentos de funciones

Los argumentos de las funciones normalmente están dentro de dos conjuntos amplios: un conjunto provee los **datos** a computar y el otro los argumentos que controlan los **detalles** de la computación. Por ejemplo:

- En `log()` , los datos son `x` , y los detalles son la `base` del algoritmo.
- En `mean()` , los datos son `x` , y los detalles son la cantidad de datos para recortar de los extremos (`trim`) y cómo lidiar con los valores faltantes (`na.rm`).
- En `t.test()` , los datos son `x` e `y` , y los detalles del test son `alternative` , `mu` , `paired` , `var.equal` , y `conf.level` .
- En `str_c()` puedes suministrar cualquier número de caracteres a `...` , y los detalles de la concatenación son contralos por `sep` y `collapse` .

Generalmente, argumentos relativos a los datos deben ir primero. El detalle de los mismos podría estar al final y con valores por defecto. Se especifica un valor por defecto de la misma manera en la que se llama a una función con un argumento nombrado:

Copy

```
# Computar intervalo de confianza alrededor de la media usando la aproximación normal
mean_ci <- function(x, conf = 0.95) {
 se <- sd(x) / sqrt(length(x))
 alpha <- 1 - conf
 mean(x) + se * qnorm(c(alpha / 2, 1 - alpha / 2))
}

x <- runif(100)
mean_ci(x)
#> [1] 0.4976111 0.6099594
mean_ci(x, conf = 0.99)
#> [1] 0.4799599 0.6276105
```

El valor por defecto debería ser casi siempre el valor más común. Las pocas excepciones que existen a esta regla deben realizarse con cuidado. Por ejemplo, tiene sentido que `na.rm` por defecto sea `FALSE` porque los valores faltantes son importantes. Aunque `na.rm = TRUE` es lo que usualmente pones en tu codigo, es una mala idea que el comportamiento por defecto sea ignorar silenciosamente los valores faltantes.

Cuando llamas una función, generalmente omites los nombres de los argumentos de datos justamente porque son los más comúnmente usados. Si quieres usar un valor distinto al por defecto en de un argumento de detalle, debes usar el nombre completo:

```
# Bien
mean(1:10, na.rm = TRUE)

# Mal
mean(x = 1:10, , FALSE)
mean(, TRUE, x = c(1:10, NA))
```

Puedes referirte a un argumento por su prefijo único (ej. `mean(x, n = TRUE)`), pero generalmente es mejor evitarlo dadas las posibilidades de confusión.

Ten en cuenta que cuando llamas a una función, debes colocar un espacio alrededor de `=` y siempre poner un espacio después de la coma, no antes (como cuando escribes en español). El uso del espacio en blanco hace más fácil echar un vistazo a la función para identificar los componentes importantes.

```
# Bien
promedio <- mean(pies / 12 + pulgadas, na.rm = TRUE)

# Mal
promedio<-mean(pies/12+pulgadas,na.rm=TRUE)
```

19.5.1 **Elección de nombres**

Los nombres de los argumentos también son importantes. A R no le importa, pero sí a quienes leen tu código (¡incluyéndo a tu futuro-yo!). En general, deberías preferir nombres largos y más descriptivos, aunque hay un puñado de nombres muy comunes y muy cortos. Vale la pena memorizar estos:

- `x`, `y`, `z`: vectores.
- `w`: un vector de pesos.
- `df`: un data frame.
- `i`, `j`: índices numéricos (usualmente filas y columnas).
- `n`: longitud, o número de filas.
- `p`: número de columnas.

En caso contrario, deberías considerar hacerlos coincidir con nombres de argumentos de funciones de R que ya existen. Por ejemplo, usa `na.rm` para determinar si los valores faltantes deberían ser eliminados.

19.5.2 **Chequear valores**

A medida que vayas escribiendo más funciones, eventualmente llegarás al punto en el que no recordarás cómo opera una determinada función. En este punto es común que llames a la función con inputs inválidos. Para evitar este problema, a menudo es útil hacer las restricciones explícitas. Por ejemplo, imagina que has escrito algunas funciones para calcular estadísticos de resumen ponderados:

```
wt_mean <- function(x, w) {
 sum(x * w) / sum(w)
}
wt_var <- function(x, w) {
 mu <- wt_mean(x, w)
 sum(w * (x - mu) ^ 2) / sum(w)
}
wt_sd <- function(x, w) {
 sqrt(wt_var(x, w))
}
```

¿Qué pasa si `x` y `w` no son de la misma longitud?

```
wt_mean(1:6, 1:3)
#> [1] 7.666667
```

En este caso, debido a las reglas de reciclado de vectores de R, no obtenemos un error.

Es una buena práctica verificar las condiciones previas importantes y arrojar un error (con `stop()`, *parar*), si estas no son verdaderas:

```
wt_mean <- function(x, w) {
 if (length(x) != length(w)) {
 stop("`x` y `w` deben tener la misma extensión", call. = FALSE)
 }
 sum(w * x) / sum(w)
}
```

Ten cuidado de no llevar esto demasiado lejos. Debe haber un equilibrio entre la cantidad de tiempo que inviertes en hacer que tu función sea sólida y la cantidad de tiempo que pasas escribiéndola. Por ejemplo, si además agregas a la función un argumento `na.rm`, probablemente no lo verificaste con cuidado:

```
wt_mean <- function(x, w, na.rm = FALSE) {
 if (!is.logical(na.rm)) {
 stop("`na.rm` debe ser lógico")
 }
 if (length(na.rm) != 1) {
 stop("`na.rm` debe tener extensión 1")
 }
 if (length(x) != length(w)) {
 stop("`x` y `w` deben tener la misma extensión", call. = FALSE)
 }

 if (na.rm) {
 miss <- is.na(x) | is.na(w)
 x <- x[!miss]
 w <- w[!miss]
 }
 sum(w * x) / sum(w)
}
```

Esto es mucho trabajo con poca ganancia adicional. Una opción útil es incorporar `stopifnot()`: esto comprueba que cada argumento sea `TRUE`. En caso contrario genera un mensaje de error.

```
wt_mean <- function(x, w, na.rm = FALSE) {
 stopifnot(is.logical(na.rm), length(na.rm) == 1)
 stopifnot(length(x) == length(w))

 if (na.rm) {
 miss <- is.na(x) | is.na(w)
 x <- x[!miss]
 w <- w[!miss]
 }
 sum(w * x) / sum(w)
}
wt_mean(1:6, 6:1, na.rm = "foo")
#> Error in wt_mean(1:6, 6:1, na.rm = "foo"): is.logical(na.rm) is not TRUE
```

Ten en cuenta que al usar `stopifnot()` afirmas lo que debería ser cierto en lugar de verificar lo que podría estar mal.

19.5.3 **Punto-punto-punto (...)**

Muchas funciones en R tienen un número arbitrario de inputs:

```
sum(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
#> [1] 55
stringr::str_c("a", "b", "c", "d", "e", "f")
#> [1] "abcdef"
```

¿Cómo operan estas funciones? Estas se sostienen en un argumento especial: `...` (llamado punto-punto-punto). Este argumento especial captura cualquier número de argumentos que no estén contemplados de otra forma.

Es práctico porque puedes enviar estos `...` a otra función. Este es un argumento multipropósito útil si tu función principalmente envuelve (*wraps*) a otra función. Por ejemplo, usualmente creamos estas funciones de ayuda alrededor de `str_c()`:

```
commas <- function(...) stringr::str_c(..., collapse = ", ")
commas(letters[1:10])
#> [1] "a, b, c, d, e, f, g, h, i, j"

rule <- function(..., pad = "-") {
 title <- paste0(...)
 width <- getOption("width") - nchar(title) - 5
 cat(title, " ", stringr::str_dup(pad, width), "\n", sep = "")
}
rule("Important output")
#> Important output ----------------------------------------------------------
```

Copy

Aquí `...` nos permite enviar cualquier argumento con el que no queramos lidiar hacia `str_c()`. Esto es muy conveniente, pero tiene un costo asociado: cualquier argumento mal escrito no generará un error. Esto hace que sea más fácil que los errores de tipeo pasen inadvertidos:

```
x <- c(1, 2)
sum(x, na.mr = TRUE)
#> [1] 4
```

Copy

Si solo quieres capturar los valores de `...`, entonces utiliza `list(...)`.

19.5.4 Evaluación diferida

Los argumentos en R se evalúan de forma "perezosa": no se computan hasta que se los necesita. Esto significa que si nunca se los usa, nunca son llamados. Esta es una propiedad importante de R como lenguaje de programación, pero generalmente no es fundamental cuando escribes tus propias funciones para el análisis de datos. Puedes leer más acerca de la evaluación diferida en https://adv-r.hadley.nz/functions.html#lazy-evaluation.

19.5.5 Ejercicios

1. ¿Qué realiza `commas(letters, collapse = "-")`? ¿Por qué?

2. Sería bueno si se pudiera suministrar múltiples caracteres al argumento `pad`, por ejemplo, `rule("Title", pad = "-+")`. ¿Por qué esto actualmente no funciona? ¿Cómo podrías solucionarlo?

3. ¿Qué realiza el argumento `trim` a la función `mean()`? ¿Cuándo podrías utilizarlo?

4. El valor de defecto del argumento `method` para `cor()` es `c("pearson", "kendall", "spearman")`. ¿Qué significa esto? ¿Qué valor se utiliza por defecto?

19.6 Valores de retorno

Darse cuenta qué es lo que tu función debería devolver suele ser bastante directo: ¡es el porqué de crear la función en primer lugar! Hay dos cosas que debes considerar al retornar un valor:

1. ¿Devolver un valor antes hace que tu función sea más fácil de leer?

2. ¿Puedes hacer tu función apta para utilizarla con pipes (`%>%`)?

19.6.1 Sentencias de retorno explícitas

El valor devuelto por una función suele ser la última sentencia que esta evalúa; sin embargo, puedes optar por devolver algo anticipadamente haciendo uso de la función `return()` (*retornar* o *devolver* en inglés). Creemos que es mejor reservar el uso de la función `return()` para los casos en los que es posible devolver anticipadamente una solución más simple. Una razón común para hacer esto es que los argumentos estén vacíos:

```
complicated_function <- function(x, y, z) {
 if (length(x) == 0 || length(y) == 0) {
 return(0)
 }
 # Código complicado aquí
}
```

Otra razón puede ser porque tienes una sentencia `if` con un bloque complicado y uno sencillo. Por ejemplo, podrías escribir una sentencia *if* de esta manera:

```
f <- function() {
 if (x) {
 # Haz
 # algo
 # que
 # tome
 # muchas
 # líneas
 # para
 # ser
 # expresado
 } else {
 # retorna algo corto
 }
}
```

Si el primer bloque es muy largo, para cuando lleges al `else` ya te habrás olvidado de la condición. Una forma de reescribir esto es usar un retorno anticipado para el caso sencillo:

```
f <- function() {
 if (!x) {
 return(algo_corto)
 }

 # Haz
 # algo
 # que
 # tome
 # muchas
 # líneas
 # para
 # ser
 # expresado
}
```

Esto tiende a hacer el código más fácil de entender, ya que no necesitas tanto contexto para interpretarlo.

19.6.2 **Escribir funciones aptas para un pipe**

Si quieres escribir funciones que sean aptas para usarlas con un pipe (`%>%`), es importante que pienses en los valores de retorno. Conocer el tipo de objeto de tu valor de retorno significará que tu secuencia de pipes “simplemente funcionará”. Por ejemplo, en **dplyr** y **tidyr** el tipo de objeto es un data frame.

Hay dos tipos básicos de funciones aptas para pipes: transformaciones y efectos secundarios. En las **transformaciones**, se ingresa un objeto como primer argumento y se retorna una versión modificada del mismo. En el caso de los **efectos_secundarios**, el objeto ingresado no es modificado, sino que la función realiza una acción sobre el objeto (como dibujar un gráfico o guardar un archivo). Las funciones de efectos secundarios deben retornar “invisiblemente” el primer argumento, de manera que aún cuando no se impriman, puedan ser utilizados en una secuencia de pipes. Por ejemplo, esta función imprime el número de valores faltantes en un data frame:

```
mostrar_faltantes <- function(df) {
 n <- sum(is.na(df))
 cat("Valores faltantes: ", n, "\n", sep = "")

 invisible(df)
}
```

Si la llamamos de manera interactiva, `invisible()` implica que el `df` input no se imprime:

```
mostrar_faltantes(mtautos)
#> Valores faltantes: 0
```

Pero sigue estando ahí, solamente que no se imprime por defecto:

```
x <- mostrar_faltantes(mtautos)
#> Valores faltantes: 0
class(x)
#> [1] "data.frame"
dim(x)
#> [1] 32 11
```

Y todavía podemos usarlo en un pipe:

```
mtautos %>%
 mostrar_faltantes() %>%
 mutate(millas = ifelse(millas < 20, NA, millas)) %>%
 mostrar_faltantes()
#> Valores faltantes: 0
#> Valores faltantes: 18
```

19.7 Entorno

El último componente de una función es su entorno. Esto no es algo que debas entender con profundidad cuando recién empiezas a escribir funciones. Sin embargo, es importante saber un poco acerca de los entornos, ya que son cruciales para que algunas funciones trabajen. El entorno de una función controla cómo R encuentra el valor asociado a un nombre. Por ejemplo, toma la siguiente función:

```
f <- function(x) {
 x + y
}
```

En muchos lenguajes de programación, esto sería un error, porque `y` no está definida dentro de la función. En R, esto es un código válido ya que R usa reglas llamadas de **ámbito léxico** (*lexical scoping*) para encontrar el valor asociado a un nombre. Como `y` no está definida dentro de la función, R mirará dentro del **entorno** donde la función fue definida:

```
y <- 100
f(10)
#> [1] 110

y <- 1000
f(10)
#> [1] 1010
```

Este comportamiento parece una receta para errores (*bugs*) y, de hecho, debes evitar crear deliberadamente funciones como esta. Sin embargo, en líneas generales no causa demasiados problemas (especialmente si reinicias regularmente R para hacer borrón y cuenta nueva). La ventaja de este comportamiento es que, desde el punto de vista del lenguaje, permite que R sea muy consistente. Cada nombre es buscado usando el mismo conjunto de reglas. Para `f()` esto incluye el comportamiento de dos cosas que podrías no esperar: `{` y `+`. Esto te permite hacer cosas enrevesadas como la siguiente:

```
`+` <- function(x, y) {
 if (runif(1) < 0.1) {
 sum(x, y)
 } else {
 sum(x, y) * 1.1
 }
}
table(replicate(1000, 1 + 2))
#>
#>   3 3.3
#> 100 900
rm(`+`)
```

Copy

Este es un fenómeno común en R. R pone pocos límites a tu poder. Puedes hacer muchas cosas que no podrías hacer en otro lenguaje de programación. Puedes hacer cosas que el 99% de las veces son extremadamente desaconsejables (¡como sobrescribir manualmente cómo funciona la adición!). Pero este poder y flexibilidad es lo que hace que herramientas como **ggplot2** y **dplyr** sean posibles. Aprender cómo hacer el mejor uso de esta flexibilidad está mas allá del alcance de este libro, pero puedes leer al respecto en *Advanced R*.

« 18 Pipes

20 Vectores »

20 Vectores

20.1 Introducción

Hasta ahora este libro se ha enfocado en los tibbles y los paquetes que trabajan con ellos. Pero a medida que empieces a escribir tus propias funciones y a profundizar en R, es necesario que aprendas sobre vectores, esto es, sobre los objetos que están a la base de los tibbles. Si aprendiste R de una manera más tradicional, probablemente ya te hayas familiarizado con los vectores, ya que la mayoría de los recursos sobre R parten con vectores y luego abordan los tibbles. Creemos que es mejor empezar con los tibbles porque resultan útiles de inmediato y luego explorar los componentes que están a la base. Los vectores son particularmente importantes, ya que la mayoría de las funciones que escribirás trabajan con ellos. Es posible escribir funciones que trabajen con tibbles (como **ggplot2**, **dplyr** y **tidyr**); sin embargo, las herramientas que necesitas para hacerlo aún son idiosincráticas y no están lo suficientemente maduras. Estamos trabajando en una mejor aproximación (https://github.com/hadley/lazyeval), pero no estará lista a tiempo para la publicación del libro. Incluso aún cuando esté completa, será necesaro que entiendas los vectores: hará que sea más fácil escribir una capa amigable con el usuario encima de ellos.

20.1.1 Prerrequisitos

Este capítulo se enfoca en las estructuras de datos de R base, por lo que no es esencial cargar ningún paquete. Sin embargo, usaremos un conjunto de funciones del paquete **purrr** para evitar algunas inconsistencias de R base.

```
library(tidyverse)
```

Copy

20.2 Vectores básicos

Hay dos tipos de vectores:

1. Vectores **atómicos**, de los cuales existen seis tipos: **lógico**, **entero**, **doble**, **caracter**, **complejo** y **sin procesar** (*raw*). Los vectores de tipo entero y doble son conocidos de manera colectiva como vectores numéricos.
2. Las **listas**, que a veces son denominadas como vectores recursivos debido a que pueden contener otras listas.

La diferencia principal entre vectores atómicos y listas es que los vectores atómicos son **homogéneos**, mientras las listas pueden ser **heterogéneas**. Existe otro objeto relacionado: `NULL` (nulo). `NULL` es a menudo utilizado para representar la ausencia de un vector (en oposición a `NA` que se utiliza para representar la ausencia de un valor en un vector). `NULL` se comporta típicamente como un vector de longitud 0 (cero). La figura 20.1 resume las interrelaciones.

On this page

View source
Edit this page

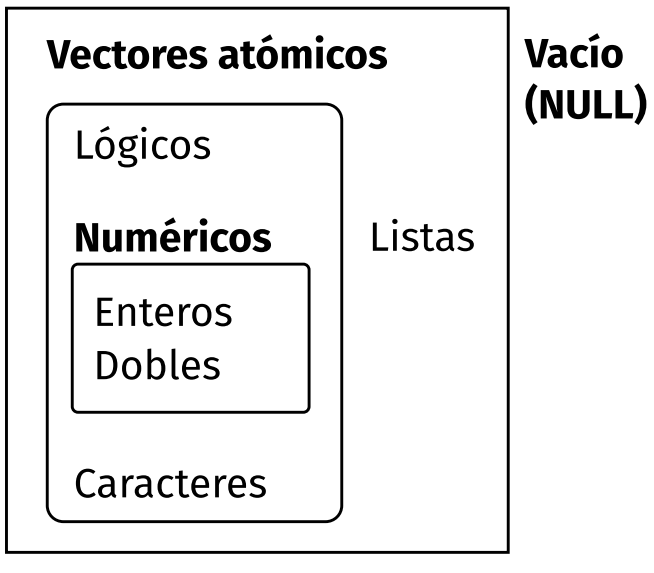


Figure 20.1: La jerarquía de los tipos de vectores en R

Cada vector tiene dos propiedades clave:

1. Su **tipo** (*type*), que puedes determinar con `typeof()` (*tipo de*).

```
typeof(letters)
#> [1] "character"
typeof(1:10)
#> [1] "integer"
```

1. Su **longitud** (*length*), que puede determinar con `length()`.

```
x <- list("a", "b", 1:10)
length(x)
#> [1] 3
```

Los vectores pueden contener también metadata adicional arbitraria en forma de atributos. Estos atributos son usados para crear **vectores aumentados**, los que se basan en un comportamiento distinto. Existen tres tipos de vectores aumentados: * Los factores (*factors*), construidos sobre la base de vectores de enteros. * Las fechas y fechas-hora (*date-times*), construidas a partir de vectores numéricos. * Los dataframes y tibbles, construidos a partir de listas.

Este capítulo te introducirá en estos importantes vectores, desde los más simples a los más complicados. Comenzarás con vectores atómicos, luego seguirás con listas y finalizarás con vectores aumentados.

20.3 Tipos importantes de vectores atómicos

Los cuatro tipos más importantes de vectores atómicos son lógico, entero, doble y carácter. Los de tipo *raw* y complejo son raramente usados durante el análisis de datos, por lo tanto, no discutiremos sobre ellos aquí.

20.3.1 Lógico

Los vectores de tipo lógico son el tipo más sencillo de vectores atómicos, ya que solo pueden tomar tres valores posibles: `FALSE`, `TRUE` y `NA`. Los vectores lógicos son construidos usualmente con operadores de comparación, tal como se describe en la sección comparaciones. También puedes crearlos manualmente con la función `c()`:

```
1:10 %% 3 == 0
#>  [1] FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE

c(TRUE, TRUE, FALSE, NA)
#> [1]  TRUE  TRUE FALSE    NA
```

20.3.2 Numérico

Los vectores de tipo entero y doble se conocen de manera colectiva como vectores numéricos. En R, los números por defecto son representados como *double*. Para generar un entero, coloca una L después del número:

```
typeof(1)
#> [1] "double"
typeof(1L)
#> [1] "integer"
1.5L
#> [1] 1.5
```

Copy

La distinción entre enteros y dobles usualmente no resulta importante, aunque existen dos diferencias relevantes de las que debes ser consciente:

1. Los números dobles son aproximaciones. Representan números de punto flotante que no siempre pueden ser representados de manera precisa con un cantidad fija de memoria. Esto significa que debes considerar a todos los dobles como aproximaciones. Por ejemplo, ¿cuál es el cuadrado de la raíz cuadrada de dos?

```
x <- sqrt(2) ^ 2
x
#> [1] 2
x - 2
#> [1] 4.440892e-16
```

Copy

Este comportamiento es común cuando trabajas con números de punto flotante: la mayoría de los cálculos incluyen algunos errores de aproximación. En lugar de comparar números de punto flotante usando `==`, debes usar ´dplyr::near()`, que provee tolerancia numérica.

2. Los números enteros tienen un valor especial, `NA`, mientras que los dobles tienen cuatro tipos: `NA`, `NaN`, `Inf` and `-Inf`. Los tres valores especiales `NaN`, `Inf` and `-Inf` pueden surgir durante una división:

```
c(-1, 0, 1) / 0
#> [1] -Inf  NaN  Inf
```

Copy

Evita usar `==` para chequear estos valores especiales. En su lugar usa la funciones de ayuda `is.finite()`, `is.infinite()`, y `is.nan()`:

	0	Inf	NA	NaN
`is.finite()`	x			
`is.infinite()`		x		
`is.na()`			x	x
`is.nan()`				x

20.3.3 **Caracter**

Los vectores de caracteres son los tipos de vectores atómicos más complejos, ya que cada elemento del mismo es un *string* (una cadena de caracteres) y un *string* puede contener una cantidad arbitraria de datos. Ya has aprendido un montón acerca de cómo trabajar con *strings* en el capítulo sobre Cadenas de caracteres. Pero queremos mencionar una característica importante de la implementación que subyace a los *strings*: R usa una reserva global de *strings*. Esto significa que cada string único solo es almacenado en la memoria una vez y cada uso de un *string* apunta a esa representación. Esto reduce la cantidad de memoria necesaria para *strings* duplicados. Puedes ver este comportamiento en práctica con `pryr::object_size()`:

```
x <- "Esta es una cadena de caracteres razonablemente larga."
pryr::object_size(x)
#> Registered S3 method overwritten by 'pryr':
#>   method      from
#>   print.bytes Rcpp
#> 168 B

y <- rep(x, 1000)
pryr::object_size(y)
#> 8.16 kB
```

`y` no utiliza 1000 veces más memoria que `x`, ya que cada elemento de `y` solo apunta al mismo *string*. Cada uno de estos *punteros* utiliza 8 bytes, por lo que 1000 punteros hacia un *string* de 168 B es igual a 8 * 1000 + 168 = 8.16 kB.

20.3.4 **Valores faltantes**

Cada tipo de vector atómico tiene su propio valor faltante (*missing value*):

```
NA # lógico
#> [1] NA
NA_integer_ # entero
#> [1] NA
NA_real_ # doble o real
#> [1] NA
NA_character_ # caracter
#> [1] NA
```

Normalmente no necesitas saber sobre los diferentes tipos porque siempre puedes usar `NA` (del inglés *Not Available*, *no disponible*), el que se convertirá al tipo correcto usando las reglas implícitas de coerción. Sin embargo, existen algunas funciones que son estrictas acerca de sus inputs, por lo que es útil tener presente este conocimiento, así tu código puede ser lo suficientemente específico cuando lo necesites.

20.3.5 **Ejercicios**

1. Describe la diferencia entre `is.finite(x)` y `!is.infinite(x)`.
2. Lee el código fuente de `dplyr:: near()` (Pista: para ver el código fuente, escribe el nombre de la función sin `()`). ¿Funcionó?
3. Un vector de tipo lógico puede tomar 3 valores posibles. ¿Cuántos valores posibles puede tomar un vector de tipo entero? ¿Cuántos valores posibles puede tomar un vector de tipo doble? Usa google para investigar sobre esto.
4. Idea al menos 4 funciones que te permitan convertir un vector de tipo doble a entero. ¿En qué difieren las funciones? Describe las diferencias con precisión.
5. ¿Qué funciones del paquete **readr** te permiten convertir una cadena de caracteres en un vector de tipo lógico, entero y doble?

20.4 **Usando vectores atómicos**

Ahora que conoces los diferentes tipos de vectores atómicos, es útil repasar algunas herramientas importantes para trabajar con ellos. Estas incluyen:

1. Cómo realizar una conversión de un determinado tipo a otro y en qué casos esto sucede automáticamente.
2. Cómo decidir si un objeto es de un tipo específico de vector.
3. Qué sucede cuando trabajas con vectores de diferentes longitudes.
4. Cómo nombrar los elementos de un vector
5. Cómo extraer los elementos de interés de un vector.

20.4.1 **Coerción**

Existen dos maneras de convertir, o coercionar, un tipo de vector a otro:

1. La coerción explícita ocurre cuando llamas una función como `as.logical()`, `as.integer()`, `as.double()`, o `as.character()`. Cuando te encuentres usando coerción expíicita, comprueba si es posible hacer algún tipo de arreglo antes que permita que al vector nunca se le llegue a asignar el tipo incorrecto. Por ejemplo, podrías necesitar ajustar la especificación de `col_types` (*tipos de columna*) del paquete **readr** cuando importas los datos.

2. La coerción implícita ocurre cuando usas un vector en un contexto específico en el que se espera que sea de cierto tipo. Por ejemplo, cuando usas un vector de tipo lógico con la función numérica `summary` (*resumen*), o cuando usas un vector de tipo doble donde se espera que sea de tipo entero. Debido a que la coerción explícita se usa raramente y es mucho más fácil de entender, nos enfocaremos acá en la coerción implícita. Ya viste el tipo más importante de coerción implícita: cuando se usa un vector de tipo lógico en un contexto numérico. En ese caso, el valor `TRUE` es convertido a `1` y `FALSE` a `0`. Esto significa que la suma de un vector de tipo lógico es el número de valores verdaderos, y que la media es de un vector lógico es la proporción de valores verdaderos:

```
x <- sample(20, 100, replace = TRUE)
y <- x > 10
sum(y) # ¿Cuántos valores son más grandes que 10?
#> [1] 38
mean(y) # ¿Qué proporción es mayor que 10?
#> [1] 0.38
```

Copy

Quizás veas código (usualmente más antiguo) que se basa en la coerción implícita en la dirección opuesta, es decir, de un valor entero a uno lógico:

```
if (length(x)) {
 # hacer algo
}
```

Copy

En este caso, `0` es convertido a `FALSE` y todo lo demás es convertido a `TRUE`. Creemos que esto hace que tu código sea más difícil de entender, por lo que no lo recomendamos. En su lugar, utiliza explícitamente: `length(x) > 0`.

Es también importante entender qué ocurre cuando creas un vector que contiene múltiples tipos usando `c()`: los tipos más complejos siempre ganan.

```
typeof(c(TRUE, 1L))
#> [1] "integer"
typeof(c(1L, 1.5))
#> [1] "double"
typeof(c(1.5, "a"))
#> [1] "character"
```

Copy

Un vector atómico no puede contener una mezcla de diferentes tipos, ya que el tipo es una propiedad del vector completo, no de los elementos individuales. Si necesitas mezclar diferentes tipos en el mismo vector, entonces deberías utilizar una lista, sobre la que aprenderás en breve.

20.4.2 **Funciones de prueba**

Algunas veces quieres diferentes cosas dependiendo del tipo de vector. Una opción es utilizar `typeof()`; otra es usar una función de prueba que devuelva `TRUE` o `FALSE`. Si bien R base provee muchas funciones como `is.vector()` e `is.atomic()`, estas a menudo devuelven resultados inesperados. En su lugar, es más seguro utilizar las funciones `is_*` provistas por el paquete **purrr**, que se resumen en la tabla de más abajo.

	lgl	int	dbl	chr	list
`is_logical()`	x				
`is_integer()`		x			
`is_double()`			x		
`is_numeric()`		x	x		
`is_character()`				x	

	lgl	int	dbl	chr	list
`is_atomic()`	x	x	x	x	
`is_list()`					x
`is_vector()`	x	x	x	x	x

Cada predicado además viene con una versión “escalar”, como `is_scalar_atomic()`, que chequea que la longitud sea 1. Esto es útil, por ejemplo, si quieres chequear que un argumento de tu función sea un solo valor lógico.

20.4.3 Escalares y reglas de reciclado

Así como se coercionan implícitamente los tipos de vectores para que sean compatibles, R también implicitamente coerciona la longitud de los vectores. Esto se denomina **reciclado** de vectores (*vector recicling*), debido a que el vector de menor longitud se repite, o recicla, hasta igualar la longitud del vector más largo. Generalmente, esto es más útil cuando estás trabajando con vectores y “escalares”. Hemos puesto “escalares” entre comillas porque R en realidad no tiene escalares: en su lugar, un solo número es un vector de longitud 1. Debido a que no existen los escalares, la mayoría de las funciones pre-definidas están vectorizadas, lo que implica que operarán en un vector de números. Esa es la razón de por qué, por ejemplo, el siguiente código funciona:

```
sample(10) + 100 # (sample = muestreo)
#>  [1] 107 104 103 109 102 101 106 110 105 108
runif(10) > 0.5
#>  [1] FALSE  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

Copy

En R, las operaciones matemáticas básicas funcionan con vectores. Esto significa que nunca necesitarás la ejecución de una interacción explícita cuando realices cálculos matemáticos sencillos. Es intuitivo lo que debería pasar si agregas dos vectores de la misma longitud, o un vector y un “escalar”. Pero ¿qué sucede si agregas dos vectores de diferentes longitudes?

```
1:10 + 1:2
#>  [1]  2  4  4  6  6  8  8 10 10 12
```

Copy

Aquí, R expandirá el vector más corto a la misma longitud del vector más largo, que es lo que denominamos *reciclaje*. Esto se realiza de manera silenciosa, excepto cuando la longitud del vector más largo no es un múltiplo entero de la longitud del vector más corto:

```
1:10 + 1:3
#> Warning in 1:10 + 1:3: longer object length is not a multiple of shorter object
#> length
#>  [1]  2  4  6  5  7  9  8 10 12 11
```

Copy

Si bien el vector reciclado puede ser usado para crear código sucinto e ingenioso, también puede ocultar problemas de manera silenciosa. Por esta razón, las funciones vectorizadas en el tidyverse mostrarán errores cuando reciclas cualquier otra cosa que no sea un escalar. Si realmente quieres reutilizar, necesitarás hacerlo de manera explícita con `rep()`:

```
tibble(x = 1:4, y = 1:2)
#> Error: Tibble columns must have compatible sizes.
#> * Size 4: Existing data.
#> * Size 2: Column `y`.
#> ℹ Only values of size one are recycled.

tibble(x = 1:4, y = rep(1:2, 2))
#> # A tibble: 4 x 2
#>       x     y
#>   <int> <int>
#> 1     1     1
#> 2     2     2
#> 3     3     1
#> 4     4     2

tibble(x = 1:4, y = rep(1:2, each = 2))
#> # A tibble: 4 x 2
#>       x     y
#>   <int> <int>
#> 1     1     1
#> 2     2     1
#> 3     3     2
#> 4     4     2
```

20.4.4 **Nombrar vectores**

Todos los tipos de vectores pueden ser nombrados. Puedes asignarles un nombre al momento de crearlos con `c()`:

```
c(x = 1, y = 2, z = 4)
#> x y z
#> 1 2 4
```

O después de haberlos creado con `purrr::set_names()`:

```
set_names(1:3, c("a", "b", "c"))
#> a b c
#> 1 2 3
```

Los vectores con nombres son particularmente útiles para la creación de subconjuntos, como se describe a continuación.

20.4.5 **Creación de subconjuntos (*subsetting*)**

Hasta ahora hemos utilizado `dplyr::filter()` para filtrar filas en un tibble. `filter()` solo funciona con tibbles, por lo que necesitaremos una nueva herramienta para trabajar con vectores: `[`. `[` es la función para crear subconjuntos (*subsetting*) y podemos llamarla como `x[a]`. Existen cuatro tipos de cosas con las que puedes un crear un *subset* de un vector:

1. Un vector numérico que contenga solo enteros. Los enteros deben ser todos positivos, todos negativos, o cero.

   Crear subconjuntos con enteros positivos mantiene los elementos en aquellas posiciones:

```
x <- c("uno", "dos", "tres", "cuatro", "cinco")
x[c(3, 2, 5)]
#> [1] "tres"  "dos"   "cinco"
```

   Repitiendo una posición, puedes en realidad generar un output de mayor longitud que el i

```
x[c(1, 1, 5, 5, 5, 2)]
#> [1] "uno"   "uno"   "cinco" "cinco" "cinco" "dos"
```

```
    Los valores negativos eliminan elementos en las posiciones especificadas:
```



```
x[c(-1, -3, -5)]
#> [1] "dos"    "cuatro"
```



```
    Es un error mezclar valores positivos y negativos:
```



```
x[c(1, -1)]
#> Error in x[c(1, -1)]: only 0's may be mixed with negative subscripts
```



```
    El mensaje menciona crear subsets utilizando cero, lo que no retorna valores.
```



```
x[0]
#> character(0)
```



```
    Esto a menudo no es útil, pero puede ser de ayuda si quieres crear estructuras de datos inusuales con las que testear tus funciones.
```


2. Crear subsets con un vector lógico mantiene todos los valores que correspondan al valor `TRUE`. Esto es usualmente útil en conjunto con las funciones de comparación.

```
x <- c(10, 3, NA, 5, 8, 1, NA)

# Todos los valores no faltantes de x
x[!is.na(x)]
#> [1] 10  3  5  8  1

# Todos los valores pares (o faltantes!) de x
x[x %% 2 == 0]
#> [1] 10 NA  8 NA
```


3. Si tienes un vector con nombre, puedes subdivirlo en un vector de tipo caracter.

```
x <- c(abc = 1, def = 2, xyz = 5)
x[c("xyz", "def")]
#> xyz def
#>   5   2
```



```
    Al igual que con los enteros positivos, también puedes usar un vector del tipo caracter p
duplicar entradas individuales.
```


4. El tipo más sencillo de *subsetting* es nada, `x[]`, lo que retorna el valor completo de `x`. Esto no es útil para crear subconjuntos de vectores, pero sí lo es para el caso de las matrices (y otras estructuras de alta dimensionalidad), ya que te permite seleccionar toda las filas o todas las columnas, dejando el índice en blanco. Por ejemplo, si `x` tiene dos dimensiones, `x[1, ]` selecciona la primera fila y todas las columnas y `x[, -1]` selecciona todas las filas y todas las columnas excepto la primera.

Para aprender más acerca de las aplicaciones de la creación de subconjuntos, puedes leer el capítulo "Subsetting" de *Advanced R*: http://adv-r.hadley.nz/Subsetting.html#applications.

Existe una importante variación de `[` llamada `[[`. `[[` solo extrae un único elemento y siempre descarta nombres. Es una buena idea usarla cada vez que quieras dejar en claro que estás extrayendo un único item, como en un bucle *for* (*for loop*). La diferencia entre `[` y `[[` es más importante para el caso de las listas, como veremos en breve.

20.4.6 **Ejercicios**

1. ¿Qué es lo que `mean(is.na(x))` te dice acerca del vector 'x'? ¿Y qué es lo que te dice `sum(!is.finite(x))`?

2. Lee detenidamente la documentación de `is.vector()`. ¿Qué es lo que esta función realmente testea? ¿Por qué la función `is.atomic()` no concuerda con la definición de vectores atómicos vista anteriormente?

3. Compara y contrasta `setNames()` con `purrr::set_names()`.

4. Crea funciones que tomen un vector como input y devuelvan:

    1. El último valor. ¿Deberías usar `[` o `[[`?
    2. Los elementos en posiciones pares.
    3. Cada elemento excepto el último valor.
    4. Solo las posiciones pares (sin valores perdidos).

5. ¿Por qué `x[-which(x > 0)]` no es lo mismo que `x[x <= 0]`?

6. ¿Qué sucede cuando realizas un subset con un entero positivo que es mayor que la longitud del vector? ¿Qué sucede cuando realizas un subset con un nombre que no existe?

20.5 Vectores Recursivos (listas)

Las listas son un escalón más en complejidad respecto de los vectores atómicos, ya que pueden contener otras listas en su interior. Esto las hace adecuadas para representar estructuras jerárquicas o de tipo árbol. Puedes crear una lista con ´list()´:

```
x <- list(1, 2, 3)
x
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] 2
#>
#> [[3]]
#> [1] 3
```

Un herramienta muy útil para trabajar con listas es `str()` ya que se enfoca en la e**str**uctura, no en los contenidos.

```
str(x)
#> List of 3
#>  $ : num 1
#>  $ : num 2
#>  $ : num 3

x_nombrada <- list(a = 1, b = 2, c = 3)
str(x_nombrada)
#> List of 3
#>  $ a: num 1
#>  $ b: num 2
#>  $ c: num 3
```

A diferencia de los vectores atómicos, ´list()´ puede contener una mezcla de objetos:

```
y <- list("a", 1L, 1.5, TRUE)
str(y)
#> List of 4
#>  $ : chr "a"
#>  $ : int 1
#>  $ : num 1.5
#>  $ : logi TRUE
```

¡Las listas incluso pueden contener otras listas!

```
z <- list(list(1, 2), list(3, 4))
str(z)
#> List of 2
#>  $ :List of 2
#>   ..$ : num 1
#>   ..$ : num 2
#>  $ :List of 2
#>   ..$ : num 3
#>   ..$ : num 4
```

Copy

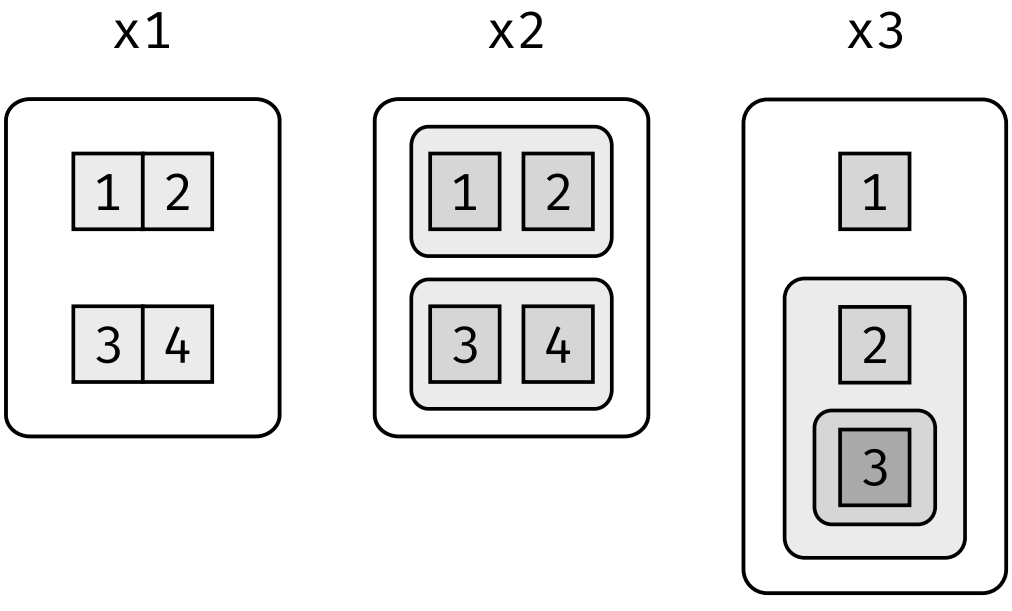
20.5.1 **Visualizando listas**

Para explicar funciones de manipulación de listas más complejas, es útil tener una representacion visual de las listas. Por ejemplo, considera estas tres listas:

```
x1 <- list(c(1, 2), c(3, 4))
x2 <- list(list(1, 2), list(3, 4))
x3 <- list(1, list(2, list(3)))
```

Copy

Así es como las representaremos visualmente:



Existen tres principios en la imagen anterior:

1. Las listas tienen esquinas redondeadas; los vectores atómicos esquinas cuadradas.
2. Los elementos hijos están dibujados dentro de sus padres y tienen un fondo ligeramente más oscuro para facilitar la visualización de la jerarquía.
3. No es importante la orientación de los elementos hijos (esto es, las filas o columnas), por lo que si elegimos determinada orientación será para ahorrar espacio o para ilustrar una propiedad importante en el ejemplo.

20.5.2 **Subconjuntos (*Subsetting*)**

Existen tres maneras de extraer subconjuntos de una lista, los que ilustraremos con una lista denominada `a`:

```
a <- list(a = 1:3, b = "una cadena", c = pi, d = list(-1, -5))
```

Copy

- El corchete simple `[` extrae una sub-lista. El resultado siempre será una lista.

```
str(a[1:2])
#> List of 2
#>  $ a: int [1:3] 1 2 3
#>  $ b: chr "una cadena"
str(a[4])
#> List of 1
#>  $ d:List of 2
#>   ..$ : num -1
#>   ..$ : num -5
```

Copy

```
    Al igual que con los vectores, puedes extraer subconjuntos con un vector lógico, de enter
de caracteres.
```

Copy

- El doble corchete `[[` extrae un solo componente de una lista. Elimina un nivel de la jerarquía de la lista.

```
str(a[[1]])
#>  int [1:3] 1 2 3
str(a[[4]])
#> List of 2
#>  $ : num -1
#>  $ : num -5
```

Copy

- `$` es un atajo para extraer elementos con nombre de una lista. Funciona de modo similar al doble corchete `[[`, excepto que no necesitas utilizar comillas.

```
a$a
#> [1] 1 2 3
a[["a"]]
#> [1] 1 2 3
```

Copy

La diferencia entre `[` y `[[` es muy importante para las listas, ya que `[[` se adentra en una lista mientras que `[` retorna una lista nueva, más pequeña. Compara el código y el output de arriba con la representación visual de la Figura 20.2.

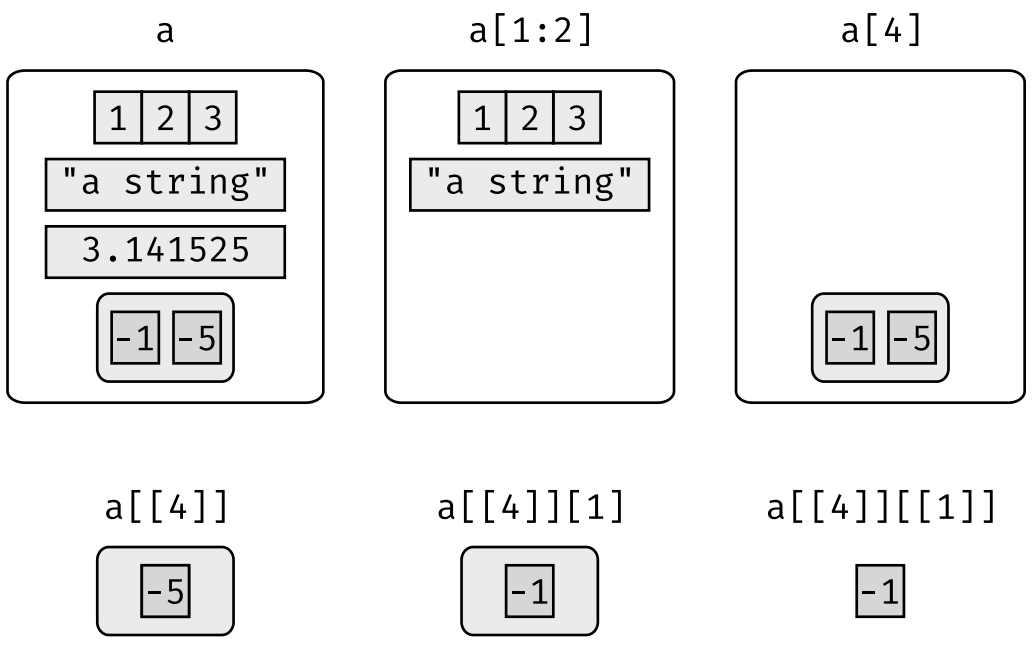
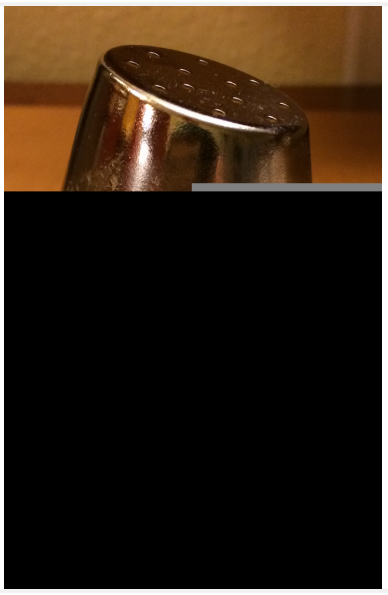


Figure 20.2: Subdividir una lista, de manera visual.

20.5.3 **Listas de Condimentos**

La diferencia entre `[` y `[[` es muy importante, pero es muy fácil confundirse. Para ayudarte a recordar, te mostraremos un pimentero inusual.



Si este pimientero es tu lista `x`, entonces, `x[1]` es un pimientero que contiene un solo paquete de pimienta:

`x[2]` luciría igual, pero contendría el segundo paquete. `x[1:2]` sería un pimentero que contiene dos paquetes de pimienta.

`x[[1]]` es:
Si quisieras obtener el contenido del paquete de pimienta, necesitarías utilizar `x[[1]][[1]]`:

20.5.4 Ejercicios

1.Dibuja las siguientes listas como sets anidados:

```
1. `list(a, b, list(c, d), list(e, f))`
1. `list(list(list(list(list(list(a))))))`
```

Copy

2.¿Qué pasaría si hicieras *subsetting* a un tibble como si fuera una lista? ¿Cuáles son las principales diferencias entre una lista y un tibble?

20.6 Atributos

Cualquier vector puede contener metadatos arbitrarios adicionales mediante sus **atributos**. Puedes pensar en los atributos como una lista de vectores con nombre que pueden ser adjuntadas a cualquier otro objeto. Puedes obtener y definir valores de atributos individuales con `attr()` o verlos todos al mismo tiempo con `attributes()`.

```
x <- 1:10
attr(x, "saludo")
#> NULL
attr(x, "saludo") <- "¡Hola!"
attr(x, " despedida") <- "¡Adiós!"
attributes(x)
#> $saludo
#> [1] "¡Hola!"
#>
#> $` despedida`
#> [1] "¡Adiós!"
```

Copy

Existen tres atributos muy importantes que son utilizados para implementar partes fundamentals de R:

1. Los **nombres** son utilizados para nombrar los elementos de un vector.
2. Las **dimensiones** (o dims, abreviado) hacen que un vector se comporte como una matriz o *array*.
3. La **clase** es utilizada para implementar el sistema orientado a objetos S3.

Ya revisamos los nombres más arriba y no abordaremos las dimensiones porque en este libro no utilizamos matrices. Nos queda describir el atributo *clase*, que controla cómo trabajan las **funciones genéricas**. Las funciones genéricas son clave para la programación orientada a objetos en R, ya que hacen

que las funciones se comporten de manera diferente para diferentes clases de inputs. Está fuera del alcance de este libro tener una discusión más profunda sobre la programacion orientada a objetos, pero puedes leer más al respecto en el libro *Advanced R*: https://adv-r.hadley.nz/s3.html.

Así es como luce una función genérica típica:

```
as.Date
#> function (x, ...)
#> UseMethod("as.Date")
#> <bytecode: 0x7f84cb9e78b0>
#> <environment: namespace:base>
```

La llamada a "UseMethod" significa que esta es una función genérica y que llamará a un **método** específico, esto es, una función basada en la clase del primer argumento. (Todos los métodos son funciones; no todas las funciones son métodos). Puedes listar todos los métodos existentes para una función genérica con `methods()`:

```
methods("as.Date")
#> [1] as.Date.character   as.Date.default     as.Date.factor
#> [4] as.Date.numeric     as.Date.POSIXct     as.Date.POSIXlt
#> [7] as.Date.vctrs_sclr* as.Date.vctrs_vctr*
#> see '?methods' for accessing help and source code
```

Por ejemplo, si `x` es un vector de caracteres, `as.Date()` llamará a `as.Date.character()`; si es un factor, llamará a `as.Date.factor()`.

Puedes ver la implementación específica de un método con: `getS3method()`:

```
getS3method("as.Date", "default")
#> function (x, ...)
#> {
#>     if (inherits(x, "Date"))
#>         x
#>     else if (is.null(x))
#>         .Date(numeric())
#>     else if (is.logical(x) && all(is.na(x)))
#>         .Date(as.numeric(x))
#>     else stop(gettextf("do not know how to convert '%s' to class %s",
#>         deparse1(substitute(x)), dQuote("Date")), domain = NA)
#> }
#> <bytecode: 0x7f84cb86f468>
#> <environment: namespace:base>
getS3method("as.Date", "numeric")
#> function (x, origin, ...)
#> {
#>     if (missing(origin)) {
#>         if (!length(x))
#>             return(.Date(numeric()))
#>         if (!any(is.finite(x)))
#>             return(.Date(x))
#>         stop("'origin' must be supplied")
#>     }
#>     as.Date(origin, ...) + x
#> }
#> <bytecode: 0x7f84cdfb5800>
#> <environment: namespace:base>
```

La S3 genérica más importante es `print()`: controla cómo el objeto es impreso cuando tipeas su nombre en la consola. Otras funciones genéricas importantes son las funciones de subconjuntos `[`, `[[` y `$`.

20.7 Vectores aumentados

Los vectores atómicos y las listas son los bloques sobre los que se construyen otros tipos importantes de vectores, como factores y fechas. A estos vectores le llamamos **vectores aumentados**, ya que son vectores con **atributos** adicionales, incluyendo la clase. Debido a que los vectores aumentados tienen una clase, se

comportan de manera diferente a los vectores atómicos sobre los que están construidos. En este libro, hacemos uso de cuatro importantes vectores aumentados:

- Factores
- Fechas
- Fechas-hora
- Tibbles

A continuación encontrarás una descripción de cada uno de ellos.

20.7.1 **Factores**

Los factores están diseñados para representar datos categóricos que pueden tomar un set fijo de valores posibles. Están construidos sobre la base de enteros y tienen un atributo de *niveles* (*levels*):

```
x <- factor(c("ab", "cd", "ab"), levels = c("ab", "cd", "ef"))
typeof(x)
#> [1] "integer"
attributes(x)
#> $levels
#> [1] "ab" "cd" "ef"
#>
#> $class
#> [1] "factor"
```

20.7.2 **Fechas y fechas-hora**

Las fechas en R son vectores numéricos que representan el número de días desde el 1° de enero de 1970.

```
x <- as.Date("1971-01-01")
unclass(x)
#> [1] 365

typeof(x)
#> [1] "double"
attributes(x)
#> $class
#> [1] "Date"
```

Los vectores fecha-hora son vectores numéricos de clase `POSIXct`, que representan el número de segundos desde el 1° de enero de 1970. (En caso de que te lo preguntes, "POSIXct" es el acrónimo de *"Portable Operating System Interface" calendar time*, es decir, tiempo calendario de la interfaz portable del sistema operativo")

```
x <- lubridate::ymd_hm("1970-01-01 01:00")
unclass(x)
#> [1] 3600
#> attr(,"tzone")
#> [1] "UTC"

typeof(x)
#> [1] "double"
attributes(x)
#> $class
#> [1] "POSIXct" "POSIXt"
#>
#> $tzone
#> [1] "UTC"
```

El atributo `tzone` es opcional. Controla cómo se imprime la hora, no a qué tiempo absoluto hace referencia.

```
attr(x, "tzone") <- "US/Pacific"
x
#> [1] "1969-12-31 17:00:00 PST"
attr(x, "tzone") <- "US/Eastern"
x
#> [1] "1969-12-31 20:00:00 EST"
```

Existe otro tipo de fechas-hora llamado POSIXlt. Estos se construyen sobre la base de listas nombradas (*named lists*).

```
y <- as.POSIXlt(x)
typeof(y)
#> [1] "list"
attributes(y)
#> $names
#>  [1] "sec"    "min"    "hour"   "mday"   "mon"    "year"   "wday"   "yday"
#>  [9] "isdst"  "zone"   "gmtoff"
#>
#> $class
#> [1] "POSIXlt" "POSIXt"
#>
#> $tzone
#> [1] "US/Eastern" "EST"        "EDT"
```

Los POSIXlts son pocos comunes dentro del tidyverse. Sí lo son en R base, ya que son necesarios para extraer components específicos de una fecha, como el año o el mes. Debido a que el paquete **lubridate** provee funciones de ayuda para efectuar dicha extracción, ya no los necesitarás. Siempre es más sencillo trabajar con POSIXct, por lo que si te encuentras con un POSIXlt, deberías convertirlo a un vector de fecha-hora con `lubridate::as_date_time()`.

20.7.3 Tibbles

Los tibbles son listas aumentadas: tienen las clases "tbl_df", "tbl" y "data.frame", y atributos `names` (para columna) y `row.names` (para fila):

```
tb <- tibble::tibble(x = 1:5, y = 5:1)
typeof(tb)
#> [1] "list"
attributes(tb)
#> $names
#> [1] "x" "y"
#>
#> $row.names
#> [1] 1 2 3 4 5
#>
#> $class
#> [1] "tbl_df"     "tbl"        "data.frame"
```

La diferencia entre un tibble y una lista, consiste en que todos los elementos de un data frame deben ser vectores de la misma longitud. Todas las funciones que utilizan tibbles imponen esta condición.

Los data.frames tradicionales tienen una estructura muy similar a los tibbles:

```
df <- data.frame(x = 1:5, y = 5:1)
typeof(df)
#> [1] "list"
attributes(df)
#> $names
#> [1] "x" "y"
#>
#> $class
#> [1] "data.frame"
#>
#> $row.names
#> [1] 1 2 3 4 5
```

La diferencia principal entre ambos es la clase. La clase tibble incluye “data.frame”, lo que significa que los tibbles heredan el comportamiento regular de un data frame por defecto.

20.7.4 **Ejercicios:**

1. ¿Qué valor retorna `hms::hms(3600)` ? ¿Cómo se imprime? ¿Cuál es la tipo primario sobre en el que se basa el vector aumentado? ¿Qué atributos utiliza?

2. Intenta crear un tibble que tenga columnas con diferentes longitudes. ¿Qué es lo que ocurre?

3. Teniendo en cuenta la definición anterior, ¿está bien tener una lista como columna en un tibble?

« 19 Funciones

21 Iteración »

21 Iteración

21.1 Introducción

En funciones, hablamos sobre la importancia de reducir la duplicación en el código creando funciones, en lugar de copiar y pegar. Reducir la duplicación de código tiene tres beneficios principales:

1. Es más fácil ver el objetivo de tu código; lo diferente llama más atención a la vista que aquello que permanece igual.

2. Es más sencillo responder a cambios en los requerimientos. A medida que tus necesidades cambian, solo necesitarás realizar cambios en un lugar, en vez de recordar cambiar en cada lugar donde copiaste y pegaste el código.

3. Es probable que tengas menos errores porque cada línea de código es utilizada en más lugares.

Una herramienta para reducir la duplicación de código son las funciones, que reducen dicha duplicación al identificar patrones repetidos de código y extraerlos en piezas independientes que pueden reutilizarse y actualizarse fácilmente. Otra herramienta para reducir la duplicación es la **iteración**, que te ayuda cuando necesitas hacer la misma tarea con múltiples entradas: repetir la misma operación en diferentes columnas o en diferentes conjuntos de datos. En este capítulo aprenderás sobre dos paradigmas de iteración importantes: la programación imperativa y la programación funcional. Por el lado imperativo, tienes herramientas como *for loops* y *while loops*, que son un gran lugar para comenzar porque hacen que la iteración sea muy explícita, por lo que es obvio qué está pasando. Sin embargo, los bucles *for* son bastante detallados y requieren bastante código que se duplica para cada bucle. La programación funcional (PF) ofrece herramientas para extraer este código duplicado, por lo que cada patrón común de bucle obtiene su propia función. Una vez que domines el vocabulario de PF, podrás resolver muchos problemas comunes de iteración con menos código, mayor facilidad y menos errores.

21.1.1 Prerrequisitos

Una vez que hayas dominado los bucles *for* proporcionados por R base, aprenderás algunas de las potentes herramientas de programación proporcionadas por **purrr**, uno de los paquetes principales de *tidyverse*.

```
library(tidyverse)
library(datos)
```

Copy

21.2 Bucles *for*

Imagina que tenemos este simple *tibble*:

```
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
```

Copy

Queremos calcular la mediana de cada columna. *Podrías hacerlo* copiando y pegando el siguiente código:

On this page

View source

Edit this page

```
median(df$a)
#> [1] -0.2457625
median(df$b)
#> [1] -0.2873072
median(df$c)
#> [1] -0.05669771
median(df$d)
#> [1] 0.1442633
```

Copy

Pero eso rompe nuestra regla de oro: nunca copiar y pegar más de dos veces. En cambio, podríamos usar un bucle *for*:

```
output <- vector("double", ncol(df))  # 1. output
for (i in seq_along(df)) {            # 2. secuencia
  output[[i]] <- median(df[[i]])      # 3. cuerpo
}
output
#> [1] -0.24576245 -0.28730721 -0.05669771  0.14426335
```

Copy

Cada bucle tiene tres componentes:

1. **output**: `output <- vector("double", length(x))`. Antes de comenzar el bucle, siempre debes asignar suficiente espacio para la salida. Esto es muy importante para la eficiencia: si aumentas el bucle *for* en cada iteración usando, por ejemplo, `c()`, el bucle *for* será muy lento.

   Una forma general de crear un vector vacío de longitud dada es la función `vector()`. Tiene dos argumentos: el tipo de vector (*"logical"*, *"integer"*, *"double"*, *"character"*, etc) y su longitud.

2. La **secuencia**: `i in seq_along(df)`. Este código determina sobre qué iterar: cada ejecución del bucle *for* asignará a `i` un valor diferente de `seq_along(df)`. Es útil pensar en `i` como un pronombre, como "eso".

   Es posible que no hayas visto `seq_along()` con anterioridad. Es una versión segura de la más familiar `1:length(l)`, con una diferencia importante: si se tiene un vector de longitud cero, `seq_along()` hace lo correcto:

   ```
   y <- vector("double", 0)
   seq_along(y)
   #> integer(0)
   1:length(y)
   #> [1] 1 0
   ```

   Copy

   Probablemente no vas a crear un vector de longitud cero deliberadamente, pero es fácil crearlos accidentalmente. Si usamos `1: length(x)` en lugar de `seq_along(x)`, es posible que obtengamos un mensaje de error confuso.

3. El **cuerpo**: `output[[i]] <- median(df[[i]])`. Este es el código que hace el trabajo. Se ejecuta repetidamente, con un valor diferente para `i` cada vez. La primera iteración ejecutará `output[[1]] <- median(df[[1]])`, la segunda ejecutará `output [[2]] <- median (df [[2]])`, y así sucesivamente.

¡Eso es todo lo que hay para el bucle *for*! Ahora es un buen momento para practicar creando algunos bucles *for* básicos (y no tan básicos) usando los ejercicios que se encuentran a continuación. Luego avanzaremos en algunas variaciones de este bucle que te ayudarán a resolver otros problemas que surgirán en la práctica.

21.2.1 Ejercicios

1. Escribe bucles *for* para:

   1. Calcular la media de cada columna en `datos::mtautos`.
   2. Determinar el tipo de cada columna en `datos::vuelos`.
   3. Calcular el número de valores únicos en cada columna de `datos::flores`.
   4. Generar diez normales aleatorias de distribuciones con medias -10, 0, 10 y 100.

Piensa en el resultado, la secuencia y el cuerpo **antes** de empezar a escribir el bucle.

2. Elimina el bucle *for* en cada uno de los siguientes ejemplos aprovechando alguna función existente que trabaje con vectores:

```
out <- ""
for (x in letters) {
  out <- stringr::str_c(out, x)
}

x <- sample(100)
sd <- 0
for (i in seq_along(x)) {
  sd <- sd + (x[i] - mean(x)) ^ 2
}
sd <- sqrt(sd / (length(x) - 1))

x <- runif(100)
out <- vector("numeric", length(x))
out[1] <- x[1]
for (i in 2:length(x)) {
  out[i] <- out[i - 1] + x[i]
}
```

3. Combina tus habilidades para escribir funciones y bucles *for*:

    1. Escribe un bucle *for* que imprima (`prints()`) la letra de la canción de niños “Cinco ranitas verdes” (u otra).

    2. Convierte la canción infantil “Cinco monitos saltaban en la cama” en una función. Generalizar a cualquier cantidad de monitos en cualquier estructura para dormir.

    3. Convierte la canción “99 botellas de cerveza en la pared” en una función. Generalizar a cualquier cantidad, de cualquier tipo de recipiente que contenga cualquier líquido sobre cualquier superficie.

4. Es común ver bucles *for* que no preasignan el output y en su lugar aumentan la longitud de un vector en cada paso:

```
output <- vector("integer", 0)
for (i in seq_along(x)) {
  output <- c(output, lengths(x[[i]]))
}
output
```

¿Cómo afecta esto el rendimiento? Diseña y ejecuta un experimento.

21.3 **Variaciones de bucles *for***

Una vez que tienes el bucle *for* básico en tu haber, hay algunas variaciones que debes tener en cuenta. Estas variaciones son importantes independientemente de cómo hagas la iteración, así que no te olvides de ellas una vez que hayas dominado las técnicas de programación funcional (PF) que aprenderás en la próxima sección.

Hay cuatro variaciones del bucle *for* básico:

1. Modificar un objeto existente, en lugar de crear un nuevo objeto.
2. Iterar sobre nombres o valores, en lugar de índices.
3. Manejar outputs de longitud desconocida.
4. Manejar secuencias de longitud desconocida.

21.3.1 **Modificar un objeto existente**

Algunas veces querrás usar un bucle *for* para modificar un objeto existente. Por ejemplo, recuerda el desafío que teníamos en el capítulo sobre funciones. Queríamos reescalar cada columna en un *data frame*:

```r
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}

df$a <- rescale01(df$a)
df$b <- rescale01(df$b)
df$c <- rescale01(df$c)
df$d <- rescale01(df$d)
```

Para resolver esto con un bucle *for*, volvamos a pensar en los tres componentes:

1. **Output**: ya tenemos el *output* — ¡es lo mismo que la entrada!

2. **Secuencia**: podemos pensar en un *data frame* como una lista de columnas, por lo que podemos iterar sobre cada columna con `seq_along(df)`.

3. **Cuerpo**: aplicar `rescale01()`.

Esto nos da:

```r
for (i in seq_along(df)) {
  df[[i]] <- rescale01(df[[i]])
}
```

Por lo general, se modificará una lista o un *data frame* con este tipo de bucle, así que recuerda utilizar `[[` y no `[`. Te habrás fijado que usamos `[[` en todos nuestros bucles *for*: creemos que es mejor usar `[[` incluso para vectores atómicos porque deja en claro que queremos trabajar con un solo elemento.

21.3.2 Patrones de bucle

Hay tres formas básicas de hacer un bucle sobre un vector. Hasta ahora hemos visto la más general: iterar sobre los índices numéricos con `for (i in seq_along(xs))`, y extraer el valor con `x[[i]]`. Hay otras dos formas:

1. Iterar sobre los elementos: `for (x in xs)`. Esta forma es la más útil si solo te preocupas por los efectos secundarios, como graficar o grabar un archivo, porque es difícil almacenar el output de forma eficiente.

2. Iterar sobre los nombres: `for (nm in names(xs))`. Esto te entrega el nombre, que se puede usar para acceder al valor con `x[[nm]]`. Esto es útil si queremos utilizar el nombre en el título de un gráfico o en el nombre de un archivo. Si estás creando un output con nombre, asegúrate de nombrar el vector de resultados de esta manera:

   ```r
   resultados <- vector("list", length(x))
   names(resultados) <- names(x)
   ```

3. Iterar sobre los índices numéricos es la forma más general, porque dada la posición se puede extraer tanto el nombre como el valor:

```r
for (i in seq_along(x)) {
  name <- names(x)[[i]]
  value <- x[[i]]
}
```

21.3.3 Longitud de *output* desconocida

Es posible que algunas veces no sepas el tamaño que tendrá el output. Por ejemplo, imagina que quieres simular algunos vectores aleatorios de longitudes aleatorias. Podrías tener la tentación de resolver este problema haciendo crecer el vector progresivamente:

```
medias <- c(0, 1, 2)

output <- double()
for (i in seq_along(medias)) {
  n <- sample(100, 1)
  output <- c(output, rnorm(n, medias[[i]]))
}
str(output)
#>  num [1:138] 0.912 0.205 2.584 -0.789 0.588 ...
```

Pero esto no es muy eficiente porque en cada iteración, R tiene que copiar todos los datos de las iteraciones anteriores. En términos técnicos, obtienes un comportamiento "cuadrático" ($O(n^2)$), lo que significa que un bucle que tiene tres veces más elementos tomaría nueve (3^2) veces más tiempo en ejecutarse.

Una mejor solución es guardar los resultados en una lista y luego combinarlos en un solo vector una vez que se complete el ciclo:

```
out <- vector("list", length(medias))
for (i in seq_along(medias)) {
  n <- sample(100, 1)
  out[[i]] <- rnorm(n, medias[[i]])
}
str(out)
#> List of 3
#>  $ : num [1:76] -0.3389 -0.0756 0.0402 0.1243 -0.9984 ...
#>  $ : num [1:17] -0.11 1.149 0.614 0.77 1.392 ...
#>  $ : num [1:41] 1.88 2.46 2.62 1.82 1.88 ...
str(unlist(out))
#>  num [1:134] -0.3389 -0.0756 0.0402 0.1243 -0.9984 ...
```

Aquí usamos `unlist()` (*deslistar* en inglés) para aplanar una lista de vectores en un solo vector. Una opción más estricta es usar `purrr::flatten_dbl()` (*aplanar dobles*) — arrojará un error si el input no es una lista de dobles.

Este patrón ocurre también en otros lugares:

1. Podrías estar generando una cadena larga. En lugar de pegar (`paste()`) cada iteración con la anterior, guarda el output en un vector de caracteres y luego combina ese vector en una cadena con `paste(output, collapse = "")`.
2. Podrías estar generando un *data frame* grande. En lugar de enlazar (`rbind()`) secuencialmente en cada iteración, guarda el resultado en una lista y luego utiliza `dplyr::bind_rows(output)` para combinar el output en un solo *data frame*.

Cuidado con este patrón. Cuando lo veas, cambia a un objeto de resultado más complejo y luego combínalo en un solo paso al final.

21.3.4 **Longitud de secuencia desconocida**

A veces ni siquiera sabemos cuánto tiene que durar la secuencia de entrada. Esto es común cuando se hacen simulaciones. Por ejemplo, es posible que se quiera realizar un bucle hasta que se obtengan tres caras seguidas. No podemos hacer ese tipo de iteración con un bucle *for*. En su lugar, podemos utilizar un bucle *while* (*mientras*, en inglés). Un bucle *while* es más simple que un bucle *for* porque solo tiene dos componentes, una condición y un cuerpo:

```
while (condición) {
  # cuerpo
}
```

Un bucle *while* también es más general que un bucle *for*, porque podemos reescribir este último como un bucle *while*, pero no podemos reescribir todos los bucles *while* bucles *for*:

```
for (i in seq_along(x)) {
  # cuerpo
}

# Equivalente a
i <- 1
while (i <= length(x)) {
  # cuerpo
  i <- i + 1
}
```

Así es como podríamos usar un bucle *while* para encontrar cuántos intentos se necesitan para obtener tres caras seguidas:

```
lanzamiento <- function() sample(c("S", "C"), 1)

lanzamientos <- 0
ncaras <- 0

while (ncaras < 3) {
  if (lanzamiento() == "C") {
    ncaras <- ncaras + 1
  } else {
    ncaras <- 0
  }
  lanzamientos <- lanzamientos + 1
}
lanzamientos
#> [1] 21
```

Mencionamos los bucles *while* brevemente, porque casi nunca los usamos. Se utilizan con mayor frecuencia para hacer simulaciones, un tema que está fuera del alcance de este libro. Sin embargo, es bueno saber que existen en caso que nos encontremos con problemas en los que el número de iteraciones no se conoce de antemano.

21.3.5 **Ejercicios**

1. Imagina que tienes un directorio lleno de archivos CSV que quieres importar. Tienes sus ubicaciones en un vector, `files <- dir("data/", pattern = "\\.csv$", full.names = TRUE)`, y ahora quieres leer cada uno con `read_csv()`. Escribe un bucle *for* que los cargue en un solo *data frame*.

2. ¿Qué pasa si utilizamos `for (nm in names(x))` y `x` no tiene nombres (*names*)? ¿Qué pasa si solo algunos elementos están nombrados? ¿Qué pasa si los nombres no son únicos?

3. Escribe una función que imprima el promedio de cada columna numérica en un *data frame*, junto con su nombre. Por ejemplo, `mostrar_promedio(flores)` debe imprimir:

   ```
   mostrar_promedio(flores)
   #> Largo.Sepalo:  5.84
   #> Ancho.Sepalo:  3.06
   #> Largo.Petalo:  3.76
   #> Ancho.Petalo:  1.20
   ```

   (Desafío adicional: ¿qué función utilizamos para asegurarnos que los números queden alineados a pesar que los nombres de las variables tienen diferentes longitudes?)

4. ¿Qué hace este código? ¿Cómo funciona?

```
trans <- list(
  cilindrada = function(x) x * 0.0163871,
  transmision = function(x) {
    factor(x, labels = c("automática", "manual"))
  }
)
for (var in names(trans)) {
  mtautos[[var]] <- trans[[var]](mtautos[[var]])
}
```

21.4 Bucles *for* vs. funcionales

Los bucles *for* no son tan importantes en R como en otros lenguajes porque R es un lenguaje de programación funcional. Esto significa que es posible envolver los bucles en una función y llamar a esa función en lugar de usar el bucle *for* directamente.

Para ver por qué esto es importante, consideremos (nuevamente) este *data frame* simple:

```
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
```

Imagina que quieres calcular la media de cada columna. Podríamos hacer eso con un bucle *for*:

```
output <- vector("double", length(df))
for (i in seq_along(df)) {
  output[[i]] <- mean(df[[i]])
}
output
#> [1] -0.3260369  0.1356639  0.4291403 -0.2498034
```

Como te das cuenta que vas querer calcular los promedios de cada columna con bastante frecuencia, extraer el bucle en una función:

```
col_media <- function(df) {
  output <- vector("double", length(df))
  for (i in seq_along(df)) {
    output[i] <- mean(df[[i]])
  }
  output
}
```

Pero entonces pensamos que también sería útil poder calcular la mediana y la desviación estándar, así que copiamos y pegamos la función `col_media()` y reemplazamos `mean()` con `median()` y `sd()`:

```
col_mediana <- function(df) {
  output <- vector("double", length(df))
  for (i in seq_along(df)) {
    output[i] <- median(df[[i]])
  }
  output
}
col_desvest <- function(df) {
  output <- vector("double", length(df))
  for (i in seq_along(df)) {
    output[i] <- sd(df[[i]])
  }
  output
}
```

¡Oh oh! Copiaste y pegaste este código dos veces, por lo que es hora de pensar cómo generalizarlo. Ten en cuenta que la mayoría de este código corresponde al texto trillado del bucle *for*, lo que hace difícil ver la única cosa que es diferente entre las funciones (`mean()`, `median()`, `sd()`).

¿Qué podŕias hacer si ves un conjunto de funciones como esta?:

```
f1 <- function(x) abs(x - mean(x)) ^ 1
f2 <- function(x) abs(x - mean(x)) ^ 2
f3 <- function(x) abs(x - mean(x)) ^ 3
```

Copy

Por suerte, habrás notado que hay mucha duplicación que puedes extraer con un argumento adicional:

```
f <- function(x, i) abs(x - mean(x)) ^ i
```

Copy

Redujiste la posibilidad de errores (porque ahora tienes 1/3 menos de código) y hiciste más fácil generalizar para situaciones nuevas.

Podemos hacer exactamente lo mismo con `col_media()`, `col_mediana()` y `col_desvest()` agregando un argumento que proporciona la función a aplicar en cada columna:

```
col_resumen <- function(df, fun) {
  out <- vector("double", length(df))
  for (i in seq_along(df)) {
    out[i] <- fun(df[[i]])
  }
  out
}
col_resumen(df, median)
#> [1] -0.51850298  0.02779864  0.17295591 -0.61163819
col_resumen(df, mean)
#> [1] -0.3260369  0.1356639  0.4291403 -0.2498034
```

Copy

La idea de pasar una función a otra es extremadamente poderosa y es uno de los comportamientos que hacen de R un lenguaje de programación funcional. Puede que te tome un tiempo comprender la idea, pero vale la pena el esfuerzo. En el resto del capítulo aprenderás y usarás el paquete **purrr**, que proporciona funciones que eliminan la necesidad de muchos de los bucles comunes. La familia de funciones de R base *apply* (*aplicar*: `apply()`, `lapply()`, `tapply()`, etc.) resuelve un problema similar; sin embargo, *purrr* es más consistente y, por lo tanto, es más fácil de aprender.

El objetivo de usar las funciones de *purrr* en lugar de los bucles es que te permite dividir los desafíos comunes de manipulación de listas en partes independientes:

1. ¿Cómo puedes resolver el problema para un solo elemento de la lista? Una vez que encuentres la solución, *purrr* se encargará de generalizarla a cada elemento de la lista.
2. Si estás resolviendo un problema complejo, ¿cómo puedes dividirlo en pequeñas etapas que te permitan avanzar paso a paso hacia la solución? Con **purrr** obtienes muchas piezas pequeñas que puedes ensamblar utilizando el *pipe* (`%>%`).

Esta estructura facilita la resolución de nuevos problemas. También hace que sea más fácil entender las soluciones a problemas antiguos cuando relees código que escribiste en el pasado.

21.4.1 Ejercicios

1. Lee la documentación para `apply()`. En el caso 2d, ¿qué dos bucles *for* generaliza?
2. Adapta `col_resumen()` para que solo se aplique a las columnas numéricas. Podrías querer comenzar con la función `is_numeric()` que devuelve un vector lógico que tenga un *TRUE* por cada columna numérica.

21.5 Las funciones *map*

El patrón de iterar sobre un vector, hacer algo con cada elemento y guardar los resultados es tan común que el paquete **purrr** proporciona una familia de funciones que lo hacen por ti. Hay una función para cada tipo de output:

- `map()` crea una lista.
- `map_lgl()` crea un vector lógico.
- `map_int()` crea un vector de enteros.
- `map_dbl()` crea un vector de dobles.
- `map_chr()` crea un vector de caracteres.

Cada función `map` (*mapa*, en español) toma un vector como input, aplica una función a cada elemento y luego devuelve un nuevo vector que tiene la misma longitud (y los mismos nombres) que el input. El tipo de vector está determinado por el sufijo de la función *map*.

Una vez que domines estas funciones, descubrirás que lleva mucho menos tiempo resolver los problemas de iteración. Sin embargo, nunca debes sentirse mal por usar un bucle *for* en lugar de una función *map*. Las funciones *map* son un nivel superior de abstracción y puede llevar mucho tiempo entender cómo funcionan. Lo importante es que resuelvas el problema en el que estás trabajando, no que escribas el código más conciso y elegante (¡aunque eso es definitivamente algo a lo que aspirar!).

Algunas personas te dirán que evites los bucles *for* porque son lentos. ¡Están equivocados! (Bueno, al menos están bastante desactualizados, ya que los bucles *for* han dejado de ser lentos desde hace muchos años). Los principales beneficios de usar funciones como `map()` no es la velocidad, sino la claridad: hacen que tu código sea más fácil de escribir y leer.

Podemos usar estas funciones para realizar los mismos cálculos que el último bucle *for*. Esas funciones de resumen devolvían valores decimales, por lo que necesitamos usar `map_dbl()`:

```
map_dbl(df, mean)
#>          a          b          c          d
#> -0.3260369  0.1356639  0.4291403 -0.2498034
map_dbl(df, median)
#>           a           b           c           d
#> -0.51850298  0.02779864  0.17295591 -0.61163819
map_dbl(df, sd)
#>         a         b         c         d
#> 0.9214834 0.4848945 0.9816016 1.1563324
```

Comparado con el uso de un bucle for, el foco está en la operación que se está realizando (es decir, `mean()`, `median()`, `sd()`), y no en llevar la cuenta de las acciones requeridas para recorrer cada elemento y almacenar el output. Esto es aún más evidente si usamos el *pipe*:

```
df %>% map_dbl(mean)
#>          a          b          c          d
#> -0.3260369  0.1356639  0.4291403 -0.2498034
df %>% map_dbl(median)
#>           a           b           c           d
#> -0.51850298  0.02779864  0.17295591 -0.61163819
df %>% map_dbl(sd)
#>         a         b         c         d
#> 0.9214834 0.4848945 0.9816016 1.1563324
```

Existen algunas diferencias entre `map_*()` y `col_resumen()`:

- Todas las funciones de purrr están implementadas en C. Esto las hace más rápidas a expensas de la legibilidad.
- El segundo argumento, `.f`, la función a aplicar, puede ser una fórmula, un vector de caracteres o un vector de enteros. Aprenderás acerca de estos atajos útiles en la siguiente sección.
- `map_*()` usa ... ([dot dot dot] - punto punto punto) para pasar los argumentos adicionales a `.f` cada vez que se llama:

```
map_dbl(df, mean, trim = 0.5)
#>           a           b           c           d
#> -0.51850298  0.02779864  0.17295591 -0.61163819
```

- Las funciones *map* también preservan los nombres:

```
z <- list(x = 1:3, y = 4:5)
map_int(z, length)
#> x y
#> 3 2
```

21.5.1 Atajos

Existen algunos atajos que puedes usar con `.f` para ahorrar algo de tipeo Imagina que quieres ajustar un modelo lineal a cada grupo en un conjunto de datos. El siguiente ejemplo de juguete divide el dataset `mtautos` en tres partes (una para cada valor de cilindro) y ajusta el mismo modelo lineal a cada parte:

```
modelos <- mtautos %>%
  split(.$cilindros) %>%
  map(function(df) lm(millas ~ peso, data = df))
```

La sintaxis para crear una función anónima en R es bastante verbosa, por lo que purrr provee un atajo conveniente: una fórmula unilateral.

```
modelos <- mtautos %>%
  split(.$cilindros) %>%
  map(~lm(millas ~ peso, data = .))
```

Hemos usado `.` como pronombre: se refiere al elemento actual de la lista (del mismo modo que `i` se refiere al índice actual en el loop for).

Cuando examinas múltiples modelos, puedes querer extraer un estadístico resumen como lo es R^2. Para hacer eso primero necesitas correr `summary()` y luego extraer la componente `r.squared` (R-cuadrado). Podríamos hacerlo usando un atajo para las funciones anónimas:

```
modelos %>%
  map(summary) %>%
  map_dbl(~.$r.squared)
#>         4         6         8
#> 0.5086326 0.4645102 0.4229655
```

Sin embargo, extraer componentes con nombres es una operación común, por lo que purrr provee un atajo aún más corto: puedes usar una cadena de caracteres (o *string*).

```
modelos %>%
  map(summary) %>%
  map_dbl("r.squared")
#>         4         6         8
#> 0.5086326 0.4645102 0.4229655
```

También puedes usar un entero para seleccionar elementos de acuerdo a su posición:

```
x <- list(list(1, 2, 3), list(4, 5, 6), list(7, 8, 9))
x %>% map_dbl(2)
#> [1] 2 5 8
```

21.5.2 R Base

Si la familia de funciones *apply* en R base te son familiares, podrás haber notado algunas similitudes con las funciones de purrr:

- `lapply()` es básicamente idéntica a `map()`, excepto que `map()` es consistente con todas las otras funciones de purrr y puedes usar atajos para `.f`.

- `sapply()` es un envoltorio (*wrapper*) de `lapply()` que automáticamente simplifica el output. Esto es úti lpara el trabajo interactivo pero es problemático en una función, ya que nunca sabrás qué tipo de output vas a obtener:

```
x1 <- list(
  c(0.27, 0.37, 0.57, 0.91, 0.20),
  c(0.90, 0.94, 0.66, 0.63, 0.06),
  c(0.21, 0.18, 0.69, 0.38, 0.77)
)
x2 <- list(
  c(0.50, 0.72, 0.99, 0.38, 0.78),
  c(0.93, 0.21, 0.65, 0.13, 0.27),
  c(0.39, 0.01, 0.38, 0.87, 0.34)
)

umbral <- function(x, cutoff = 0.8) x[x > cutoff]
x1 %>% sapply(umbral) %>% str()
#> List of 3
#>  $ : num 0.91
#>  $ : num [1:2] 0.9 0.94
#>  $ : num(0)
x2 %>% sapply(umbral) %>% str()
#>  num [1:3] 0.99 0.93 0.87
```

Copy

- `vapply()` es una alternativa más segura a `sapply()` porque debes ingresar un argumento adicional que define el tipo de output. El único problema con `vapply()` es es que requiere mucha escritura: `vapply(df, is.numeric, logical(1))` es equivalente a `map_lgl(df, is.numeric)`. Una ventaja de `vapply()` sobre las funciones *map* de purrr es que también puede generar matrices — las funciones *map* solo generan vectores.

Aquí nos enfocamos en las funciones de purrr, ya que proveen nombres y argumentos consistentes, atajos útiles y en el futuro proveerán paralelización simple y barras de progreso.

21.5.3 Ejercicios

1. Escribe un código que use una de las funciones de map para:
    1. Calcular la media de cada columna en `datos::mautos`.
    2. Obtener de qué tipo es cada columna en `datos::vuelos`.
    3. Calcular la cantidad de valores únicos en cada columna de `datos::flores`.
    4. Generar diez normales aleatorias de distribuciones con medias -10, 0, 10 y 100.
2. ¿Cómo puedes crear un vector tal que para cada columna en un data frame indique si corresponde o no a un factor?
3. ¿Qué ocurre si usas las funciones *map* en vectores que no son listas? ¿Qué hace `map(1:5, runif)`? ¿Por qué?
4. ¿Qué hace `map(-2:2, rnorm, n = 5)`? ¿Por qué? ¿Qué hace `map_dbl(-2:2, rnorm, n = 5)`? ¿Por qué?
5. Reescribe `map(x, function(df) lm(mpg ~ wt, data = df))` para eliminar todas las funciones anónimas.

21.6 Manejando los errores

Cuando usas las funciones *map* para repetir muchas operaciones, la probabilidad de que una de estas falle es mucho más alta. Cuando esto ocurre, obtendrás un mensaje de error y no una salida. Esto es molesto: ¿por qué un error evita que accedas a todo lo que sí funcionó? ¿Cómo puedes asegurarte de que una manzana podrida no arruine todo el barril?

En esta sección aprenderás a manejar estas situaciones con una nueva función: `safely()` (*de forma segura*, en inglés). `safely()` es un adverbio: toma una función (un verbo) y entrega una versión modificada. En este caso, la función modificada nunca lanzará un error. En cambio, siempre devolverá una lista de dos elementos:

1. `result` es el resultado original. Si hubo un error, aparecerá como `NULL`,
2. `error` es un objeto de error. Si la operación fue exitosa, será `NULL`.

(Puede que estés familiarizado con la función `try()` (*intentar*) de R base. Es similar, pero dado que a veces entrega el resultado original y a veces un objeto de error, es más díficil para trabajar.)

Veamos esto con un ejemplo simple: `log()`:

```
log_seguro <- safely(log)
str(log_seguro(10))
#> List of 2
#>  $ result: num 2.3
#>  $ error : NULL
str(log_seguro("a"))
#> List of 2
#>  $ result: NULL
#>  $ error :List of 2
#>   ..$ message: chr "non-numeric argument to mathematical function"
#>   ..$ call   : language .Primitive("log")(x, base)
#>   ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```

Copy

Cuando la función es exitosa, el elemento `result` contiene el resultado y el elemento `error` es `NULL`. Cuando la función falla, el elemento `result` es `NULL` y el elemento `error` contiene un objeto de error.

`safely()` está diseñada para funcionar con *map*:

```
x <- list(1, 10, "a")
y <- x %>% map(safely(log))
str(y)
#> List of 3
#>  $ :List of 2
#>   ..$ result: num 0
#>   ..$ error : NULL
#>  $ :List of 2
#>   ..$ result: num 2.3
#>   ..$ error : NULL
#>  $ :List of 2
#>   ..$ result: NULL
#>   ..$ error :List of 2
#>   .. ..$ message: chr "non-numeric argument to mathematical function"
#>   .. ..$ call   : language .Primitive("log")(x, base)
#>   .. ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```

Copy

Esto sería más fácil de trabajar si tuvieramos dos listas: una con todos los errores y otra con todas las salidas, Esto es fácil de obtener con `purrr::transpose()` (*transponer*):

```
y <- y %>% transpose()
str(y)
#> List of 2
#>  $ result:List of 3
#>   ..$ : num 0
#>   ..$ : num 2.3
#>   ..$ : NULL
#>  $ error :List of 3
#>   ..$ : NULL
#>   ..$ : NULL
#>   ..$ :List of 2
#>   .. ..$ message: chr "non-numeric argument to mathematical function"
#>   .. ..$ call   : language .Primitive("log")(x, base)
#>   .. ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```

Copy

Queda a tu criterio cómo manejar los errores, pero típicamente puedes mirar los valores de `x` donde `y` es un error, o trabajar con los valores que `y` que están ok:

```
estan_ok <- y$error %>% map_lgl(is_null)
x[!estan_ok]
#> [[1]]
#> [1] "a"
y$result[estan_ok] %>% flatten_dbl()
#> [1] 0.000000 2.302585
```

Purrr provee otros dos adverbios útiles:

- Al igual que `safely()`, `possibly()` (*posiblemente*) siempre tendrá éxito. Es más simple que `safely()`, ya que le das un valor por defecto para devolver cuando haya un error.

```
x <- list(1, 10, "a")
x %>% map_dbl(possibly(log, NA_real_))
#> [1] 0.000000 2.302585       NA
```

- `quietly()` (*silenciosamente*) tiene un rol similar a `safely()`, pero en lugar de capturar los errores, captura el output impreso, los mensajes y las advertencias:

```
x <- list(1, -1)
x %>% map(quietly(log)) %>% str()
#> List of 2
#>  $ :List of 4
#>   ..$ result  : num 0
#>   ..$ output  : chr ""
#>   ..$ warnings: chr(0)
#>   ..$ messages: chr(0)
#>  $ :List of 4
#>   ..$ result  : num NaN
#>   ..$ output  : chr ""
#>   ..$ warnings: chr "NaNs produced"
#>   ..$ messages: chr(0)
```

21.7 Usar *map* sobre múltiples argumentos

Hasta ahora hemos "mapeado" sobre un único input. Pero a menudo tienes multiples inputs relacionados y necesitas iterar sobre ellos en paralelo. Ese es el trabajo de las funciones `map2()` y `pmap()`. Por ejemplo, imagina que quieres simular normales aleatorias con distintas medias. Ya sabes hacerlo con `map()`:

```
mu <- list(5, 10, -3)
mu %>%
  map(rnorm, n = 5) %>%
  str()
#> List of 3
#>  $ : num [1:5] 5.63 7.1 4.39 3.37 4.99
#>  $ : num [1:5] 9.34 9.33 9.52 11.32 10.64
#>  $ : num [1:5] -2.49 -4.75 -2.11 -2.78 -2.42
```

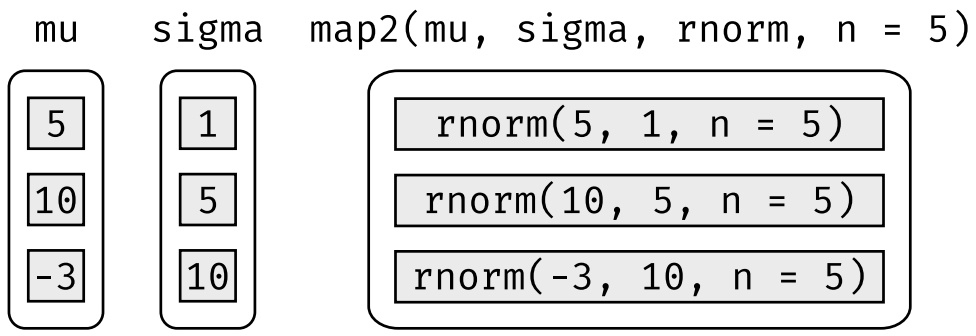
¿Qué ocurre si también necesitas cambiar la desviación estándar? Una forma de hacerlo sería iterar sobre los índices e indexar en vectores de medias y desviaciones estándar:

```
sigma <- list(1, 5, 10)
seq_along(mu) %>%
  map(~rnorm(5, mu[[.]], sigma[[.]])) %>%
  str()
#> List of 3
#>  $ : num [1:5] 4.82 5.74 4 2.06 5.72
#>  $ : num [1:5] 6.51 0.529 10.381 14.377 12.269
#>  $ : num [1:5] -11.51 2.66 8.52 -10.56 -7.89
```

Pero esto oscurece la intención del código. En su lugar podríamos usar `map2()`, que itera sobre dos vectores en paralelo:

```
map2(mu, sigma, rnorm, n = 5) %>% str()
#> List of 3
#>  $ : num [1:5] 3.83 4.52 5.12 3.23 3.59
#>  $ : num [1:5] 13.55 3.8 8.16 12.31 8.39
#>  $ : num [1:5] -15.872 -13.3 12.141 0.469 14.794
```

`map2()` genera esta serie de llamadas a funciones:



Observa que los argumentos que varían para cada llamada van *antes* de la función; argumentos que son los mismos para cada llamada van *después*.

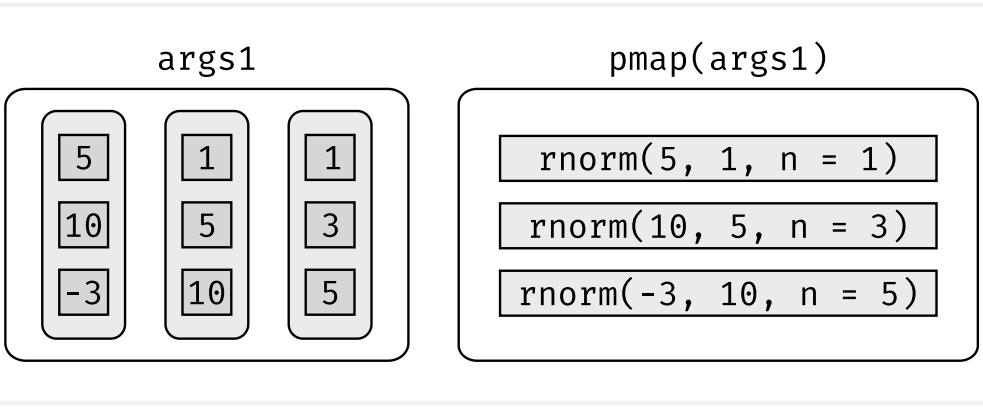
Al igual que `map()`, `map2()` es un envoltorio en torno a un bucle *for*:

```
map2 <- function(x, y, f, ...) {
  out <- vector("list", length(x))
  for (i in seq_along(x)) {
    out[[i]] <- f(x[[i]], y[[i]], ...)
  }
  out
}
```

También te podrás imaginar `map3()`, `map4()`, `map5()`, `map6()`, etc., pero eso se volvería tedioso rápidamente. En cambio, purrr provee `pmap()`, que toma una lista de argumentos. Puedes usar eso si quieres cambiar la media, desviación estándar y el número de muestras:

```
n <- list(1, 3, 5)
args1 <- list(n, mu, sigma)
args1 %>%
  pmap(rnorm) %>%
  str()
#> List of 3
#>  $ : num 5.39
#>  $ : num [1:3] 5.41 2.08 9.58
#>  $ : num [1:5] -23.85 -2.96 -6.56 8.46 -5.21
```

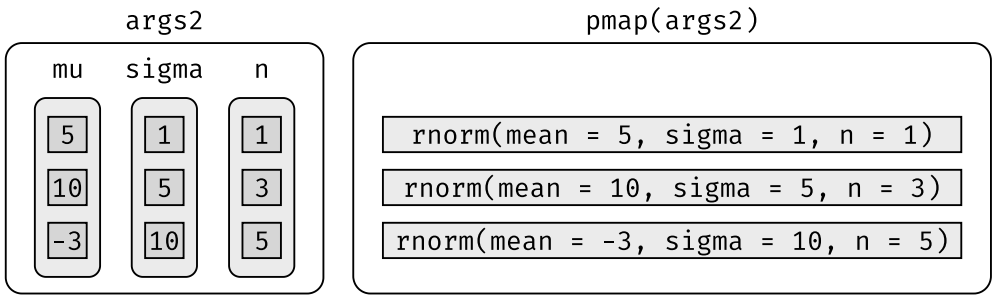
Esto se ve así:



Si no nombras todos los elementos de la lista, `pmap()` usará una coincidencia posicional al llamar la función. Esto es un poco frágil y hace el código más difícil de leer, por lo que es mejor nombrar los argumentos:

```
args2 <- list(mean = mu, sd = sigma, n = n)
args2 %>%
  pmap(rnorm) %>%
  str()
```

Esto genera llamadas más largas, pero más seguras:



Dado que los argumentos son todos del mismo largo, tiene sentido almacenarlos en un data frame:

```
params <- tribble(
  ~mean, ~sd, ~n,
    5,     1,  1,
   10,     5,  3,
   -3,    10,  5
)
params %>%
  pmap(rnorm)
#> [[1]]
#> [1] 6.018179
#>
#> [[2]]
#> [1]  8.681404 18.292712  6.129566
#>
#> [[3]]
#> [1] -12.239379  -5.755334  -8.933997  -4.222859   8.797842
```

Utilizar un data frame cuando tu código se vuelve complicado nos parece una buena aproximación, ya que asegura que cada columna tenga nombre y el mismo largo que las demás columnas.

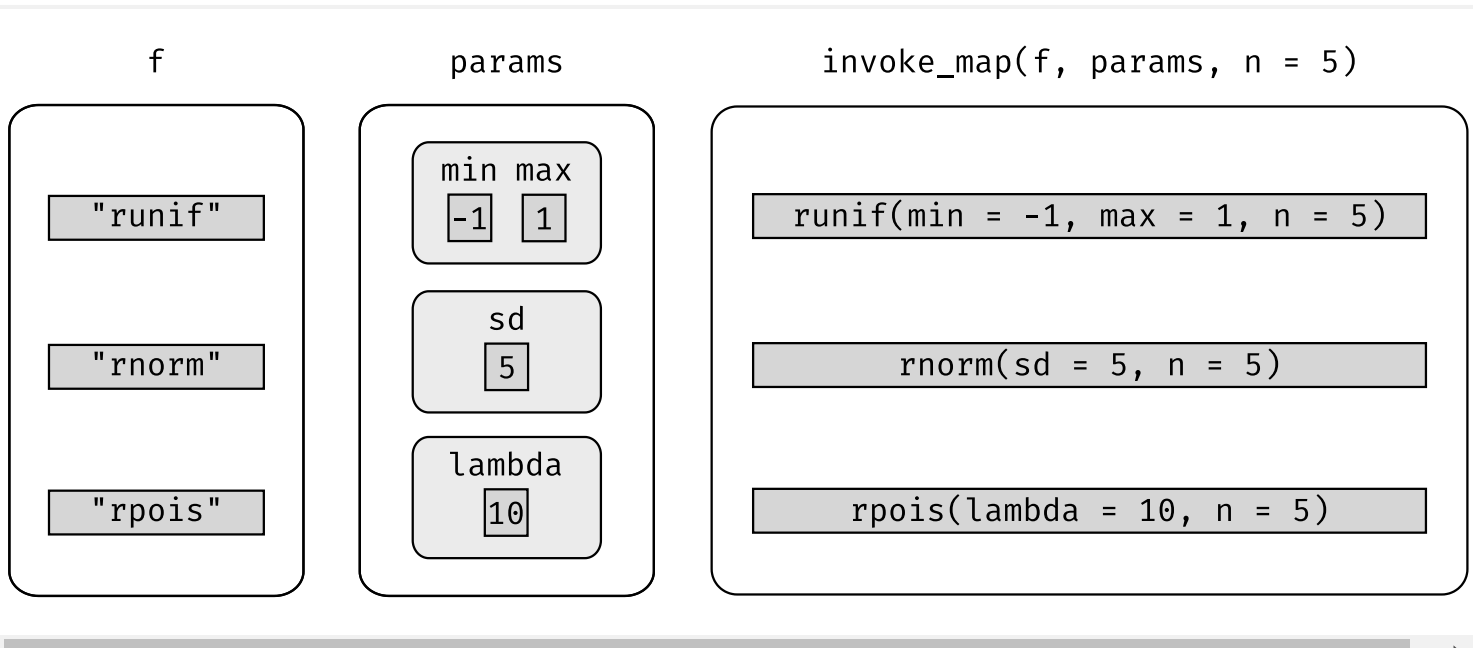
21.7.1 Invocando distintas funciones

Existe un paso adicional en términos de complejidad. Así como cambias los argumentos de la función también puedes cambiar la función misma:

```
f <- c("runif", "rnorm", "rpois")
param <- list(
  list(min = -1, max = 1),
  list(sd = 5),
  list(lambda = 10)
)
```

Para manejar este caso, puedes usar `invoke_map()`:

```
invoke_map(f, param, n = 5) %>% str()
#> List of 3
#>  $ : num [1:5] 0.479 0.439 -0.471 0.348 -0.581
#>  $ : num [1:5] 2.48 3.9 7.54 -9.12 3.94
#>  $ : int [1:5] 6 11 5 8 9
```



El primer argumento es una lista de funciones o un vector de caracteres con nombres de funciones. El segundo argumento es una lista de listas que indica los argumentos que cambian en cada función. Los argumentos subsecuentes pasan a cada función.

Nuevamente, puedes usar `tribble()` para hacer la creación de tuplas coincidentes un poco más fácil:

```
sim <- tribble(
  ~f,      ~params,
  "runif", list(min = -1, max = 1),
  "rnorm", list(sd = 5),
  "rpois", list(lambda = 10)
)
sim %>%
  mutate(sim = invoke_map(f, params, n = 10))
```

21.8 Walk

Walk es una alternativa a *map* que puedes usar cuando quieras llamar a una función por sus efectos colaterales, más que por sobre el valor que devuelve. Típicamente hacemos esto cuando queremos mostrar la salida en pantalla o guardar archivos en el disco. Lo importante es la acción, no el valor resultante. Aquí hay un ejemplo simple:

```
x <- list(1, "a", 3)

x %>%
  walk(print)
#> [1] 1
#> [1] "a"
#> [1] 3
```

Generalmente `walk()` no es tan útil si se compara con `walk2()` o `pwalk()`. Por ejemplo, si tienes una lista de gráficos y un vector con nombres de archivos, puedes usar `pwalk()` para guardar cada archivo en su ubicación correspondiente en el disco:

```
library(ggplot2)
plots <- mtcars %>%
  split(.$cyl) %>%
  map(~ggplot(., aes(mpg, wt)) + geom_point())
paths <- stringr::str_c(names(plots), ".pdf")

pwalk(list(paths, plots), ggsave, path = tempdir())
```

`walk()`, `walk2()` y `pwalk()` devuelven de forma invisible `.x`, el primer argumento. Esto las hace adecuadas para usar dentro de cadenas de pipes.

21.9 Otros patrones para los bucles *for*

Purr entrega algunas funciones que resumen otros tipos de bucles *for*. Si bien los usarás de manera menos frecuente que las funciones map, es útil conocerlas. El objetivo aquí es ilustrar brevemente cada una, con la esperanza de que vengan a tu mente en el futuro cuando veas un problema similar. Luego puedes consultar la documentación para más detalles.

21.9.1 Funciones predicativas

Algunas funciones trabajan con funciones *predicativas* que entregan un único valor `TRUE` o `FALSE`.

`keep()` y `discard()` mantienen los elementos de la entrada donde el predicado es `TRUE` o `FALSE`, respectivamente:

```
flores %>%
  keep(is.factor) %>%
  str()
#> 'data.frame':    150 obs. of  1 variable:
#>  $ Especies: Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...

flores %>%
  discard(is.factor) %>%
  str()
#> 'data.frame':    150 obs. of  4 variables:
#>  $ Largo.Sepalo: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
#>  $ Ancho.Sepalo: num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
#>  $ Largo.Petalo: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
#>  $ Ancho.Petalo: num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
```

`some()` y `every()` determinan si el predicado es verdadero para todos o para algunos de los elementos.

```
x <- list(1:5, letters, list(10))

x %>%
  some(is_character)
#> [1] TRUE

x %>%
  every(is_vector)
#> [1] TRUE
```

`detect()` encuentra el primer elemento donde el predicado es verdadero; `detect_index()` entrega su posición.

```
x <- sample(10)
x
#>  [1] 10  6  1  3  2  4  5  8  9  7

x %>%
  detect(~ . > 5)
#> [1] 10

x %>%
  detect_index(~ . > 5)
#> [1] 1
```

`head_while()` y `tail_while()` toman elementos al inicio y final de un vector cuando el predicado es verdadero:

```
x %>%
  head_while(~ . > 5)
#> [1] 10  6

x %>%
  tail_while(~ . > 5)
#> [1] 8 9 7
```

21.9.2 Reducir y acumular

A veces tendrás una lista compleja que quieres reducir a una lista simple aplicando repetidamente una función que reduce un par a un elemento único. Esto es útil si quieres aplicar un verbo de dos tablas de dplyr a múltiples tablas. Por ejemplo, si tienes una lista de data frames, y quieres reducirla a un unico data frame uniendo los elementos:

```r
dfs <- list(
  age = tibble(name = "John", age = 30),
  sex = tibble(name = c("John", "Mary"), sex = c("M", "F")),
  trt = tibble(name = "Mary", treatment = "A")
)

dfs %>% reduce(full_join)
#> Joining, by = "name"
#> Joining, by = "name"
#> # A tibble: 2 x 4
#>   name    age sex   treatment
#>   <chr> <dbl> <chr> <chr>
#> 1 John     30 M     <NA>
#> 2 Mary     NA F     A
```

O puedes tener una lista de vectores y quieres encontrar la intersección:

```r
vs <- list(
  c(1, 3, 5, 6, 10),
  c(1, 2, 3, 7, 8, 10),
  c(1, 2, 3, 4, 8, 9, 10)
)

vs %>% reduce(intersect)
#> [1]  1  3 10
```

La función `reduce()` (*reducir*) toma una función "binaria" (e.g. una función con dos inputs primarios) y la aplica repetidamente a una lista hasta que quede un solo elemento.

`accumulate()` (*acumular*) es similar, pero mantiene todos los resultados intermedios. Podría usarse para implementar una suma acumulativa:

```r
x <- sample(10)
x
#>  [1]  7  5 10  9  8  3  1  4  2  6
x %>% accumulate(`+`)
#>  [1]  7 12 22 31 39 42 43 47 49 55
```

21.9.3 Ejercicios

1. Implementa tu propia versión de `every()` usando un bucle *for*. Compárala con `purrr::every()`. ¿Qué hace la versión de purrr que la tuya no?

2. Crea una mejora de `col_resumen()` que aplique una función de resumen a cada columna numérica en un data frame.

3. Un posible equivalente de `col_resumen()` es:

```r
col_resumen3 <- function(df, f) {
  is_num <- sapply(df, is.numeric)
  df_num <- df[, is_num]

  sapply(df_num, f)
}
```

Pero tiene una cantidad de bugs que queda ilustrada con las siguientes entradas:

```r
df <- tibble(
  x = 1:3,
  y = 3:1,
  z = c("a", "b", "c")
)
# OK
col_resumen3(df, mean)
# Tiene problemas: no siempre devuelve un vector numérico
col_resumen3(df[1:2], mean)
col_resumen3(df[1], mean)
col_resumen3(df[0], mean)
```

Copy

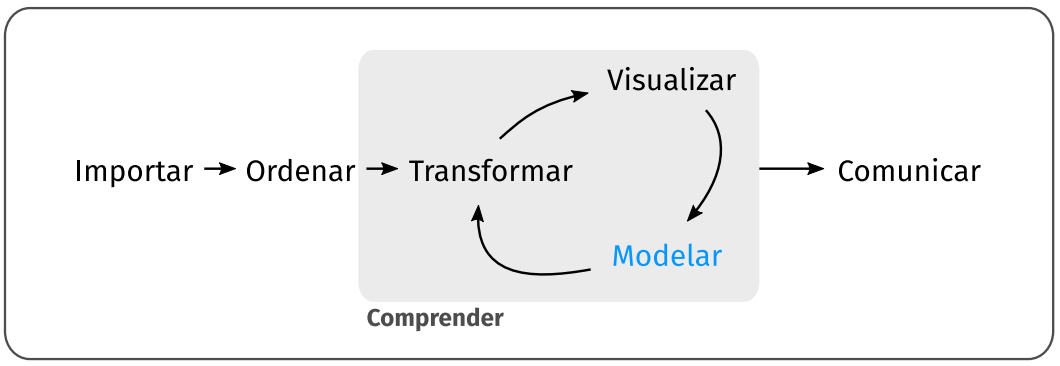
¿Qué causa los *bugs*?

« 20 Vectores

22 Introducción »

22 Introducción

Ahora que estás equipado con herramientas de programación poderosas, finalmente podemos regresar a modelar. Utilizarás tus nuevas herramientas de domador de datos y de programación para ajustar muchos modelos y entender como funcionan. El foco de este libro es la exploración, no la confirmación o la inferencia formal. Pero aprenderás un par de herramientas básicas que te ayudarán a entender la variación en tus modelos.



El objetivo de un modelo es proveer un resumen simple y de baja dimensionalidad sobre un conjunto de datos. Idealmente, el modelo capturará "señales" verdaderas (por ejemplo patrones generados por el fenómeno de interés) e ignorará el "ruido" (es decir, variaciones aleatorias que no nos interesan). Sólo vamos a cubrir modelos "predictivos", los cuales, como su nombre sugiere, generan predicciones. Hay otro tipo de modelo que no vamos a discutir: los modelos de "descubrimiento de datos". Estos modelos no hacen predicciones, si no que ayudan a descubrir relaciones interesantes entre tus datos. (Estos dos tipos de modelos suelen ser llamados supervisados y no supervisados, pero no creo que esa terminología sea particularmente esclarecedora.) Este libro no te dará un entendimiento profundo de la teoría matemática que subyace a los modelos. Sin embargo, construirá tu intuición sobre cómo funcionan los modelos estadísticos y te proporcionará una familia de herramientas útiles que te permitirán utilizar modelos para comprender mejor tus datos:

- En [conceptos básicos], aprenderás cómo los modelos funcionan mecánicamente, centrándonos en la importante familia de modelos lineales. Aprenderás herramientas generales para obtener información sobre lo que un modelo predictivo te dice sobre tus datos, centrándonos en conjuntos de datos simples simulados.

- En construcción de modelos, aprenderás cómo usar modelos para extraer patrones conocidos en datos reales. Una vez que reconozcas un patrón importante, es útil hacerlo explícito en un modelo, porque entonces podrás ver más fácilmente aquellas señales sutiles que quedan en tus datos.

- En muchos modelos, aprenderás cómo usar muchos modelos simples para ayudar a comprender conjuntos de datos complejos. Esta es una técnica poderosa, pero para acceder necesitarás combinar herramientas de modelado y de programación.

Deliberadamente dejamos fuera del capítulo las explicaciones de herramientas para evaluar cuantitativamente los modelos porque dicha evaluación precisa requiere conocer un par de grandes ideas que simplemente no tenemos espacio para cubrir aquí. Por ahora, dependerás de la evaluación cualitativa y de tu escepticismo natural. En [Aprender más sobre los modelos], te indicaremos otros recursos donde podrás seguir aprendiendo.

22.1 Generación de hipótesis vs. confirmación de hipótesis

En este libro vamos a usar los modelos como una herramienta para la exploración, completando la tríada de las herramientas para EDA que se introdujeron en la Parte 1. Los modelos no se suelen enseñar de esta manera, pero como verás, son herramientas importantes para la exploración. Tradicionalmente, el enfoque del modelado está en la inferencia, es decir, para confirmar que una hipótesis es verdadera; hacerlo correctamente no es complicado, pero si difícil. Hay un par de ideas que debes comprender para poder hacer la inferencia correctamente:

On this page

View source
Edit this page

1. Cada observación puede ser utilizada para exploración o para confirmación, pero no para ambas.

2. Puedes usar una observación tantas veces como quieras para la exploración, pero solo una vez para confirmación. En el instante que usaste una observación dos veces, pasaste de confirmar a explorar.

Esto es necesario porque, para confirmar una hipótesis, debes usar datos independientes de los datos utilizados para generarla. De lo contrario, serás demasiado optimista. No hay absolutamente nada de malo en la exploración, pero nunca debes vender un análisis exploratorio como análisis confirmatorio porque es fundamentalmente erróneo.

If you are serious about doing an confirmatory analysis, one approach is to split your data into three pieces before you begin the analysis: Si realmente quieres realizar un análisis confirmatorio, un enfoque es dividir los datos en tres partes antes de comenzar el análisis:

1. El 60% de los datos van a un conjunto de **entrenamiento** (del inglés, *training*) o exploración. Puedes hacer lo que quieras con estos datos, desde visualizarlo a ajustar toneladas de modelos.

2. 20% va a un conjunto de **consulta** (del inglés, *query*). Se puede usar esta información para comparar modelos o hacer visualizaciones a mano, pero no está permitido usarlo como parte de un proceso automatizado.

3. El otro 20% se reserva para un conjunto de **validación** (del inglés, *test*). Sólo se pueden usar estos datos UNA VEZ, para probar tu modelo final.

Esta partición de los datos, te permite explorar con los datos de entrenamiento, generando ocasionalmente hipótesis candidatas que se verifican con el conjunto de consultas. Cuando estés seguro de tener el modelo correcto, se verifica una vez con los datos del conjunto de validación.

(Se debe tener en cuenta que, incluso cuando realice modelado confirmatorio, se necesita hacer EDA. Si no se realiza ninguna EDA, los problemas de calidad que tengan los datos quedarán ocultos).

« 21 Iteración

23 Modelos: conceptos básicos »

23 Modelos: conceptos básicos

23.1 Introducción

El objetivo de un modelo es proveer un resumen de baja dimensión de un conjunto de datos (o *dataset*, en inglés). En el contexto de este libro vamos a usar modelos para particionar los datos en patrones y residuos. Patrones fuertes esconderán tendencias sutiles, por lo que usaremos modelos para remover capas de estructuras mientras exploramos el conjunto de datos.

Sin embargo, antes de que podamos comenzar a usar modelos en conjuntos de datos interesantes y reales, necesitarás los conceptos básicos de cómo funcionan los modelos. Por dicha razón, este capítulo es único porque usa solamente datos simulados. Estos conjunto de datos son muy simples y no muy interesantes, pero nos ayudarán a entender la esencia del modelado antes de que apliques las mismas técnicas con datos reales en el próximo capítulo.

Hay dos partes en un modelo:

1. Primero, defines una **familia de modelos** que expresa un patrón que quieras capturar. El patrón debe ser preciso, pero también genérico. Por ejemplo, el patrón podría ser una línea recta, o una curva cuadrática. Expresarás la familia de modelos con una ecuación como `y = a_1 * x + a_2` o `y = a_1 * x ^ a_2`. Aquí, `x` e `y` son variables conocidas de tus datos, y `a_1` y `a_2` son parámetros que pueden variar al capturar diferentes patrones.

2. Luego, generas un **modelo ajustado** al encontrar un modelo de la familia que sea lo más cercano a tus datos. Esto toma la familia de modelos genérica y la vuelve específica, como `y = 3 * x + 7` o `y = 9 * x ^ 2`.

Es importante enteder que el modelo ajustado es solamente el modelo más cercano a la familia de modelos. Esto implica que tu tienes el "mejor" modelo (de acuerdo a cierto criterio); no implica que tu tienes un buen modelo y ciertamente no implica que ese modelo es "verdadero". George Box lo explica muy bien en su famoso aforismo:

> Todos los modelos están mal, algunos son útiles.

Vale la pena leer el contexto más completo de la cita:

> Ahora sería muy notable si cualquier sistema existente en el mundo real pudiera ser representado exactamente por algún modelo simple. Sin embargo, modelos simples astutamente escogidos a menudo proporcionan aproximaciones notablemente útiles.
>
> Por ejemplo, la ley PV = RT que relaciona la presión P, el volumen V y la temperatura T de un gas "ideal" a través de una constante R no es exactamente verdadera para cualquier gas real, pero frecuentmente provee una aproximación útil y, además, su estructura es informativa ya que proviene de un punto de vista físico del comportamiento de las moléculas de un gas.
>
> Para tal modelo, no hay necesidad de preguntarse "¿Es el modelo verdadero?". Si la "verdad" debe ser la "verdad completa", la respuesta debe ser "No". La única pregunta de interés es "¿Es el modelo esclarecedor y útil?".

El objetivo de un modelo no es descubrir la verdad, sino descubrir una aproximación simple que sea útil.

23.1.1 Prerrequisitos

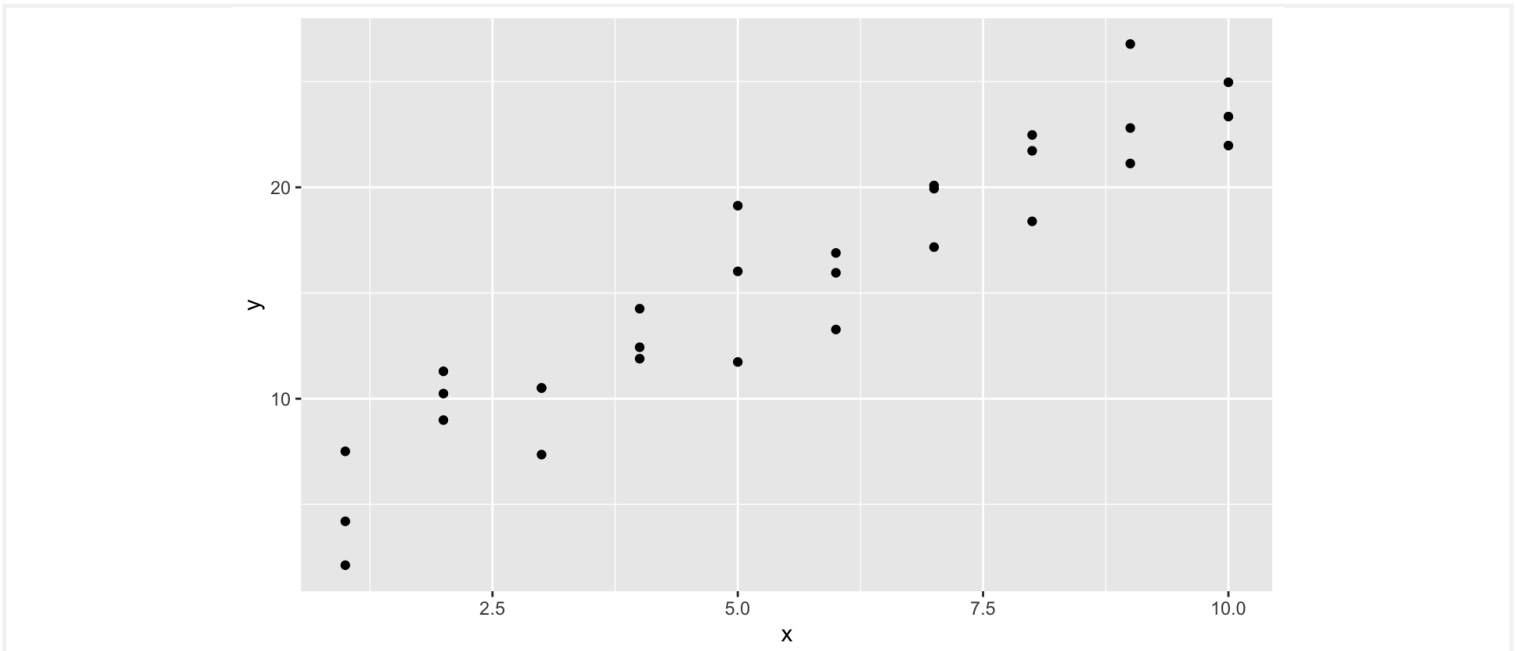
En este capítulo usaremos el paquete **modelr** que encapsula (del inglés *wrapper*) las funciones de modelado de R base para que funcionen naturalmente en un *pipe*.

```
library(tidyverse)

library(modelr)
options(na.action = na.warn)
```

23.2 Un modelo simple

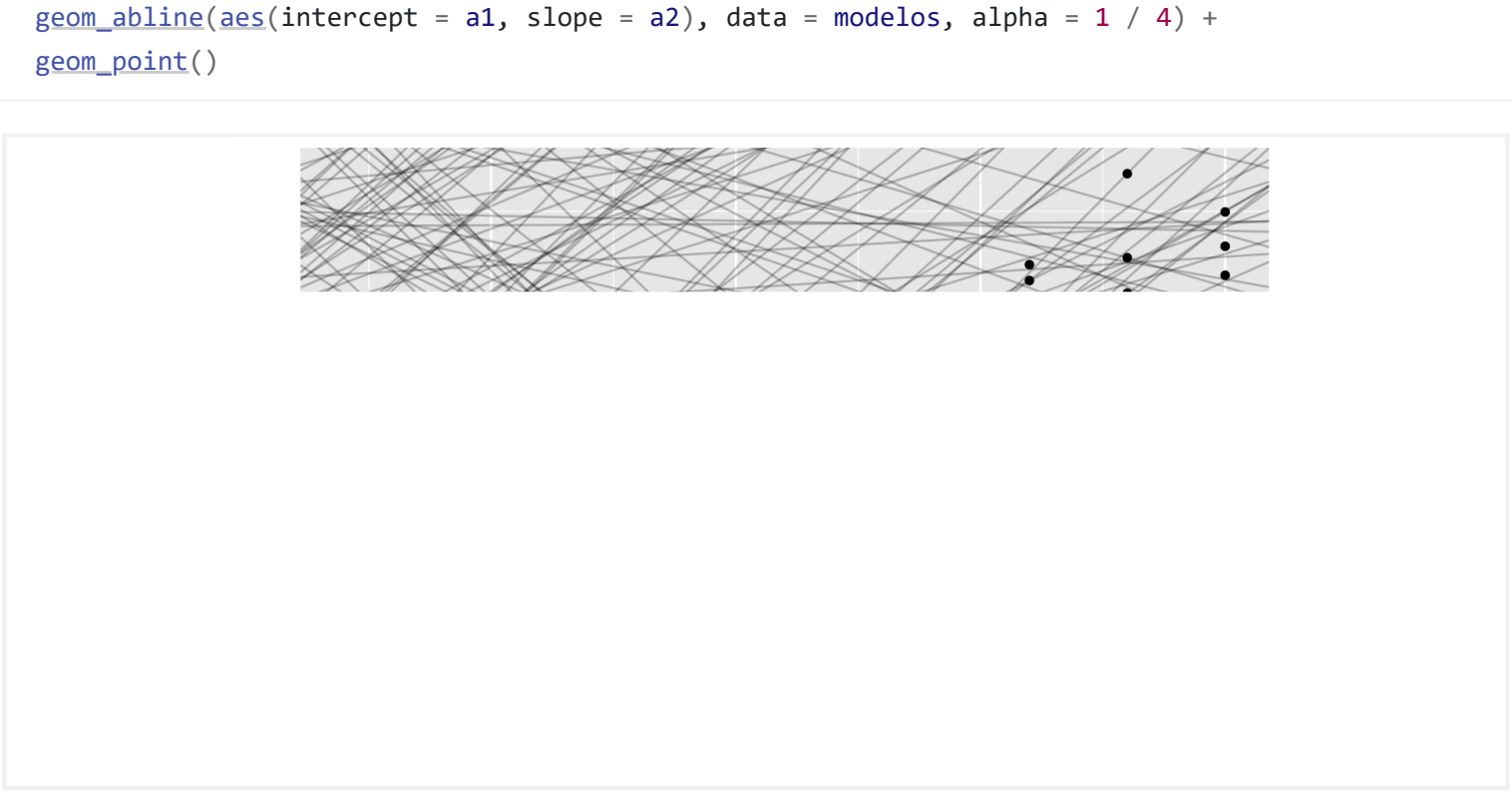
Miremos el conjunto de datos simulado `sim1`, incluído dentro del paquete **modelr**. Este contiene dos variables continuas, `x` e `y`. Grafiquémoslas para ver como están relacionadas:

```
ggplot(sim1, aes(x, y)) +
  geom_point()
```



Puedes ver un fuerte patrón en los datos. Usemos un modelo para capturar dicho patrón y hacerlo explícito. Es nuestro trabajo proporcionar la forma básica del modelo. En este caso, la relación parece ser lineal, es decir: `y = a_0 + a_1 * x`. Comencemos por tener una idea de cómo son los modelos de esa familia generando aleatoriamente unos pocos y superponiéndolos sobre los datos. Para este caso simple, podemos usar `geom_abline()` que toma una pendiente e intercepto (u ordenada al origen) como parámetros. Más adelante, aprenderemos técnicas más generales que funcionan con cualquier modelo.

```
modelos <- tibble(
  a1 = runif(250, -20, 40),
  a2 = runif(250, -5, 5)
)

ggplot(sim1, aes(x, y)) +
  geom_abline(aes(intercept = a1, slope = a2), data = modelos, alpha = 1 / 4) +
  geom_point()
```



Hay 250 modelos en el gráfico, ¡pero muchos son realmente malos! Necesitamos encontrar los modelos buenos especificando nuestra intuición de que un buen modelo está “cerca” de los datos. Necesitamos una manera de cuantificar la distancia entre los datos y un modelo. Entonces podemos ajustar el modelo encontrando el valor de `a_0` y `a_1` que genera el modelo con la menor distancia a estos datos.

Un lugar fácil para comenzar es encontrar la distancia vertical entre cada punto y el modelo, como lo muestra el siguiente diagrama. (Nota que he cambiado ligeramente los valores x para que puedas ver las distancias individuales.)

La distancia es solo la diferencia entre el valor dado por el modelo (la **predicción**), y el valor real y en los datos (la **respuesta**).

Para calcular esta distancia, primero transformamos nuestra familia de modelos en una función de R. Esta función toma los parámetros del modelo y los datos como inputs, y retorna el valor predicho por el modelo como output:

```
model1 <- function(a, data) {
  a[1] + data$x * a[2]
}
model1(c(7, 1.5), sim1)
#>  [1]  8.5  8.5  8.5 10.0 10.0 10.0 11.5 11.5 11.5 13.0 13.0 13.0 14.5 14.5 14.5
#> [16] 16.0 16.0 16.0 17.5 17.5 17.5 19.0 19.0 19.0 20.5 20.5 20.5 22.0 22.0 22.0
```

Luego, necesitaremos calcular la distancia entre lo predicho y los valores reales. En otras palabras, el siguiente gráfico muestra 30 distancias: ¿Cómo las colapsamos en un único número?

Una forma habitual de hacer esto en estadística es usar la "raíz del error cuadrático medio" (del inglés *root-mean-squared deviation*). Calculamos la diferencia entre los valores reales y los predichos, los elevamos al cuadrado, luego se promedian y tomamos la raíz cuadrada. Esta distancia cuenta con propiedades matemáticas interesantes, pero no nos referiremos a ellas en este capítulo. ¡Tendrás que creer en mi palabra!

```
measure_distance <- function(mod, data) {
  diff <- data$y - model1(mod, data)
  sqrt(mean(diff^2))
}
measure_distance(c(7, 1.5), sim1)
#> [1] 2.665212
```

Ahora podemos usar purrr para calcular la distancia de todos los modelos definidos anteriormente. Necesitamos una función auxiliar debido a que nuestra función de distancia espera que el modelo sea un vector numérico de longitud 2.

```
sim1_dist <- function(a1, a2) {
  measure_distance(c(a1, a2), sim1)
}

modelos <- modelos %>%
  mutate(dist = purrr::map2_dbl(a1, a2, sim1_dist))
modelos
#> # A tibble: 250 x 3
#>       a1      a2  dist
#>    <dbl>   <dbl> <dbl>
#> 1 -15.2   0.0889  30.8
#> 2  30.1  -0.827   13.2
#> 3  16.0   2.27    13.2
#> 4 -10.6   1.38    18.7
#> 5 -19.6  -1.04    41.8
#> 6   7.98  4.59    19.3
#> # … with 244 more rows
```

A continuación, vamos a superponer los mejores 10 modelos en los datos. He coloreado los modelos usando `-dist`: esto es una forma fácil de asegurarse de que los mejores modelos (es decir, aquellos con la menor distancia) tengan los colores más brillantes.

```
ggplot(sim1, aes(x, y)) +
  geom_point(size = 2, colour = "grey30") +
  geom_abline(
    aes(intercept = a1, slope = a2, colour = -dist),
    data = filter(modelos, rank(dist) <= 10)
  )
```

También podemos pensar estos modelos como observaciones y visualizar un diagrama de dispersión (o *scatterplot*, en inglés) de `a1` versus `a2`, nuevamente coloreado usando `-dist`. No podremos ver directamente cómo el modelo contrasta con los datos, pero podemos ver muchos modelos a la vez. Nuevamente, he destacado los mejores 10 modelos, esta vez dibujando círculos rojos bajo ellos.

```
ggplot(modelos, aes(a1, a2)) +
  geom_point(data = filter(modelos, rank(dist) <= 10), size = 4, colour = "red") +
  geom_point(aes(colour = -dist))
```

En lugar de probar con múltples modelos aleatorios, se puede sistematizar y generar una cuadrícula de puntos igualmente espaciados (esto se llama búsqueda en cuadrícula). He seleccionado los parámetros de la cuadrícula por aproximación, mirando donde se ubican los mejores modelos en el gráfico anterior.

```
grid <- expand.grid(
  a1 = seq(-5, 20, length = 25),
  a2 = seq(1, 3, length = 25)
) %>%
  mutate(dist = purrr::map2_dbl(a1, a2, sim1_dist))

grid %>%
  ggplot(aes(a1, a2)) +
  geom_point(data = filter(grid, rank(dist) <= 10), size = 4, colour = "red") +
  geom_point(aes(colour = -dist))
```

Cuando superpones los mejores 10 modelos en los datos originales, se ven bastante bien:

```
ggplot(sim1, aes(x, y)) +
  geom_point(size = 2, colour = "grey30") +
  geom_abline(
    aes(intercept = a1, slope = a2, colour = -dist),
    data = filter(grid, rank(dist) <= 10)
  )
```

Podrás imaginarte que de forma iterativa puedo hacer la cuadrícula más y más fina hasta reducir los resultados al mejor modelo. Existe una forma mejor de resolver el problema: una herramienta de minimización llamada búsqueda de Newton-Raphson. La intuición detrás de Newton-Raphson es bastante simple: tomas un punto de partida y buscas la pendiente más fuerte en torno a ese punto. Puedes bajar por esa pendiente un poco, para luego repetir el proceso varias veces, hasta que no se puede descender más. En R, esto se puede hacer con la función `optim()`:

```
best <- optim(c(0, 0), measure_distance, data = sim1)
best$par
#> [1] 4.222248 2.051204

ggplot(sim1, aes(x, y)) +
  geom_point(size = 2, colour = "grey30") +
  geom_abline(intercept = best$par[1], slope = best$par[2])
```

No te preocupes demasiado acerca de los detalles de cómo funciona `optim()`. La intuición es lo importante en esta parte. Si tienes una función que define la mínima distancia entre un modelo y un conjunto de datos, un algoritmo que pueda minimizar la distancia modificando los parámetros del modelo

te permitirá encontrar el mejor modelo. Lo interesante de este enfoque es que funciona con cualquier familia de modelos respecto de la cual se pueda escribir una ecuación que los describa.

Existe otro enfoque que podemos usar para este modelo, debido a que es un caso especial de una familia más amplia: los modelos lineales. Un modelo lineal es de la forma `y = a_1 + a_2 * x_1 + a_3 * x_2 + ... + a_n * x_(n+1)`. Este modelo simple es equivalente a un modelo lineal generalizado en el que n tiene valor 2 y `x_1` es `x`. R cuenta con una herramienta diseñada especialmente para ajustar modelos lineales llamada `lm()`. `lm()` tiene un modo especial de especificar la familia del modelo: las fórmulas. Las fórmulas son similares a `y ~ x`, que `lm()` traducirá a una función de la forma `y = a_1 + a_2 * x`. Podemos ajustar el modelo y mirar la salida:

```
sim1_mod <- lm(y ~ x, data = sim1)
coef(sim1_mod)
#> (Intercept)           x
#>    4.220822    2.051533
```

Copy

¡Estos son exactamente los mismos valores obtenidos con `optim()`! Detrás del escenario `lm()` no usa `optim()`, sin embargo saca ventaja de la estructura matemática de los modelos lineales. Usando algunas conexiones entre geometría, cálculo y álgebra lineal, `lm()` encuentra directamente el mejor modelo en un paso, usando un algoritmo sofisticado. Este enfoque es a la vez rapído y garantiza que existe un mínimo global.

23.2.1 Ejercicios

1. Una desventaja del modelo lineal es ser sensible a valores inusuales debido a que la distancia incorpora un término al cuadrado. Ajusta un modelo a los datos simulados que se presentan a continuación y visualiza los resultados. Corre el modelo varias veces para generar diferentes conjuntos de datos simulados. ¿Qué puedes observar respecto del modelo?

```
sim1a <- tibble(
  x = rep(1:10, each = 3),
  y = x * 1.5 + 6 + rt(length(x), df = 2)
)
```

Copy

1. Una forma de obtener un modelo lineal más robusto es usar una métrica distinta para la distancia. Por ejemplo, en lugar de la raíz de la distancia media cuadrática (del inglés *root-mean-squared distance*) se podría usar la media de la distancia absoluta:

```
measure_distance <- function(mod, data) {
  diff <- data$y - model1(mod, data)
  mean(abs(diff))
}
```

Copy

Usa `optim()` para ajustar este modelo a los datos simulados anteriormente y compara el resultado con el modelo lineal.

1. Un desafío al realizar optimización numérica es que únicamente garantiza encontrar un óptimo local. ¿Qué problema se presenta al optimizar un modelo de tres parámetros como el que se presenta a continuación?

```
model1 <- function(a, data) {
  a[1] + data$x * a[2] + a[3]
}
```

Copy

23.3 Visualizando modelos

Para modelos simples, como el presentado anteriormente, puedes descubrir el patrón que captura el modelo si inspeccionas cuidadosamente la familia del modelo y los coeficientes ajustados. Si alguna vez tomaste un curso de modelado estadístico, te será habitual gastar mucho tiempo en esa tarea. Aquí, sin embargo, tomaremos otro camino. Vamos a enfocarnos en entender un modelo mirando las predicciones que genera. Esto tiene una gran ventaja: cada tipo de modelo predictivo realiza predicciones (¿qué otra podría realizar?) de modo que podemos usar el mismo conjunto de técnicas para entender cualquier tipo de modelo predictivo.

También es útil observar lo que el modelo no captura, los llamados residuos que se obtienen restando las predicciones a los datos. Los residuos son poderosos porque nos permiten usar modelos para quitar patrones fuertes y así observar tendencias sutiles que se mantengan luego de quitar los patrones más evidentes.

23.3.1 **Predicciones**

Para visualizar las predicciones de un modelo, podemos partir por generar una grilla de valores equidistantes que cubra la región donde se encuentran los datos. La forma más fácil de hacerlo es usando `modelr::data_grid()` . El primer argumento es un data frame y, por cada argumento adicional, encuentra las variables únicas y luego genera todas las combinaciones:

```
grid <- sim1 %>%
  data_grid(x)
grid
#> # A tibble: 10 x 1
#>       x
#>   <int>
#> 1     1
#> 2     2
#> 3     3
#> 4     4
#> 5     5
#> 6     6
#> # … with 4 more rows
```

(Esto será más interesante cuando se agreguen más variables al modelo.)

Luego agregamos las predicciones. Usaremos `modelr::add_predictions()` que toma un data frame y un modelo. Esto agrega las predicciones del modelo en una nueva columna en el data frame:

```
grid <- grid %>%
  add_predictions(sim1_mod)
grid
#> # A tibble: 10 x 2
#>       x  pred
#>   <int> <dbl>
#> 1     1  6.27
#> 2     2  8.32
#> 3     3 10.4
#> 4     4 12.4
#> 5     5 14.5
#> 6     6 16.5
#> # … with 4 more rows
```

(También puedes usar esta función para agregar prediccionees al conjunto de datos original.)

A continuación, graficamos las predicciones. Te preguntarás acerca de todo este trabajo adicional en comparación a usar `geom_abline()` . Pero la ventaja de este enfoque es que funciona con *cualquier* modelo en R, desde los más simples a los más complejos. La única limitante son tus habilidades de visualización. Para más ideas respecto de como visualizar modelos complejos, puedes consultar http://vita.had.co.nz/papers/model-vis.html.

```
ggplot(sim1, aes(x)) +
  geom_point(aes(y = y)) +
  geom_line(aes(y = pred), data = grid, colour = "red", size = 1)
```

23.3.2 **Residuos**

La otra cara de las predicciones son los residuos. Las predicciones te informan de los patrones que el modelo captura y los residuos te dicen lo que el modelo ignora. Los residuos son las distancias entre lo observado y los valores predichos que calculamos anteriormente.

Agregamos los residuos a los datos con `add_residuals()`, que funciona de manera similar a `add_predictions()`. Nota, sin embargo, que usamos el data frame original y no no una grilla manufacturada. Esto es porque para calcular los residuos se necesitan los valores de "y".

```
sim1 <- sim1 %>%
  add_residuals(sim1_mod)
sim1
#> # A tibble: 30 x 3
#>       x     y  resid
#>   <int> <dbl>  <dbl>
#> 1     1  4.20 -2.07
#> 2     1  7.51  1.24
#> 3     1  2.13 -4.15
#> 4     2  8.99  0.665
#> 5     2 10.2   1.92
#> 6     2 11.3   2.97
#> # … with 24 more rows
```

Existen diferentes formas de entender qué nos dicen los residuos respecto del modelo. Una forma es dibujar un polígono de frecuencia que nos ayude a entender cómo se propagan los residuos:

```
ggplot(sim1, aes(resid)) +
  geom_freqpoly(binwidth = 0.5)
```

Esto ayuda a calibrar la calidad del modelo: ¿qué tan lejos se encuentran las predicciones de los valores observados? Nota que el promedio del residuo es siempre cero.

A menudo vas a querer crear gráficos usando los residuos en lugar del predictor original. Verás mucho de eso en el capítulo siguiente:

```
ggplot(sim1, aes(x, resid)) +
  geom_ref_line(h = 0) +
  geom_point()
```

Esto parece ser ruido aleatorio, lo que sugiere que el modelo ha hecho un buen trabajo capturando los patrones del conjunto de datos.

23.3.3 Ejercicios

1. En lugar de usar `lm()` para ajustar una línea recta, puedes usar `loess()` para ajustar una curva suave. Repite el proceso de ajustar el modelo, generar la cuadrícula, predicciones y visualización con `sim1` usando `loess()` en vez de `lm()`. ¿Cómo se compara el resultado a `geom_smooth()`.

2. `add_predictions()` está pareada con `gather_predictions()` y `spread_predictions()`. ¿Cómo difieren estas tres funciones?

3. ¿Qué hace `geom_ref_line()`? ¿De qué paquete proviene? ¿Por qué es útil e importante incluir una línea de referencia en los gráficos que muestran residuos?

4. ¿Por qué quisieras mirar un polígono de frecuencias con los residuos absolutos? ¿Cuáles son las ventajas y desventajas de los residuos crudos?

23.4 Fórmulas y familias de modelos

Ya habrás visto fórmulas anteriormente cuando usamos `facet_wrap()` y `facet_grid()`. En R, las fórmulas proveen un modo general de obtener "comportamientos especiales". En lugar de evaluar los valores de las variables directamente, se capturan los valores para que sean interpretados por una función.

La mayoría de los las funciones de modelado en R usan una conversión estándar para las fórmulas y las funciones. Ya habrás visto una conversión simple `y ~ x` que se convierte en `y = a_1 + a_2 * x`. Si quieres ver lo que hace R, puedes usar la función `model_matrix()`. Esta toma un data frame y una fórmula

para entregar un tibble que define la ecuación del modelo: cada columna en la salida está asociada con un coeficiente del modelo, la función es siempre `y = a_1 * salida_1 + a_2 * salida_2`. Para el caso simple `y ~ x1` esto nos muestra algo interesante:

```
df <- tribble(
  ~y, ~x1, ~x2,
  4, 2, 5,
  5, 1, 6
)
model_matrix(df, y ~ x1)
#> # A tibble: 2 x 2
#>   `(Intercept)`    x1
#>           <dbl> <dbl>
#> 1             1     2
#> 2             1     1
```

Copy

La forma en que R agrega el intercepto (u ordenada al origen) al modelo es mediante una columna de unos. Por defecto, R siempre agregará esta columna. Si no quieres esto, necesitas excluirla explícitamente usando `-1`:

```
model_matrix(df, y ~ x1 - 1)
#> # A tibble: 2 x 1
#>      x1
#>   <dbl>
#> 1     2
#> 2     1
```

Copy

La matriz del modelo crece de manera nada sorprendente si incluyes más variables al modelo:

```
model_matrix(df, y ~ x1 + x2)
#> # A tibble: 2 x 3
#>   `(Intercept)`    x1    x2
#>           <dbl> <dbl> <dbl>
#> 1             1     2     5
#> 2             1     1     6
```

Copy

Esta notación para las fórmulas a veces se le llama "notación de Wilkinson-Rogers", la cual fue descrita inicialmente en *Symbolic Description of Factorial Models for Analysis of Variance*, escrito por G. N. Wilkinson y C. E. Rogers https://www.jstor.org/stable/2346786. Es conveniente excavar un poco y leer el artículo original si quieres entender los detalles del álgebra de modelado.

Las siguientes secciones detallan cómo esta notación de fórmulas funciona con variables categóricas, interacciones y transformaciones.

23.4.1 Variables categóricas

Generar una función a partir de una fórmula es directo cuando el predictor es una variable continua, pero las cosas son más complicadas cuando el predictor es una variable categórica. Imagina que tienes una fórmula como `y ~ sexo`, donde el sexo puede ser hombre o mujer. No tiene sentido convertir a una fórmula del tipo `y = x_0 + x_1 * sexo` debido a que `sexo` no es un número - ¡no se puede multiplicar! En su lugar, lo que R hace es convertir a `y = x_0 + x_1 * sexo_hombre` donde `sexo_hombre` tiene valor 1 si `sexo` corresponde a hombre y cero a mujer:

```r
df <- tribble(
  ~genero, ~respuesta,
  "masculino", 1,
  "femenino", 2,
  "masculino", 1
)
model_matrix(df, respuesta ~ genero)
#> # A tibble: 3 x 2
#>   `(Intercept)` generomasculino
#>           <dbl>           <dbl>
#> 1             1               1
#> 2             1               0
#> 3             1               1
```

Quizá te preguntes por qué R no crea la columna `generofemenino`. El problema es que eso crearía una columna perfectamente predecible a partir de las otras columnas (es decir, `generofemenino = 1 - generomasculino`). Desafortunadamete los detalles exactos de por qué esto es un problema van más allá del alcance del libro, pero básicamente crea una familia de modelos que es muy flexible y genera infinitos modelos igualmente cercanos a los datos.

Afortunadamente, sin embargo, si te enfocas en visualizar las predicciones no necesitas preocuparte de la parametrización exacta. Veamos algunos datos y modelos para hacer algo concreto. Aquí está el dataset `sim2` de modelr:

```r
ggplot(sim2) +
  geom_point(aes(x, y))
```

Podemos ajustar un modelo a esto y generar predicciones:

```r
mod2 <- lm(y ~ x, data = sim2)

grid <- sim2 %>%
  data_grid(x) %>%
  add_predictions(mod2)
grid
#> # A tibble: 4 x 2
#>   x      pred
#>   <chr> <dbl>
#> 1 a      1.15
#> 2 b      8.12
#> 3 c      6.13
#> 4 d      1.91
```

Efectivamente, un modelo con una variable `x` categórica va a predecir el valor medio para cada categoría. (¿Por qué? porque la media minimiza la raíz de la distancia media cuadrática.) Es fácil de ver si superponemos la predicción sobre los datos originales:

```r
ggplot(sim2, aes(x)) +
  geom_point(aes(y = y)) +
  geom_point(data = grid, aes(y = pred), colour = "red", size = 4)
```

No es posible hacer predicciones sobre niveles no observados. A veces harás esto por accidente, por lo que es bueno reconocer el siguiente mensaje de error:

```r
tibble(x = "e") %>%
  add_predictions(mod2)
#> Error in model.frame.default(Terms, newdata, na.action = na.action, xlev = object$xlevels):
factor x has new level e
```

23.4.2 Interacciones (continuas y categóricas)

¿Qué ocurre si combinas una variable continua y una categórica? `sim3` contiene un predictor categórico y otro predictor continuo. Podemos visualizarlos con un gráfico simple:

```
ggplot(sim3, aes(x1, y)) +
  geom_point(aes(colour = x2))
```

Copy

Existen dos posibles modelos que se pueden ajustar a estos datos:

```
mod1 <- lm(y ~ x1 + x2, data = sim3)
mod2 <- lm(y ~ x1 * x2, data = sim3)
```

Copy

Cuando agregas variables con `+` el modelo va a estimar cada efecto independientemente de los demás. Es posible agregar al ajuste lo que se conoce como interacción usando `*`. Por ejemplo, `y ~ x1 * x2` se traduce en `y = a_0 + a_1 * x_1 + a_2 * x_2 + a_12 * x_1 * x_2`. Observa que si usas `*`, tanto el efecto interacción como los efectos individuales se incluyen en el modelo.

Para visualizar estos modelos necesitamos dos nuevos trucos:

1. Tenemos dos predictores, por lo que necesitamos pasar ambas variables a `data_grid()`. Esto encontrará todos los valores únicos de `x1` y `x2` y luego generará todas las combinaciones,

2. Para generar predicciones de ambos modelos simultáneamente, podemos usar `gather_predictions()` que incorpora cada predicción como una fila. El complemento de `gather_predictions()` es `spread_predictions()` que incluye cada predicción en una nueva columna.

Esto combinado nos da:

```
grid <- sim3 %>%
  data_grid(x1, x2) %>%
  gather_predictions(mod1, mod2)
grid
#> # A tibble: 80 x 4
#>   model    x1 x2     pred
#>   <chr> <int> <fct> <dbl>
#> 1 mod1      1 a      1.67
#> 2 mod1      1 b      4.56
#> 3 mod1      1 c      6.48
#> 4 mod1      1 d      4.03
#> 5 mod1      2 a      1.48
#> 6 mod1      2 b      4.37
#> # … with 74 more rows
```

Copy

Podemos visualizar los resultados de ambos modelos en un gráfico usando separando en facetas:

```
ggplot(sim3, aes(x1, y, colour = x2)) +
  geom_point() +
  geom_line(data = grid, aes(y = pred)) +
  facet_wrap(~model)
```

Copy

Observa que el modelo que usa `+` tiene la misma pendiente para cada recta, pero distintos interceptos (u ordenadas al origen). El modelo que usa `*` tiene distinta pendiente e intercepto.

¿Qué modelo es el más adecuado para los datos? Podemos mirar los residuos. Aquí hemos separado facetas por modelo y por `x2` ya que facilita ver el patrón dentro de cada grupo.

```
sim3 <- sim3 %>%
  gather_residuals(mod1, mod2)

ggplot(sim3, aes(x1, resid, colour = x2)) +
  geom_point() +
  facet_grid(model ~ x2)
```

Copy

Existe un patrón poco obvio en los residuos de `mod2`. Los residuos de `mod1` muestran que el modelo tiene algunos patrones ignorados en `b`, y un poco menos ignorados en `c` y `d`. Quizá te preguntas si existe una forma precisa de determinar si acaso `mod1` o `mod2` es mejor. Existe, pero requiere un respaldo matemático

fuerte y no nos preocuparemos de eso. Lo que aquí interesa es evaluar cualitativamente si el modelo ha capturado los patrones que nos interesan.

23.4.3 **Interacciones (dos variables continuas)**

Demos un vistazo al modelo equivalente para dos variables continuas. Para comenzar, se procede de igual modo que el ejemplo anterior:

```
mod1 <- lm(y ~ x1 + x2, data = sim4)
mod2 <- lm(y ~ x1 * x2, data = sim4)

grid <- sim4 %>%
  data_grid(
    x1 = seq_range(x1, 5),
    x2 = seq_range(x2, 5)
  ) %>%
  gather_predictions(mod1, mod2)
grid
#> # A tibble: 50 x 4
#>   model    x1    x2   pred
#>   <chr> <dbl> <dbl>  <dbl>
#> 1 mod1   -1    -1    0.996
#> 2 mod1   -1    -0.5 -0.395
#> 3 mod1   -1     0   -1.79
#> 4 mod1   -1     0.5 -3.18
#> 5 mod1   -1     1   -4.57
#> 6 mod1   -0.5  -1    1.91
#> # … with 44 more rows
```

Copy

Observa el uso de `seq_range()` dentro de `data_grid()`. En lugar de usar cada valor único de `x`, usamos una cuadrícula uniformemente espaciada de cinco valores entre los números mínimo y máximo. Quizá no es lo más importante aquí, pero es una técnica útil en general. Existen otros dos argumentos en `seq_range()`:

- `pretty = TRUE` generará una secuencia "bonita", por ejemplo algo agradable al ojo humano. Esto es útil si quieres generar tablas a partir del output:

```
seq_range(c(0.0123, 0.923423), n = 5)
#> [1] 0.0123000 0.2400808 0.4678615 0.6956423 0.9234230
seq_range(c(0.0123, 0.923423), n = 5, pretty = TRUE)
#> [1] 0.0 0.2 0.4 0.6 0.8 1.0
```

Copy

- `trim = 0.1` eliminará el 10% de los valores en el extremo de la cola. Esto es útil si las variables tienen una distribución con una cola larga y te quieres enfocar en generar valores cerca del centro:

```
x1 <- rcauchy(100)
seq_range(x1, n = 5)
#> [1] -115.86934  -83.52130  -51.17325  -18.82520   13.52284
seq_range(x1, n = 5, trim = 0.10)
#> [1] -13.841101  -8.709812  -3.578522   1.552767   6.684057
seq_range(x1, n = 5, trim = 0.25)
#> [1] -2.17345439 -1.05938856  0.05467728  1.16874312  2.28280896
seq_range(x1, n = 5, trim = 0.50)
#> [1] -0.7249565 -0.2677888  0.1893788  0.6465465  1.1037141
```

Copy

- `expand = 0.1` es en cierta medida el opuesto de `trim()` ya que expande el rango en un 10%.

```
x2 <- c(0, 1)
seq_range(x2, n = 5)
#> [1] 0.00 0.25 0.50 0.75 1.00
seq_range(x2, n = 5, expand = 0.10)
#> [1] -0.050  0.225  0.500  0.775  1.050
seq_range(x2, n = 5, expand = 0.25)
#> [1] -0.1250  0.1875  0.5000  0.8125  1.1250
seq_range(x2, n = 5, expand = 0.50)
#> [1] -0.250  0.125  0.500  0.875  1.250
```

Copy

A continuación intentemos visualizar el modelo. Tenemos dos predictores continuos, por lo que te imaginarás el modelo como una superficie 3d. Podemos mostrar esto usando `geom_tile()` :

```
ggplot(grid, aes(x1, x2)) +
  geom_tile(aes(fill = pred)) +
  facet_wrap(~model)
```

Copy

¡Esto no sugiere que los modelos sean muy distintos! Pero eso es en parte una ilusión: nuestros ojos y cerebros no son muy buenos en comparar sombras de color de forma adecuada. En lugar de mirar la superficie desde arriba, podríamos mirarla desde los costados, mostrando múltiples cortes:

```
ggplot(grid, aes(x1, pred, colour = x2, group = x2)) +
  geom_line() +
  facet_wrap(~model)
ggplot(grid, aes(x2, pred, colour = x1, group = x1)) +
  geom_line() +
  facet_wrap(~model)
```

Copy

Esto muestra la interacción entre dos variables continuas que básicamente opera del mismo modo que una variable continua y una categórica. Una interacción dice que no existe un resultado fijo: necesitas considerar los valores de `x1` y `x2` simultáneamente para predecir `y` .

Podrás ver que con apenas dos variables continuas, obtener un buen resultado de visualización es difícil. Pero esto es razonable: ¡no deberías esperar que con tres o más variables sea más fácil entender la interacción! Nuevamente, estamos parcialmente a salvo porque estamos usando modelos para la exploración y crearás tus propios modelos en el tiempo. El modelo no debe ser perfecto, tiene que ayudar a revelar información acerca de los datos.

Pasé un tiempo mirando los residuos para ver si acaso `mod2` es mejor que `mod1` . Creo que lo es, pero es algo sutil. Tendrás la oportunidad de explorar esto en los ejercicios.

23.4.4 **Transformaciones**

Puedes hacer transformaciones dentro de la fórmula del modelo. Por ejemplo `log(y) ~ sqrt(x1) + x2` se transforma en `log(y) = a_1 + a_2 * sqrt(x1) + a_3 * x2` . Si la transformación involucra `+` , `*` , `^` o `-` , necesitas dejar eso dentro de `I()` para que R no lo tome como parte de la especificación del modelo. Por ejemplo, `y ~ x + I(x ^ 2)` se traduce en `y = a_1 + a_2 * x + a_3 * x^2` . Si olvidas incluir `I()` y especificas `y ~ x ^ 2 + x` , R va a calcular `y ~ x * x + x` . `x * x` lo que resulta en la interacción de `x` consigo misma, lo que se reduce a `x` . R automáticamente elimina las variables redundantes, por lo que `x + x` se convierte en `x` , lo que significa que `y ~ x ^ 2 + x` especifica la función `y = a_1 + a_2 * x` . ¡Eso probablemente no es lo que querías!

Nuevamente, si te confunde lo que el modelo hace, puedes usar `model_matrix()` para ver exactamente lo que la ecuación `lm()` está ajustando:

```
df <- tribble(
  ~y, ~x,
  1, 1,
  2, 2,
  3, 3
)
model_matrix(df, y ~ x^2 + x)
#> # A tibble: 3 x 2
#>   `(Intercept)`     x
#>           <dbl> <dbl>
#> 1             1     1
#> 2             1     2
#> 3             1     3
model_matrix(df, y ~ I(x^2) + x)
#> # A tibble: 3 x 3
#>   `(Intercept)` `I(x^2)`     x
#>           <dbl>    <dbl> <dbl>
#> 1             1        1     1
#> 2             1        4     2
#> 3             1        9     3
```

Las transformaciones son útiles porque puedes aproximar funciones no lineales. Si tuviste clases de cálculo, habrás escuchado acerca del teorema de Taylor que dice que puedes aproximar una función suave como la suma de infinitos polinomios. Esto significa que puedes usar una función polinomial para acercarte a una distancia arbitrariamente pequeña de una función suave ajustando una ecuación como `y = a_1 + a_2 * x + a_3 * x^2 + a_4 * x^3`. Escribir esta secuencia a mano es tedioso, pero R provee la función auxiliar `poly()`:

```
model_matrix(df, y ~ poly(x, 2))
#> # A tibble: 3 x 3
#>   `(Intercept)` `poly(x, 2)1` `poly(x, 2)2`
#>           <dbl>         <dbl>         <dbl>
#> 1             1     -7.07e- 1         0.408
#> 2             1     -7.85e-17        -0.816
#> 3             1      7.07e- 1         0.408
```

Sin embargo, existe un problema mayor al usar `poly()`: fuera del rango de los datos, los polinomios rápidamente se disparan a infinito positivo o negativo. Una alternativa más segura es usar spline natural, `splines::ns()`.

```
library(splines)
model_matrix(df, y ~ ns(x, 2))
#> # A tibble: 3 x 3
#>   `(Intercept)` `ns(x, 2)1` `ns(x, 2)2`
#>           <dbl>       <dbl>       <dbl>
#> 1             1       0           0
#> 2             1       0.566      -0.211
#> 3             1       0.344       0.771
```

Veamos esto cuando intentamos aproximar una función no lineal:

```
sim5 <- tibble(
  x = seq(0, 3.5 * pi, length = 50),
  y = 4 * sin(x) + rnorm(length(x))
)

ggplot(sim5, aes(x, y)) +
  geom_point()
```

Voy a ajustar cinco modelos a los datos.

```
mod1 <- lm(y ~ splines::ns(x, 1), data = sim5)
mod2 <- lm(y ~ splines::ns(x, 2), data = sim5)
mod3 <- lm(y ~ splines::ns(x, 3), data = sim5)
mod4 <- lm(y ~ splines::ns(x, 4), data = sim5)
mod5 <- lm(y ~ splines::ns(x, 5), data = sim5)

grid <- sim5 %>%
  data_grid(x = seq_range(x, n = 50, expand = 0.1)) %>%
  gather_predictions(mod1, mod2, mod3, mod4, mod5, .pred = "y")

ggplot(sim5, aes(x, y)) +
  geom_point() +
  geom_line(data = grid, colour = "red") +
  facet_wrap(~model)
```

Copy

Observa que la extrapolación fuera del rango de los datos es claramente mala. Esta es la desventaja de aproximar una función mediante un polinomio. Pero este es un problema real con cualquier modelo: el modelo nunca te dirá si el comportamiento es verdadero cuando extrapolas fuera del rango de los datos que has observado. Deberás apoyarte en la teoría y la ciencia.

23.4.5 Ejercicios

1. ¿Qué pasa si repites el análisis de `sim2` usando un modelo sin intercepto? ¿Qué ocurre con la ecuación del modelo? ¿Qué ocurre con las predicciones?

2. Usa `model_matrix()` para explorar las ecuaciones generadas por los modelos ajustados a `sim3` y `sim4`. ¿Por qué `*` es un atajo para la interacción?

3. Usando los principios básicos, convierte las fórmulas de los siguientes modelos en funciones. (Sugerencia: comienza por convertir las variables categóricas en ceros y unos.)

```
mod1 <- lm(y ~ x1 + x2, data = sim3)
mod2 <- lm(y ~ x1 * x2, data = sim3)
```

Copy

1. Para `sim4`, ¿Es mejor `mod1` o `mod2`? Yo creo que `mod2` es ligeramente mejor removiendo las tendencias, pero es bastante sutil. ¿Puedes generar un gráfico que de sustento a esta hipótesis?

23.5 Valores faltantes

Los valores faltantes obviamente no proporcionan información respecto de la relación entre las variables, por lo que modelar funciones va a eliminar todas las filas con valores faltantes. R por defecto lo hace de forma silenciosa, pero `options(na.action = na.warn)` (ejecutado en los prerrequisitos) asegura que la salida incluya una advertencia.

```
df <- tribble(
  ~x, ~y,
  1, 2.2,
  2, NA,
  3, 3.5,
  4, 8.3,
  NA, 10
)

mod <- lm(y ~ x, data = df)
#> Warning: Dropping 2 rows with missing values
```

Copy

Para suprimir los mensajes de advertencia, incluye `na.action = na.exclude`:

```
mod <- lm(y ~ x, data = df, na.action = na.exclude)
```

Copy

Siempre puedes consultar cuántas observaciones se usaron con `nobs()`:

```
nobs(mod)
#> [1] 3
```

Copy

23.6 Otras familias de modelos

Este capítulo se centró de forma exclusiva en la familia de modelos lineales, la cual asume una relación de la forma `y = a_1 * x1 + a_2 * x2 + ... + a_n * xn`. Además, los modelos lineales asumen que los residuos siguen una distribución normal, algo de lo que no hemos hablado. Existe un amplio conjunto de familias de modelos que extienden la familia de modelos lineales de varias formas interesantes. Algunos son:

- **Modelos lineales generalizados**, es decir, `stats::glm()`. Los modelos lineales asumen que la respuesta es una variable continua y que el error sigue una distribución normal. Los modelos lineales generalizados extienden los modelos lineales para incluir respuestas no continuas (es decir, datos binarios o conteos). Definen una distancia métrica basada en la idea estadística de verosimilitud.

- **Modelos generalizados aditivos**, es decir, `mgcv::gam()`, extienden los modelos lineales generalizados para incorporar funciones suaves arbitrarias. Esto significa que puedes escribir una fórmula del tipo `y ~ s(x)` que se transforma en una ecuación de la forma `y = f(x)` y dejar que `gam()` estime la función (sujeto a algunas restricciones de suavidad para que el problema sea manejable).

- **Modelos lineales penalizados**, es decir, `glmnet::glmnet()`, incorporan un término de penalización a la distancia y así penalizan modelos complejos (definidos por la distancia entre el vector de parámetros y el origen). Esto tiende a entregar modelos que generalizan mejor respecto de nuevos conjuntos de datos para la misma población.

- **Modelos lineales robustos**, es decir, `MASS:rlm()`, modifican la distancia para restar importancia a los puntos que quedan muy alejados. Esto resulta en modelos menos sensibles a valores extremos, con el inconveniente de que no son muy buenos cuando no hay valores extremos.

- **Árboles**, es decir, `rpart::rpart()`, atacan un problema de un modo totalmente distinto a los modelos lineales. Ajustan un modelo constante por partes, dividiendo los datos en partes progresivamente más y más pequeñas. Los árboles no son tremendamente efectivos por sí solos, pero son muy poderosos cuando se usan en modelos agregados como **bosques aleatorios**, del inglés *random forests*), (es decir, `randomForest::randomForest()`) o **máquinas aceleradoras de gradiente**, del inglés *gradient boosting machines* (es decir, `xgboost::xgboost`.).

Estos modelos son todos similares desde una perspectiva de programación. Una vez que hayas manejado los modelos lineales, te resultará sencillo entender la mecánica de otras clases de modelos. Ser un modelador hábil consiste en tener buenos principios generales y una gran caja de herramientas técnicas. Ahora que has aprendido algunas herramientas y algunas clases de modelos, puedes continuar aprendiendo sobre otras clases en otras fuentes.

« 22 Introducción

24 Construcción de modelos »

24 Construcción de modelos

24.1 Introducción

En el capítulo previo aprendimos cómo funcionan los modelos lineales, y aprendimos algunas herramientas básicas para entender lo que un modelo está mostrando con sus datos. El capítulo previo se enfocó en simular conjunto de datos. Este capítulo se centrará en datos reales, mostrando como puedes progresivamente construir un modelo que te ayude a entender los datos.

Tomaremos ventaja del hecho que se puede pensar que un modelo particiona tus datos en patrones y residuos. Encontraremos patrones con visualizaciones, luego los haremos concretos y precisos con un modelo. Repetiremos luego el proceso, pero reemplazaremos la variable antigua con los residuos del modelo. El objetivo es pasar de un conocimiento implícito en la data a un conocimiento explícito en un modelo cuantitativo. Esto hace que sea más fácil aplicar nuevos dominios, y más fácil de usar para otros.

Para un conjunto de datos muy grande y complejo esto será mucho trabajo. Sin duda hay enfoques alternativos - un enfoque de aprendizaje automático es simplemente enfocarse en la capacidad predictiva del modelo. Ese enfoque tiende a producir cajas negras: el modelo hace muy bien su trabajo generando predicciones, pero no sabes por qué. Esto es un enfoque totalmente razonable, pero es difícil de aplicar el conocimiento del mundo real al modelo. Eso, a su vez, hace difícil evaluar si el modelo continuará o no funcionando a largo plazo, ya que los fundamentos cambian. Para la mayoría de los modelos, esperaría que usaras alguna combinación de este enfoque y un enfoque clásico automatizado.

Es un desafio saber cuando detenerse. Necesitas darte cuenta cuando tu modelo es lo suficientemente bueno, y cuando no es conveniente invertir mas tiempo en él. Me gusta especialmente esta cita del usuario de reddit Broseidon241:

> Hace mucho tiempo en clase de arte, mi profesor me dijo “Un artista necesita saber cuándo una pieza está terminada. No puedes retocar algo a la perfección - termínalo. Si no te gusta, hazlo otra vez. O sino empieza algo nuevo”. En años posteriores, yo escuché “Una pobre costurera comete muchos errores. Una buena costurera trabaja duro para corregir esos errores. Una grandiosa costurera no tiene miedo de tirar la prenda y empezar nuevamente”.

– Broseidon241, https://www.reddit.com/r/datascience/comments/4irajq

24.1.1 Prerrequisitos

Usaremos las mismas herramientas que en el capítulo anterior, pero agregaremos algunos conjuntos de datos reales: `diamantes` y `vuelos` del paquete datos También necesitaremos lubridate para trabajar con fechas/horas en `vuelos` .

```
library(tidyverse)
library(modelr)
library(lubridate)
library(datos)
options(na.action = na.warn)
```

Copy

24.2 ¿Por qué los diamantes de baja calidad son más caros?

En el capítulo anterior vimos una sorprendente relación entre la calidad de los diamantes y su precio: diamantes de baja calidad (cortes pobres, colores malos, y claridad inferior) tienen más altos precios.

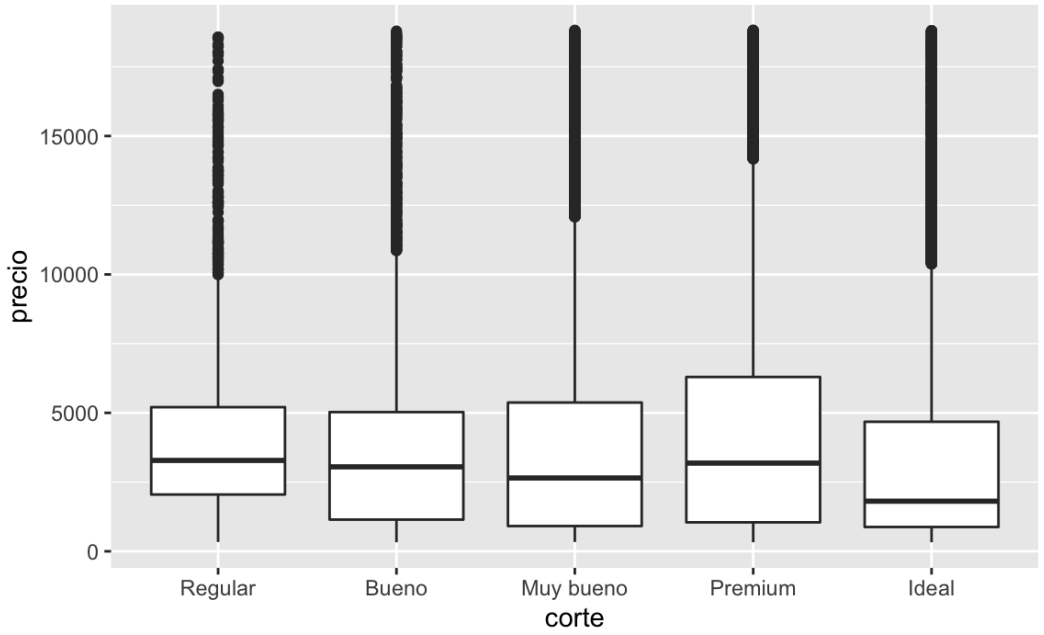
```
ggplot(diamantes, aes(corte, precio)) + geom_boxplot()
ggplot(diamantes, aes(color, precio)) + geom_boxplot()
ggplot(diamantes, aes(claridad, precio)) + geom_boxplot()
```

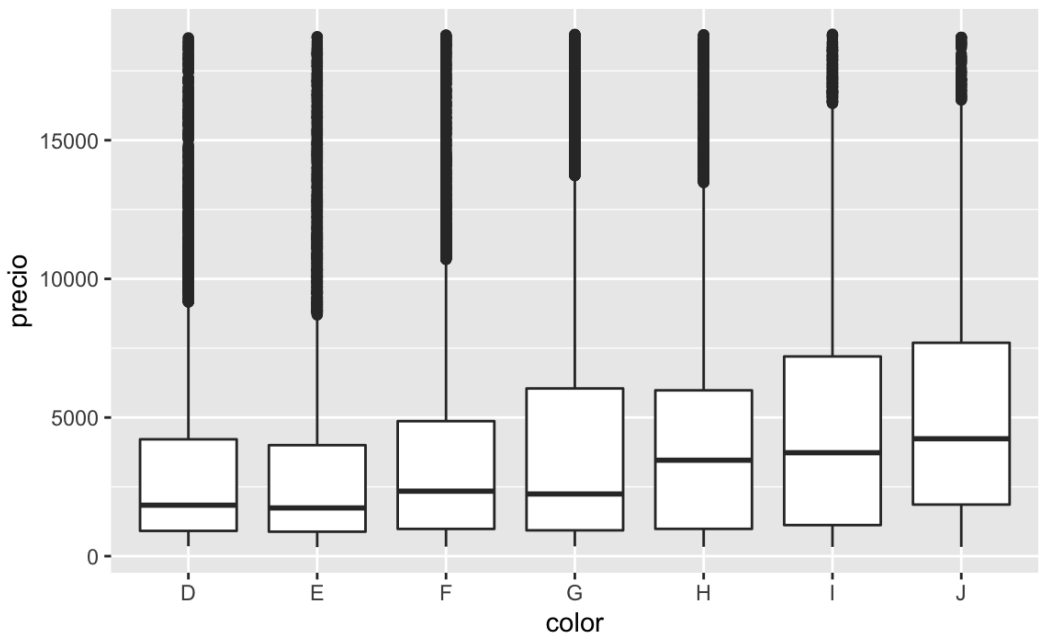
Copy

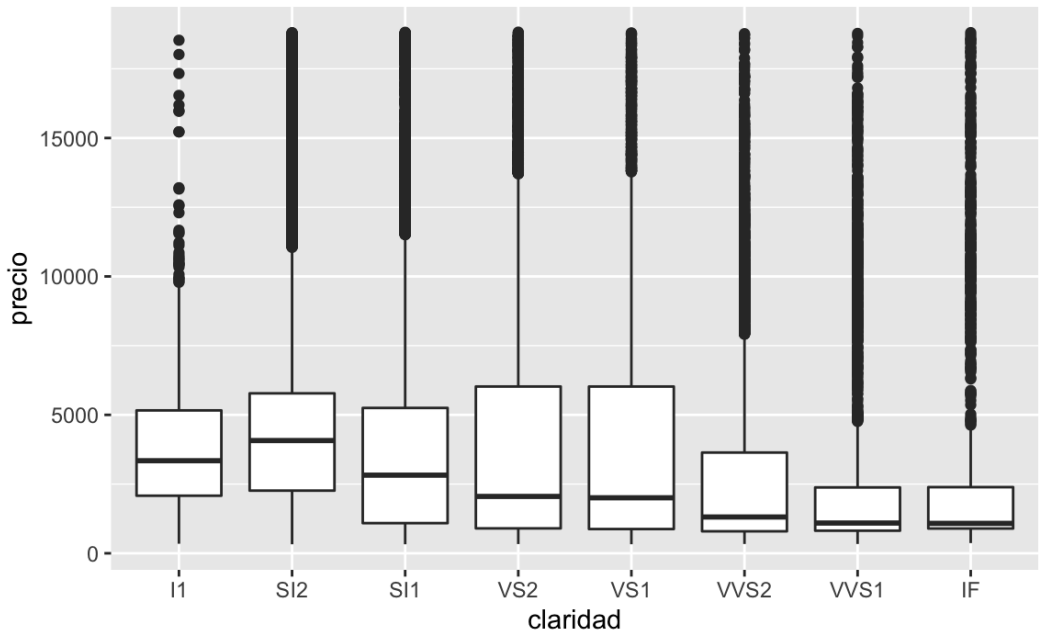
On this page

View source

Edit this page







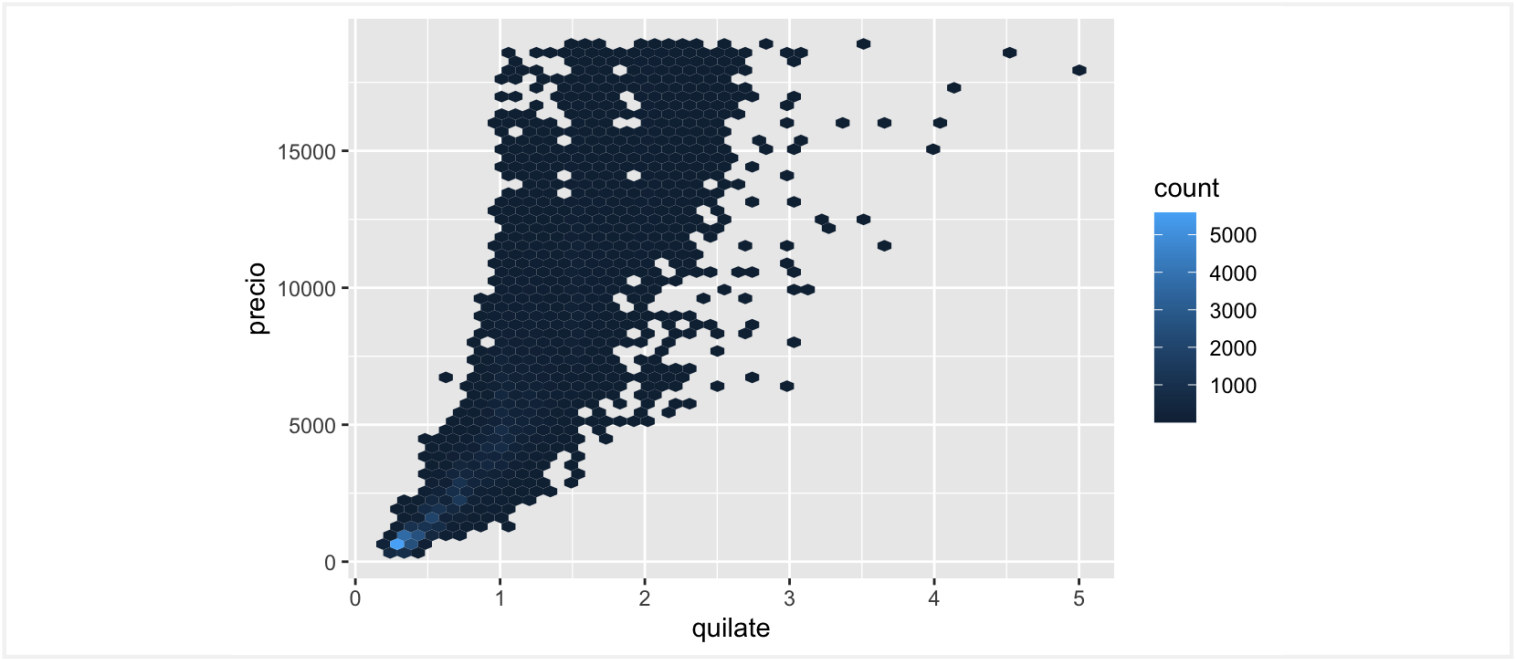
Ten en cuenta que el peor diamante es J (amarillo claro), y la peor claridad es I1 (inclusiones visibles a simple vista).

24.2.1 **Precio y quilates**

Pareciera que los diamantes de menor calidad tiene precios más altos porque hay una importante variable de confusión: el peso (`carat`) del diamante. El peso del diamante es el factor individual más importante para determinar el precio del diamante, y los diamantes de menor calidad tienden a ser más grandes.

```
ggplot(diamantes, aes(quilate, precio)) +
  geom_hex(bins = 50)
```

Copy



Podemos hacer que sea más fácil ver cómo los otros atributos de un diamante afectan su `precio` relativo al ajustar un modelo para separar el efecto de `quilates`. Pero primero, hagamos algunos ajustes al conjunto de datos de diamantes para que sea más fácil trabajar con ellos:

1. Foco en los diamantes más pequeños que 2.5 quilates (99.7% de los datos).
2. Haz una transformación logarítmica de las variables quilates y precio

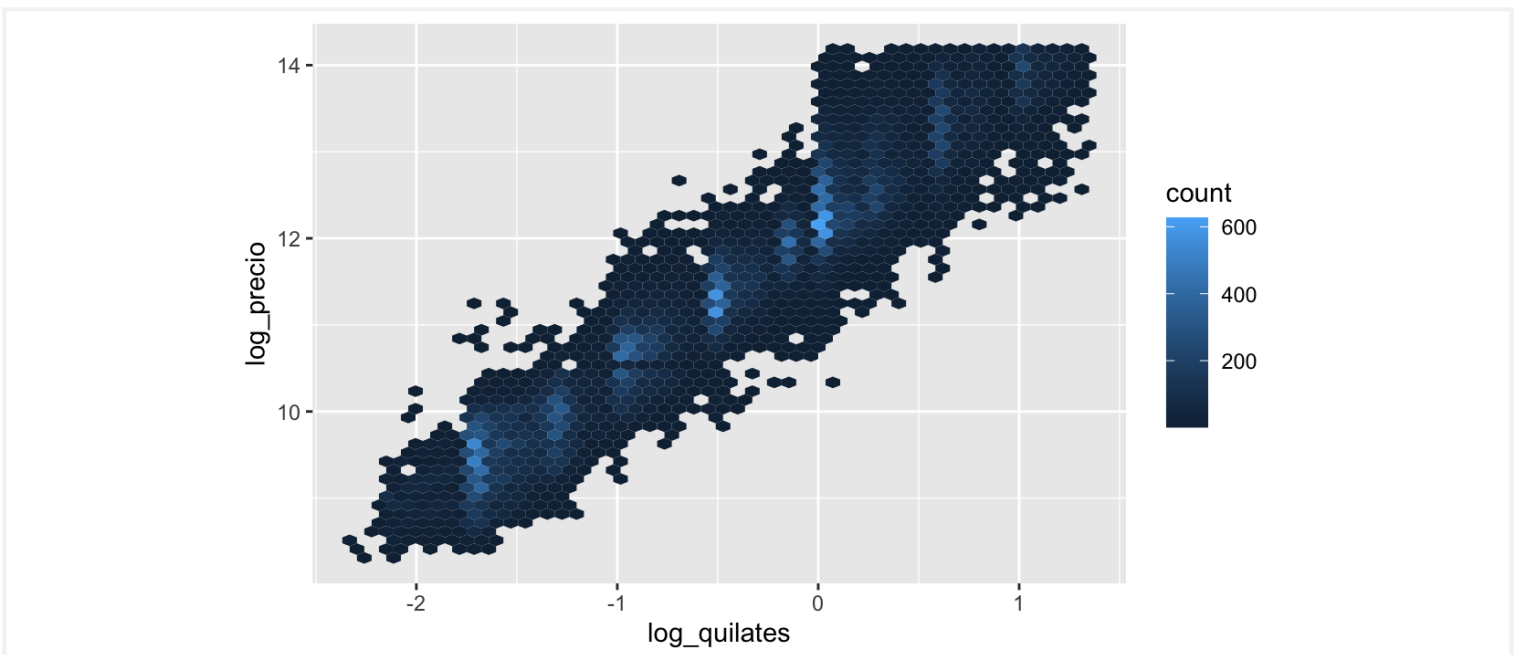
```
diamantes2 <- diamantes %>%
  filter(quilate <= 2.5) %>%
  mutate(log_precio = log2(precio), log_quilates = log2(quilate))
```

Copy

Juntos, esos cambios hacen más fácil ver la relación entre `quilates` y `precio`:

```
ggplot(diamantes2, aes(log_quilates, log_precio)) +
  geom_hex(bins = 50)
```

Copy



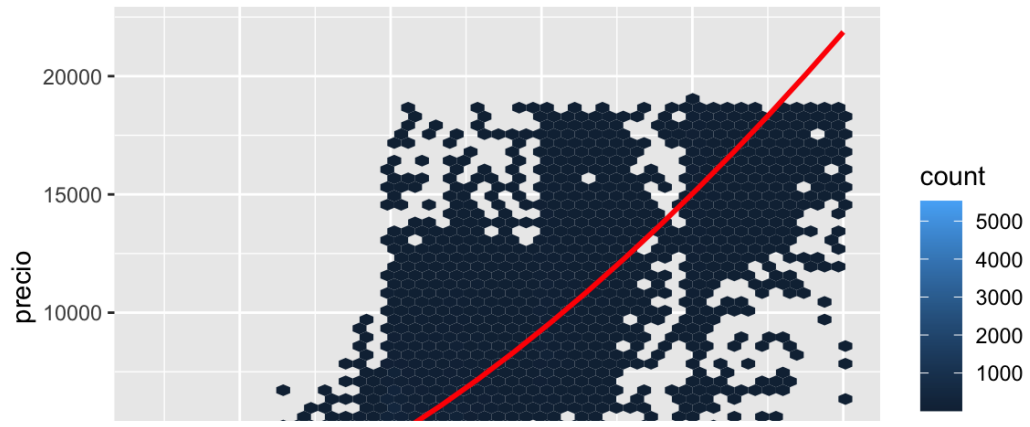
La transformación logarítmica es particularmente util aquí porque hace que el patrón sea lineal, y patrones lineales son más fáciles de usar. Tomemos el próximo paso y eliminemos ese patron lineal fuerte. Primero hacemos explícito el patrón ajustando el modelo:

```
mod_diamantes <- lm(log_precio ~ log_quilates, data = diamantes2)
```

Copy

Luego observamos lo que el modelo nos dice. Ten en cuenta que vuelvo atrás la transformación de la predicción, deshaciendo la transformación logarítmica, para poder superponer las predicciones sobre los datos originales:

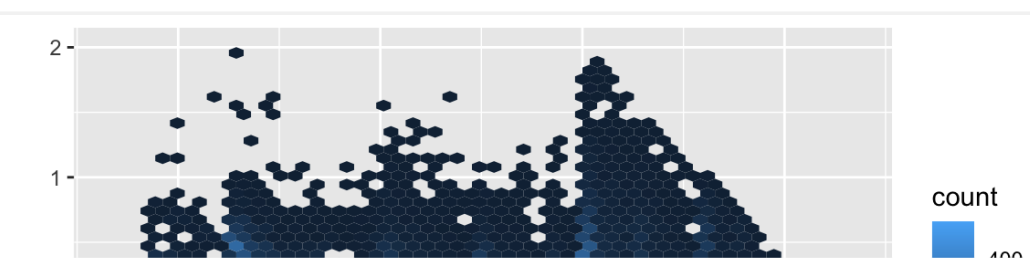
```
cuadricula <- diamantes2 %>%
  data_grid(quilate = seq_range(quilate, 20)) %>%
  mutate(log_quilates = log2(quilate)) %>%
  add_predictions(mod_diamantes, "log_precio") %>%
  mutate(precio = 2 ^ log_precio)

ggplot(diamantes2, aes(quilate, precio)) +
  geom_hex(bins = 50) +
  geom_line(data = cuadricula, colour = "red", size = 1)
```



Eso nos dice algo interesante acerca de nuestros datos. Si creemos en nuestro modelo, los diamantes grandes son mucho más baratos que lo esperado. Esto es posiblemente porque ninguno de los diamantes de estos datos cuesta más de US$19,000.

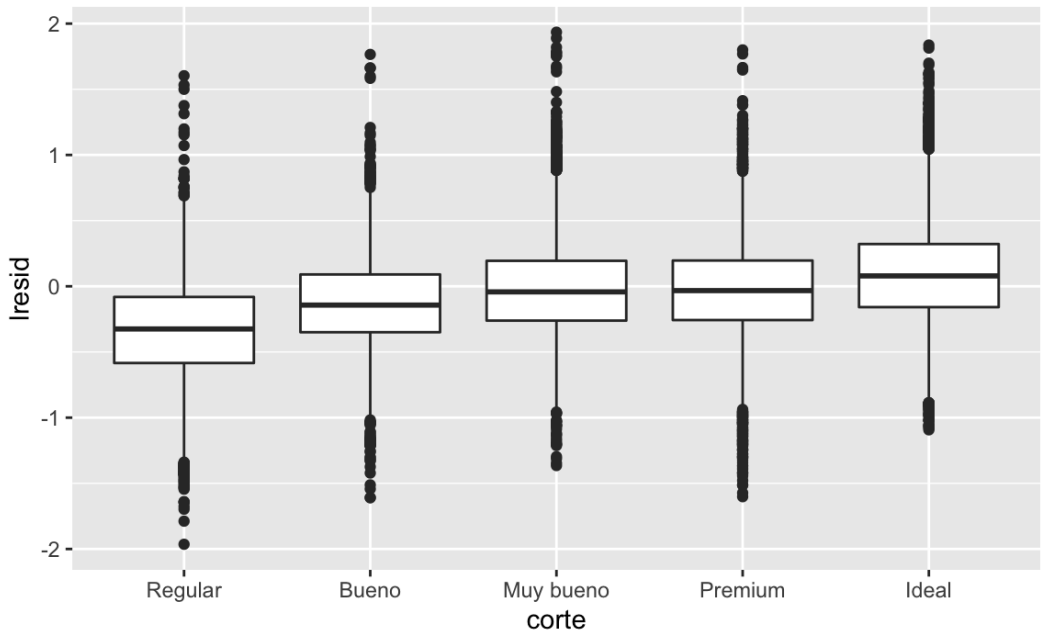
Ahora podemos ver los residuos, lo cual comprueba que hemos eliminado el patrón lineal fuerte:

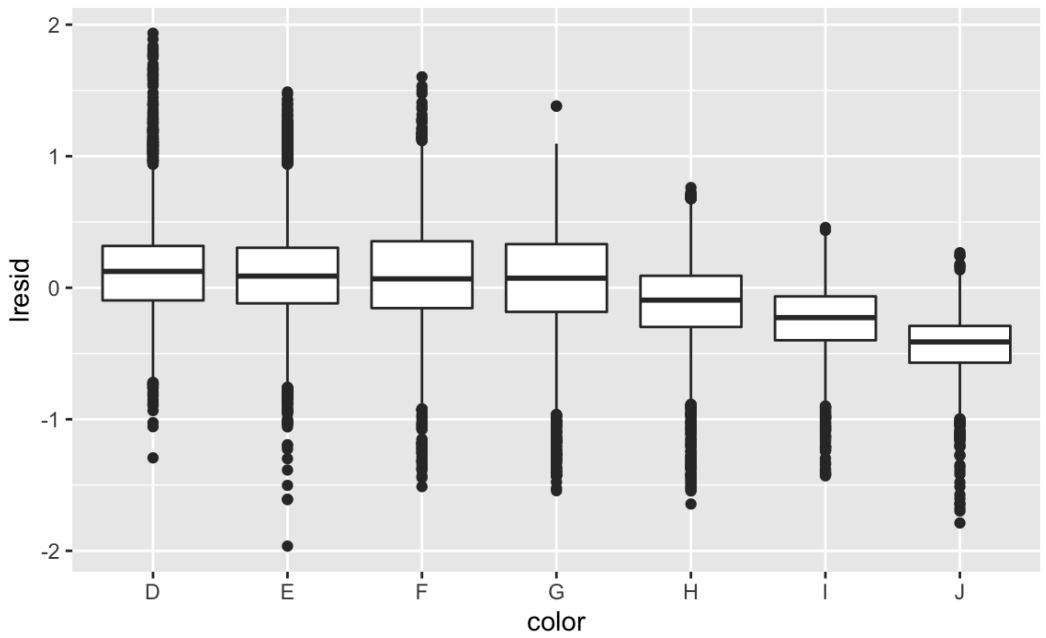
```
diamantes2 <- diamantes2 %>%
  add_residuals(mod_diamantes, "lresid")

ggplot(diamantes2, aes(log_quilates, lresid)) +
  geom_hex(bins = 50)
```

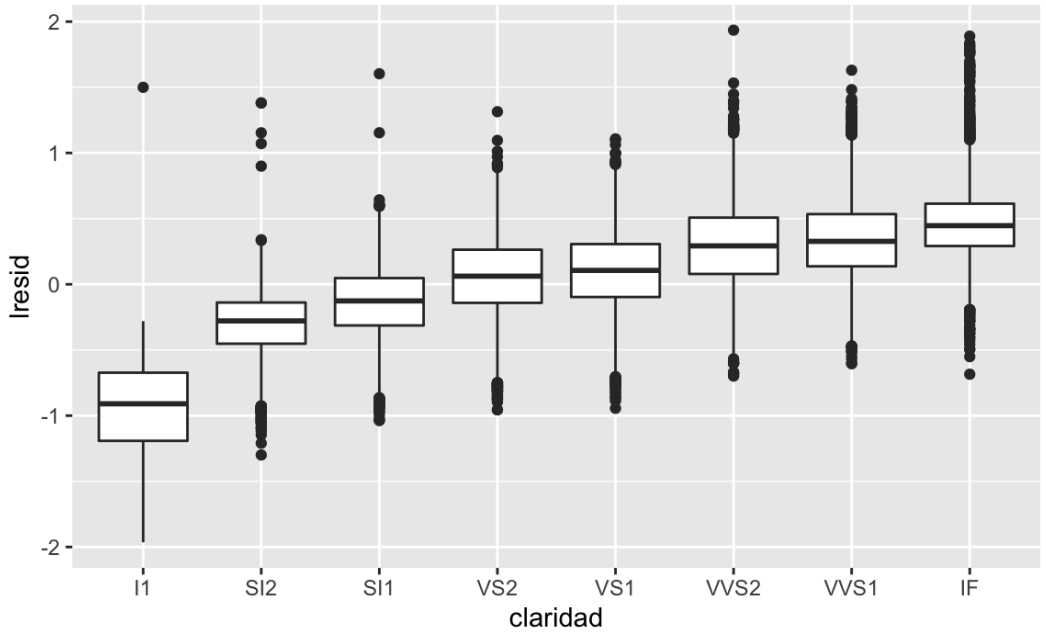


Es importante destacar que ahora podemos volver a hacer nuestros gráficos motivadores utilizando esos residuos en lugar de `precio`.

```
ggplot(diamantes2, aes(corte, lresid)) + geom_boxplot()
ggplot(diamantes2, aes(color, lresid)) + geom_boxplot()
ggplot(diamantes2, aes(claridad, lresid)) + geom_boxplot()
```







Ahora vemos la relación que esperábamos: a medida que aumenta la calidad del diamante, también lo hace su precio relativo. Para interpretar el eje `y`, necesitamos pensar que nos dicen los residuos, y en que escala están. Un residuo de -1 indica que `log_precio` era 1 unidad más baja que la predicción basada únicamente en su peso. 2^{-1} es 1/2, los puntos con un valor de -1 son la mitad del precio esperado, y los residuos con el valor 1 son el doble del precio predicho.

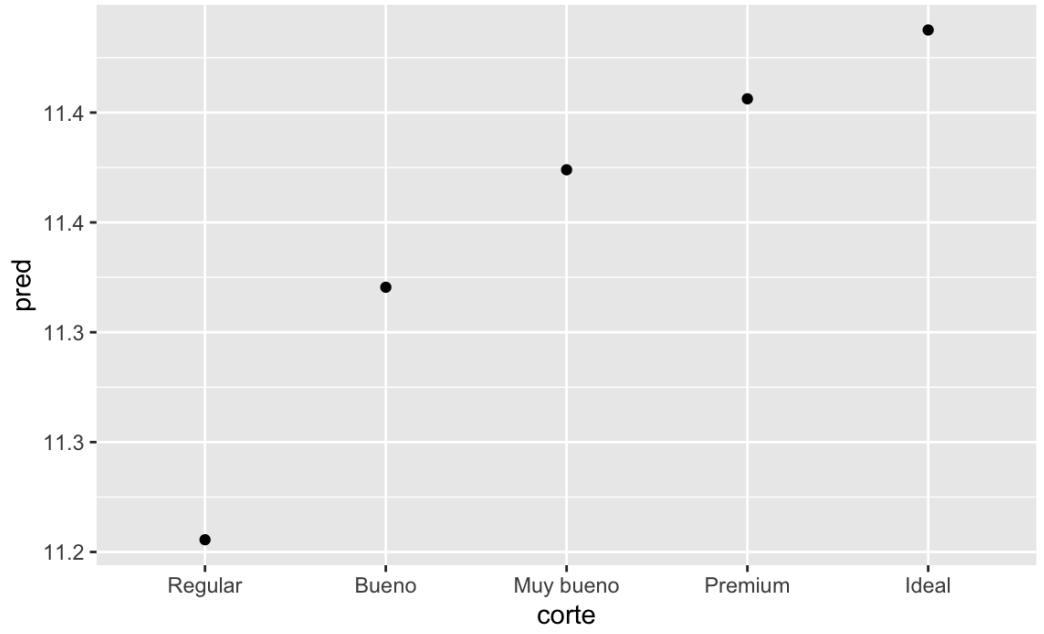
24.2.2 **Un modelo más complicado**

Si quisiéramos, podríamos continuar construyendo nuestro modelo, traspasando los resultados que hemos observado en el modelo para hacerlos explícitos. Por ejemplo, podríamos incluir `color`, `corte`, y `claridad` en el modelo para que también hagamos explícito el efecto de esas tres variables categóricas:

```
mod_diamantes2 <- lm(log_precio ~ log_quilates + color + corte + claridad, data = diamantes2
```
Copy

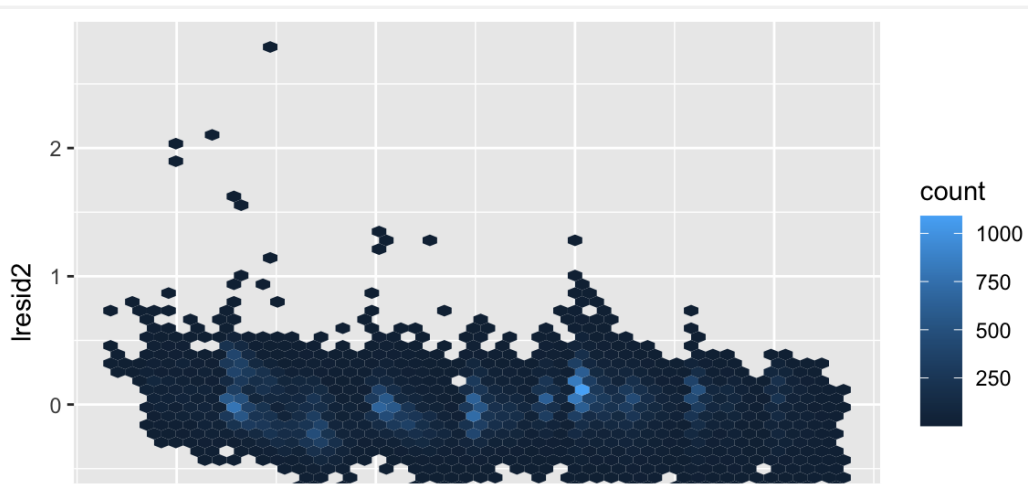
Este modelo ahora incluye cuatro predictores, por lo que es más difícil de visualizar. Afortunadamente, todos ellos son actualmente independientes lo que significa que podemos graficarlos individualmente en cuatro gráficos. Para hacer el proceso más fácil, vamos a usar el argumento `.model` en `data_grid`:

```
cuadricula <- diamantes2 %>%
  data_grid(corte, .model = mod_diamantes2) %>%
  add_predictions(mod_diamantes2)
cuadricula
#> # A tibble: 5 x 5
#>   corte     log_quilates color claridad  pred
#>   <ord>            <dbl> <chr> <chr>    <dbl>
#> 1 Regular         -0.515 G     VS2       11.2
#> 2 Bueno           -0.515 G     VS2       11.3
#> 3 Muy bueno       -0.515 G     VS2       11.4
#> 4 Premium         -0.515 G     VS2       11.4
#> 5 Ideal           -0.515 G     VS2       11.4

ggplot(cuadricula, aes(corte, pred)) +
  geom_point()
```



Si el modelo necesita variables que no has suministrado, `data_grid()` automáticamente los rellenará con el valor "typical". Para variables continuas, se usa la mediana, y para variables categóricas se usa el valor más frecuente (o valores, si hay un empate).

```
diamantes2 <- diamantes2 %>%
  add_residuals(mod_diamantes2, "lresid2")

ggplot(diamantes2, aes(log_quilates, lresid2)) +
  geom_hex(bins = 50)
```



Este gráfico indica que hay algunos diamantes con residuos bastante grandes - recuerda que un residuo de 2 indica que el diamante es 4x el precio que esperábamos. A menudo es útil mirar los valores inusuales individualmente:

```
diamantes2 %>%
  filter(abs(lresid2) > 1) %>%
  add_predictions(mod_diamantes2) %>%
  mutate(pred = round(2 ^ pred)) %>%
  select(precio, pred, quilate:tabla, x:z) %>%
  arrange(precio)
#> # A tibble: 16 x 11
#>   precio  pred quilate corte  color claridad profundidad tabla     x     y     z
#>    <int> <dbl>   <dbl> <ord>  <ord> <ord>          <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1   1013   264    0.25 Regul… F     SI2             54.4    64  4.3   4.23  2.32
#> 2   1186   284    0.25 Premi… G     SI2             59      60  5.33  5.28  3.12
#> 3   1186   284    0.25 Premi… G     SI2             58.8    60  5.33  5.28  3.12
#> 4   1262  2644    1.03 Regul… E     I1              78.2    54  5.72  5.59  4.42
#> 5   1415   639    0.35 Regul… G     VS2             65.9    54  5.57  5.53  3.66
#> 6   1415   639    0.35 Regul… G     VS2             65.9    54  5.57  5.53  3.66
#> # … with 10 more rows
```

Hasta aquí nada realmente interesante, pero probablemente valga la pena pasar tiempo considerando si esto significa un problema con nuestro modelo, o si hay errores en los datos. Si hay errores en los datos, esta podría ser una oportunidad para comprar diamantes que fueron incorrectamente tasados con valor bajo.

24.2.3 Ejercicios

1. En el gráfico de `log_quilates` vs. `log_precio`, hay unas tiras verticales brillantes. ¿Qué representan?

2. Si `log(precio) = a_0 + a_1 * log(quilates)`, ¿Qué dice eso acerca la relación entre `precio` y `quilates`?

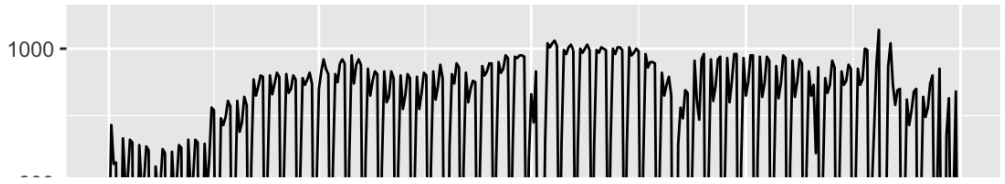
3. Extrae los diamantes que tienen residuos muy altos y muy bajos. ¿Hay algo inusual en estos diamantes? ¿Son particularmente malos o buenos?, o ¿Crees que estos son errores de precio?

4. ¿El modelo final, `mod_diamantes2`, hace un buen trabajo al predecir el precios de los diamantes? ¿Confiarías en lo que te indique gastar si fueras a comprar un diamante?

24.3 ¿Qué afecta el número de vuelos diarios?

Trabajaremos a través de un proceso similar para un conjunto de datos que parece aún más simple a primera vista: el número de vuelos que salen de NYC por día. Este es un conjunto realmente pequeño de datos — solo 365 filas y 2 columnas — y no vamos a terminar con un modelo completamente realizado, pero como veras, los pasos en el camino nos ayudarán a entender mejor los datos. Comenzaremos contando el número de vuelos por día y visualizándolos con ggplot2.

```
vuelos_por_dia <- vuelos %>%
  mutate(fecha = make_date(anio, mes, dia)) %>%
  group_by(fecha) %>%
  summarise(n = n())
vuelos_por_dia
#> # A tibble: 365 x 2
#>   fecha          n
#> * <date>     <int>
#> 1 2013-01-01   842
#> 2 2013-01-02   943
#> 3 2013-01-03   914
#> 4 2013-01-04   915
#> 5 2013-01-05   720
#> 6 2013-01-06   832
#> # … with 359 more rows

ggplot(vuelos_por_dia, aes(fecha, n)) +
  geom_line()
```

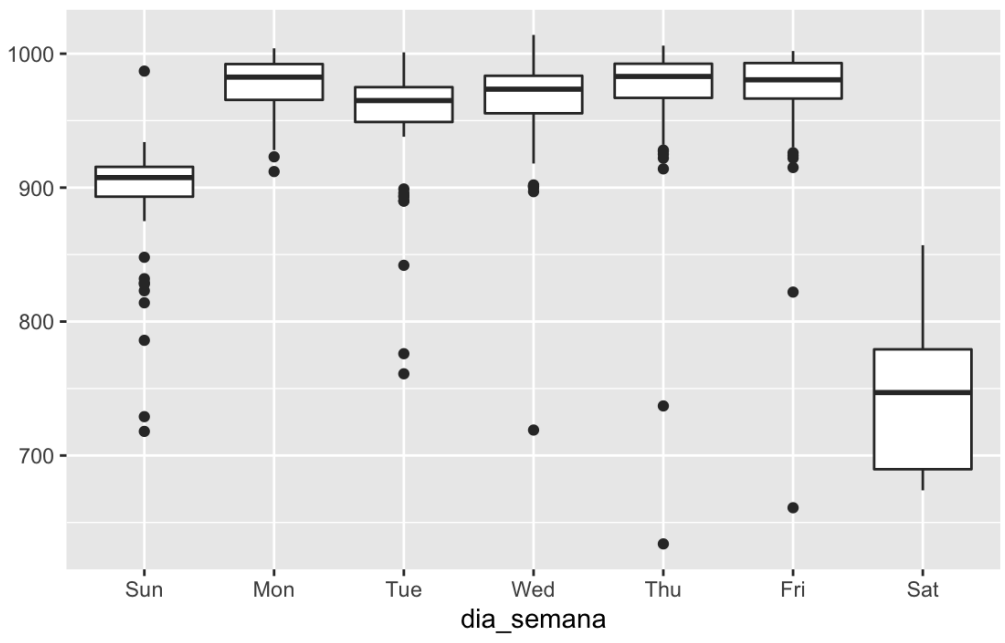


24.3.1 **Día de la semana**

Comprender la tendencia a largo plazo es un desafío porque hay un fuerte efecto de día de la semana que domina a los patrones más sutiles. Comenzaremos mirando la distribución de número de vuelos por día de la semana:

```
vuelos_por_dia <- vuelos_por_dia %>%
  mutate(dia_semana = wday(fecha, label = TRUE))
ggplot(vuelos_por_dia, aes(dia_semana, n)) +
  geom_boxplot()
```

Copy

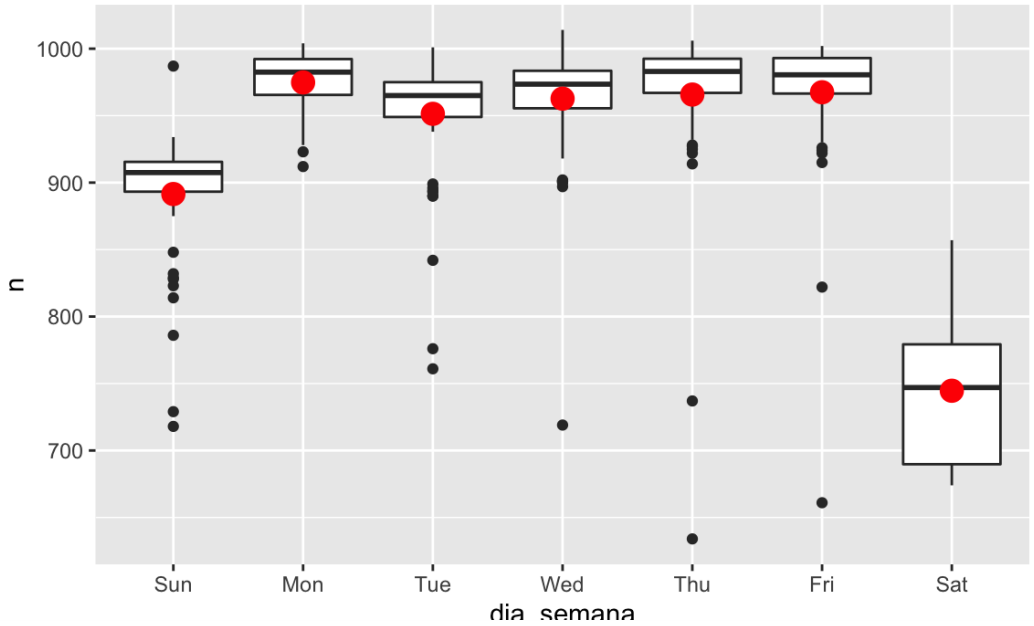


Hay pocos vuelos los fines de semana porque la mayoría de los viajes son por negocios. El efecto es particularmente pronunciado el sábado: alguna vez podrías salir un domingo para ir a una reunión un lunes por la mañana, pero es menos común que salgas un sábado ya que preferirías estar en casa con tu familia.

Una forma de eliminar este fuerte patrón es usar un modelo. Primero, ajustamos el modelo, y mostraremos sus predicciones superpuestas sobre los datos originales:

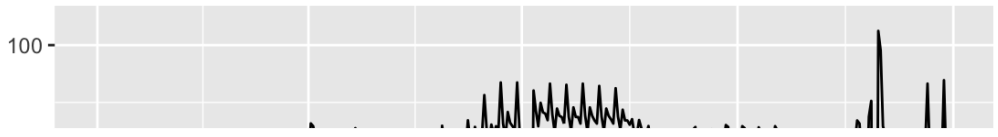
```
mod <- lm(n ~ dia_semana, data = vuelos_por_dia)

cuadricula <- vuelos_por_dia %>%
  data_grid(dia_semana) %>%
  add_predictions(mod, "n")

ggplot(vuelos_por_dia, aes(dia_semana, n)) +
  geom_boxplot() +
  geom_point(data = cuadricula, colour = "red", size = 4)
```

Copy



A continuación calculamos y visualizamos los residuos:

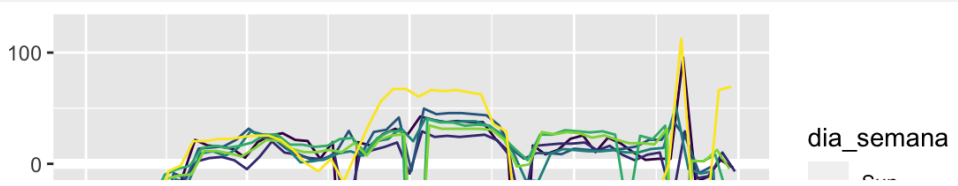
```
vuelos_por_dia <- vuelos_por_dia %>%
  add_residuals(mod)
vuelos_por_dia %>%
  ggplot(aes(fecha, resid)) +
  geom_ref_line(h = 0) +
  geom_line()
```



Notar el cambio en el eje Y: ahora estamos viendo el desvío desde el número de vuelos esperados, dado el día de la semana. Este gráfico es útil porque ahora que removimos la mayor parte del efecto día de la semana, podemos ver algo de los patrones más sutiles que quedan:

1. Nuestro modelo parece fallar a partir de junio: todavía se puede ver un patrón regular fuerte que nuestro modelo no ha capturado. Dibujando un diagrama con una línea para cada día de la semana se hace más facil de ver la causa:

```
ggplot(vuelos_por_dia, aes(fecha, resid, colour = dia_semana)) +
  geom_ref_line(h = 0) +
  geom_line()
```



Nuestro modelo falla en predecir con precisión el número de vuelos los sábados: durante el verano hay más vuelos de los que esperamos, y durante el otoño hay menos. Veremos como podemos capturar mejor este patrón en la siguiente sección.

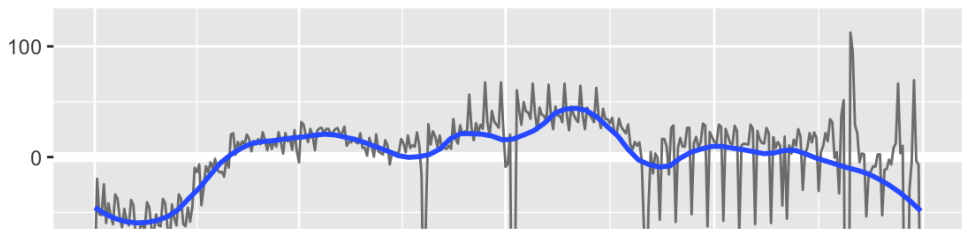
2. Hay algunos días con mucho menos vuelos que los esperados:

```
vuelos_por_dia %>%
  filter(resid < -100)
#> # A tibble: 11 x 4
#>   fecha          n dia_semana resid
#>   <date>     <int> <ord>      <dbl>
#> 1 2013-01-01   842 Tue        -109.
#> 2 2013-01-20   786 Sun        -105.
#> 3 2013-05-26   729 Sun        -162.
#> 4 2013-07-04   737 Thu        -229.
#> 5 2013-07-05   822 Fri        -145.
#> 6 2013-09-01   718 Sun        -173.
#> # … with 5 more rows
```

Si estás familiarizado con los feriados públicos de Estados Unidos, podrías reconocer los días de Año Nuevo, el 4 de julio, el día de Acción de Gracias y Navidad. Hay algunos otros que parecen no corresponder a feriados públicos. Trabajarás sobre esos días en uno de los ejercicios.

3. Parece que hay una tendencia más suave a largo plazo en el transcurso del año. Podemos destacar esa tendencia con `geom_smooth()`:

```
vuelos_por_dia %>%
  ggplot(aes(fecha, resid)) +
  geom_ref_line(h = 0) +
  geom_line(colour = "grey50") +
  geom_smooth(se = FALSE, span = 0.20)
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

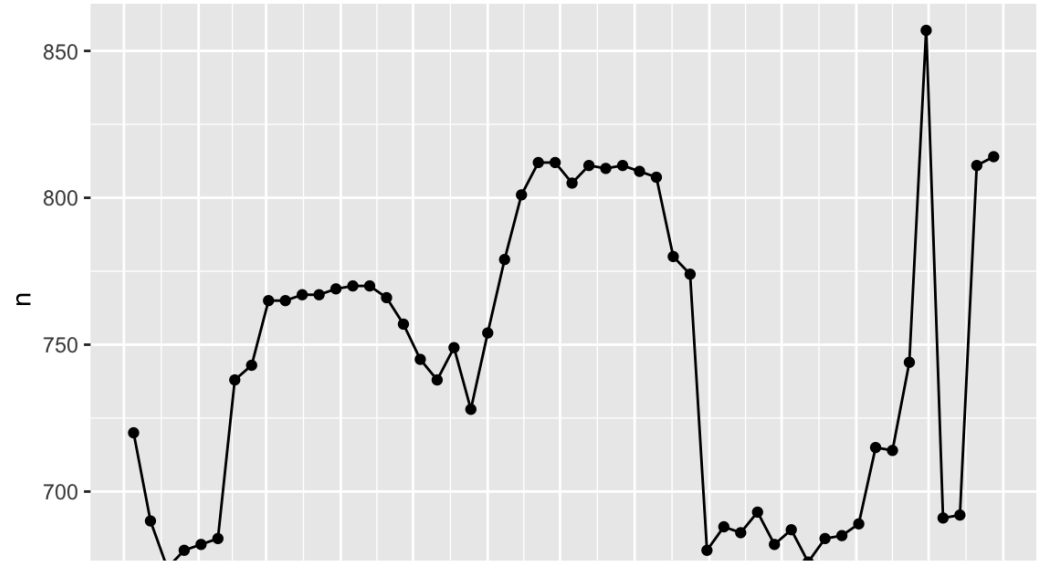


Hay menos vuelos en enero (y diciembre), y más en verano (May-Sep). No podemos hacer mucho cuantitativamente con este patrón, porque
sólo tenemos un año de datos. Pero podemos usar nuestro conocimiento para pensar en posibles explicaciones.

24.3.2 Efecto estacional del sábado

Primero abordaremos nuestra falla para predecir con exactitud el número de vuelos el sábado. Un buen lugar para empezar es volver a los números originales, enfocándonos en el sábado:

```
vuelos_por_dia %>%
  filter(dia_semana == "Sat") %>%
  ggplot(aes(fecha, n)) +
    geom_point() +
    geom_line() +
    scale_x_date(NULL, date_breaks = "1 month", date_labels = "%b")
```



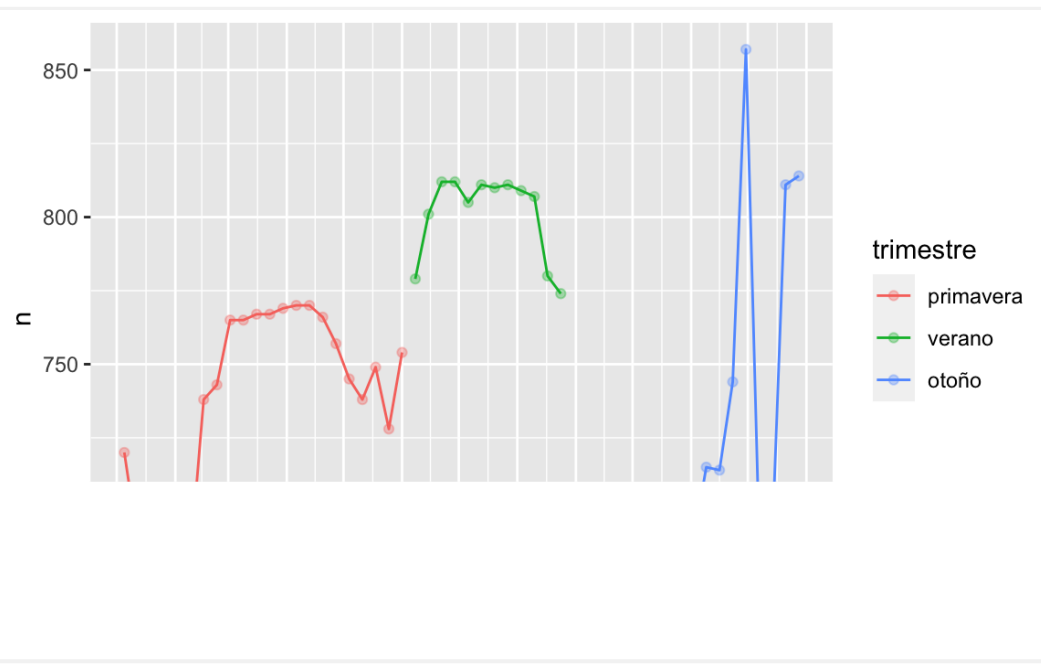
(He usado tanto puntos como líneas para dejar más claro que son los datos y qué es la interpolación).

Sospecho que este patrón es causado por las vacaciones de invierno: mucha gente va de vacaciones en verano, y a las personas no les importa viajar un sábado en sus vacaciones. Al mirar este gráfico, podemos suponer que las vacaciones de verano son de principio de junio a finales de agosto. Parece que se alinea bastante bien con el calendario escolar del estado de NY: las vacaciones de verano en 2013 fueron del 26 de junio hasta el 9 de septiembre.

¿Por qué hay más vuelos los sábados en primavera que en otoño? Le pregunté a algunos amigos estadounidenses y ellos me dijeron que es menos común planificar vacaciones familiares durante el otoño porque debido a los grandes feriados de Acción de Gracias y Navidad. No tenemos los datos para estar seguros, pero parecería una hipótesis de trabajo razonable.

Vamos a crear la variable "trimestre" que capture aproximadamente los tres períodos escolares, y verificamos nuestro trabajo con un gráfico:

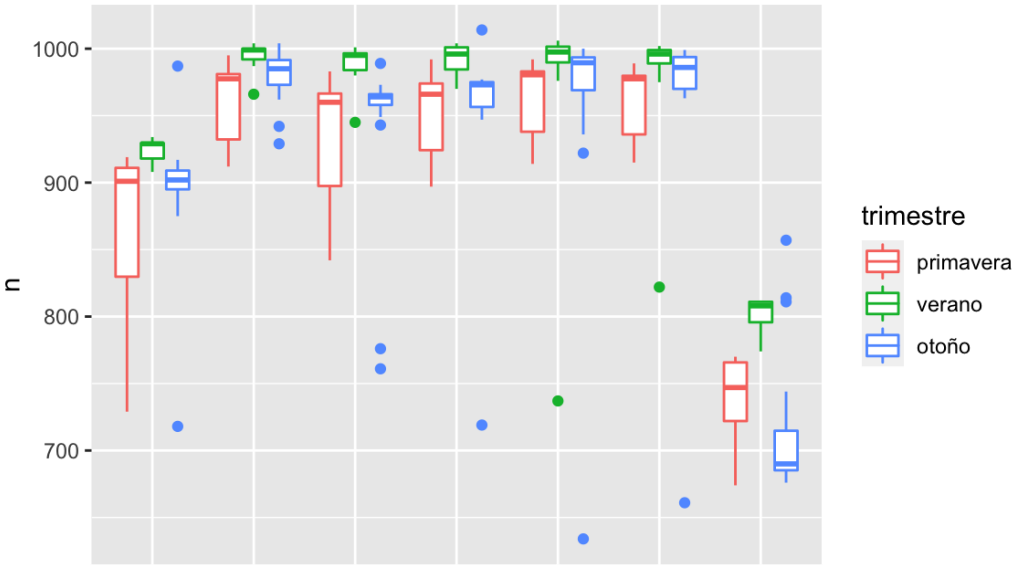
```
trimestre <- function(fecha) {
  cut(fecha,
    breaks = ymd(20130101, 20130605, 20130825, 20140101),
    labels = c("primavera", "verano", "otoño")
  )
}

vuelos_por_dia <- vuelos_por_dia %>%
  mutate(trimestre = trimestre(fecha))

vuelos_por_dia %>%
  filter(dia_semana == "Sat") %>%
  ggplot(aes(fecha, n, colour = trimestre)) +
  geom_point(alpha = 1/3) +
  geom_line() +
  scale_x_date(NULL, date_breaks = "1 month", date_labels = "%b")
```



(Modifiqué manualmente las fechas para obtener mejores cortes en el gráfico. Una técnica general y realmente poderosa es usar una visualización para ayudarte a entender lo que tu función está haciendo).

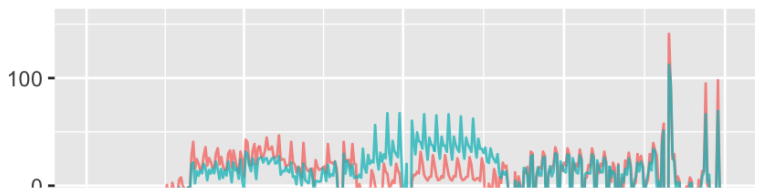
Es útil ver como esta nueva variable afecta los otros días de la semana:

```
vuelos_por_dia %>%
  ggplot(aes(dia_semana, n, colour = trimestre)) +
    geom_boxplot()
```



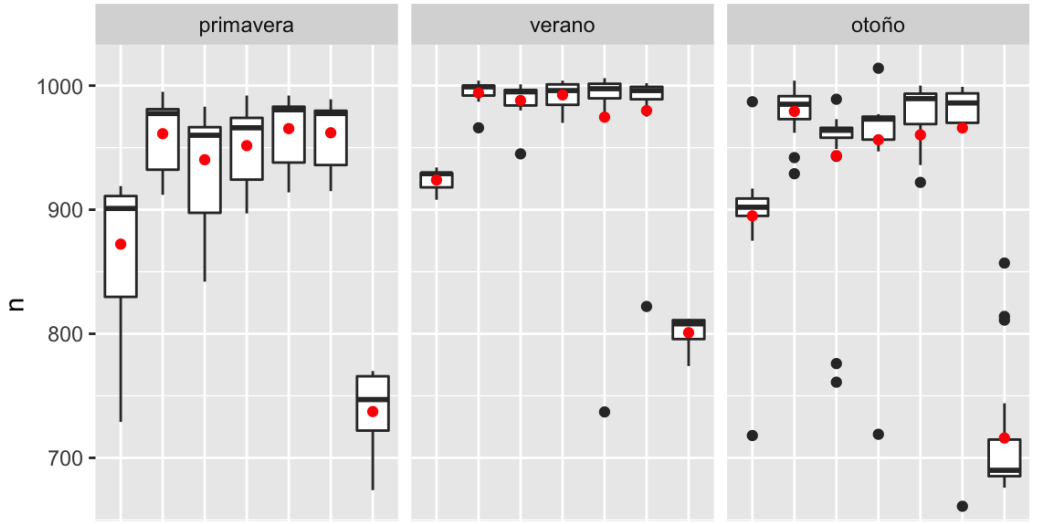
Parece que hay una variación significativa entre los periodos, por lo que es razonable ajustar el efecto de los días de la semana por separado para cada período. Esto mejora nuestro modelo, pero no tanto como podríamos esperar:

```
mod1 <- lm(n ~ dia_semana, data = vuelos_por_dia)
mod2 <- lm(n ~ dia_semana * trimestre, data = vuelos_por_dia)

vuelos_por_dia %>%
  gather_residuals(sin_trimestre = mod1, con_trimestre = mod2) %>%
  ggplot(aes(fecha, resid, colour = model)) +
    geom_line(alpha = 0.75)
```



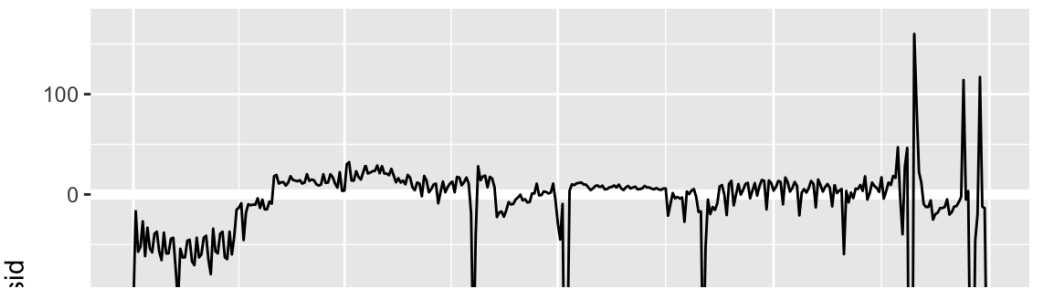
Podemos ver el problema al superponer las predicciones del modelo a los datos crudos:

```
cuadricula <- vuelos_por_dia %>%
  data_grid(dia_semana, trimestre) %>%
  add_predictions(mod2, "n")

ggplot(vuelos_por_dia, aes(dia_semana, n)) +
  geom_boxplot() +
  geom_point(data = cuadricula, colour = "red") +
  facet_wrap(~ trimestre)
```



Nuestro modelo esta encontrando el efecto *mean* , pero tenemos muchos valores atípicos grandes, por lo tanto la media tiende a estar lejos de los valores atípicos. Podemos aliviar este problema usando un modelo que es más robusto a los efectos de los valores atípicos: `MASS::rlm()` . Esto reduce en gran medida el impacto de los valores atípicos en nuestras estimaciones, y proporciona un modelo que hace un buen trabajo eliminando el patrón del día de la semana:

```
mod3 <- MASS::rlm(n ~ dia_semana * trimestre, data = vuelos_por_dia)

vuelos_por_dia %>%
  add_residuals(mod3, "resid") %>%
  ggplot(aes(fecha, resid)) +
  geom_hline(yintercept = 0, size = 2, colour = "white") +
  geom_line()
```



Ahora es mucho más fácil ver la tendencia a largo plazo, los positivos y negativos valores atípicos.

24.3.3 Variables calculadas

Si estas experimentando con muchos modelos y muchas visualizaciones, es una buena idea agrupar la creación de variables en una función para que no haya posibilidad de aplicar accidentalmente transformaciones a diferentes lugares. Por ejemplo, podríamos escribir:

```
compute_vars <- function(data) {
  data %>%
    mutate(
      trimestre = trimestre(date),
      dia_semana = wday(date, label = TRUE)
    )
}
```

Otra opción es colocar las transformaciones directamente en la fórmula del modelo:

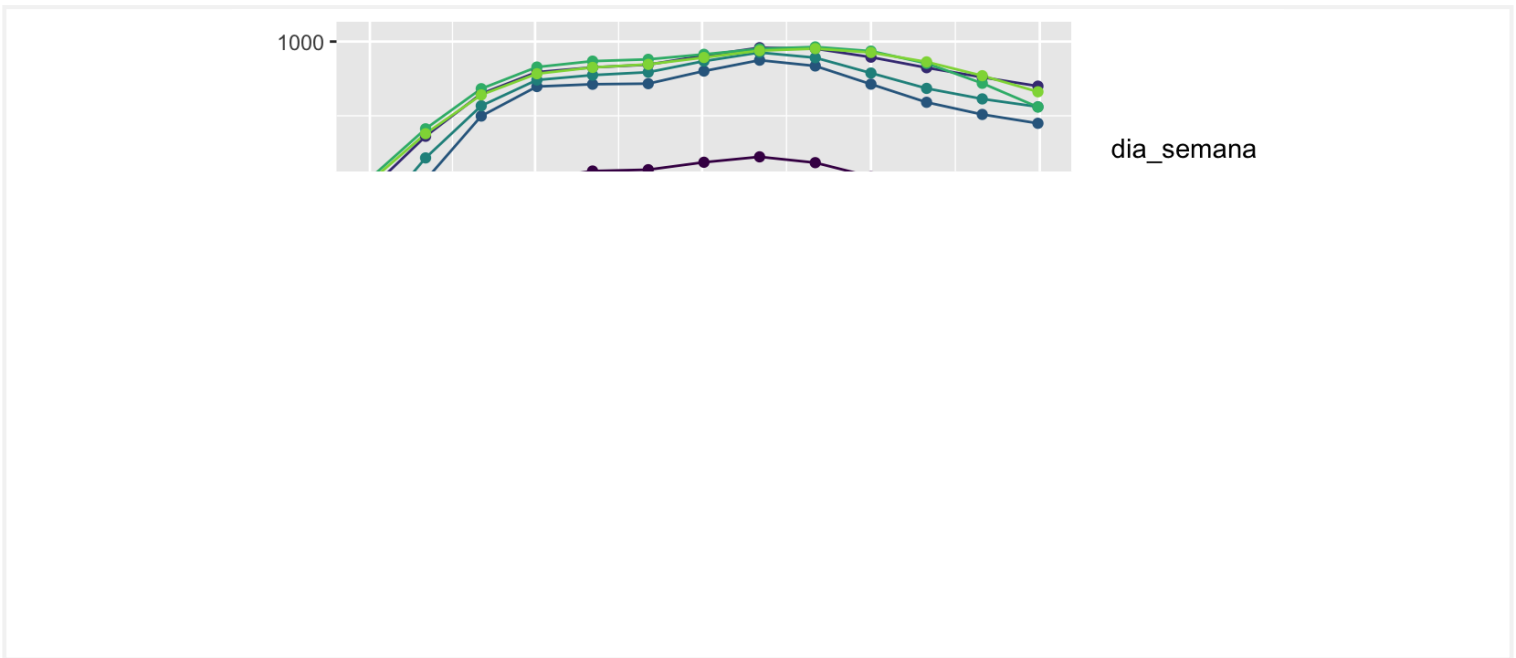
```
dia_semana2 <- function(x) wday(x, label = TRUE)
mod3 <- lm(n ~ dia_semana2(fecha) * trimestre(fecha), data = vuelos_por_dia)
```

Cualquiera de los enfoques es razonable. Hacer que una variable transformada sea explicita es útil si quieres verificar tu trabajo, o usarlas en una visualización. Pero no puedes usar fácilmente transformaciones (como splines) que devuelven múltiples columnas. Incluir las transformaciones en el modelo hace la vida más fácil cuando se trabaja con diferentes conjuntos de datos porque el modelo es autónomo.

24.3.4 Época del año: un enfoque alternativo

En la sección anterior usamos nuestro conocimiento (como el calendario escolar de Estados Unidos afecta el viaje) para mejorar el modelo. Una alternativa es utilizar nuestro conocimiento explícito en el modelo para darle a los datos más espacio para hablar. Podríamos utilizar un modelo más flexible y permitir que capture el patrón que nos interesa. Una tendencia lineal simple no es adecuada, por lo que podríamos intentar usar una spline natural para ajustarnos a una curva suave durante el año:

```
library(splines)
mod <- MASS::rlm(n ~ dia_semana * ns(fecha, 5), data = vuelos_por_dia)

vuelos_por_dia %>%
  data_grid(dia_semana, fecha = seq_range(fecha, n = 13)) %>%
  add_predictions(mod) %>%
  ggplot(aes(fecha, pred, colour = dia_semana)) +
  geom_line() +
  geom_point()
```



Vemos un patrón fuerte en el número de vuelos de los sábados. Esto es tranquilizador, porque también vimos ese patrón en los datos sin transformar. Es una buena señal cuando obtienes la misma señal desde diferentes enfoques.

24.3.5 Ejercicios

1. Usa tus habilidades detestivescas con los buscadores para intercambiar ideas sobre por qué hubo menos vuelos esperados el 20 de enero, 26 de mayo y 1 de septiembre. (Pista: todos tienen la misma explicación.) ¿Cómo generalizarías esos días a otros años?

2. ¿Qué representan esos tres días con altos residuos positivos? ¿Cómo se generalizarían esos días a otros años?

```
vuelos_por_dia %>%
  slice_max(n = 3, resid)
#> # A tibble: 3 x 5
#>   fecha          n dia_semana resid trimestre
#>   <date>     <int> <ord>      <dbl> <fct>
#> 1 2013-11-30   857 Sat        112.  otoño
#> 2 2013-12-01   987 Sun         95.5 otoño
#> 3 2013-12-28   814 Sat         69.4 otoño
```

3. Crea una nueva variable que divida la variable `dia_semana` en periodos, pero sólo para sábados, es decir, debería tener `Thu`, `Fri`, y `Sat-verano`, `Sat-primavera`, `Sat-otonio`. ¿Cómo este modelo se compara con el modelo que tiene la combinación de `dia_semana` y `trimestre`?

4. Crea una nueva variable `dia_semana` que combina el día de la semana, periodos (para sábados), y feriados públicos. ¿Cómo se ven los residuos de este modelo?

5. ¿Qué sucede si ajustas un efecto de día de la semana que varía según el mes o varía mes a mes (es decir, `n ~ dia_semana * month`)? ¿Por qué esto no es muy útil?

6. ¿Que esperarías del modelo `n ~ dia_semana + ns(fecha, 5)` ? Sabiendo lo que sabes sobre los datos, ¿porqué esperarias que no sea particularmente efectivo?

7. Presumimos que las personas que salen los domingos son probablemente viajeros de negocios quienes necesitan estar en algun lugar el lunes. Explora esa hipótesis al ver cómo se descompone en función de la distancia y tiempo: si es verdad, esperarías ver más vuelos en la tarde del domingo a lugares que estan muy lejos.

8. Es un poco frustante que el domingo y sábado esté en los extremos opuestos del gráfico. Escribe una pequeña función para establecer los niveles del factor para que la semana comience el lunes.

24.4 Aprende más sobre los modelos

Solo hemos dado una pincelada sobre el tema de modelos, pero es de esperar que hayas adquirido algunas herramientas simples, pero de uso general que puedes usar para mejorar tus propios análisis. ¡Está bien empezar de manera simple! Como has visto, incluso modelos muy simples pueden significar una gran diferencia en tu habilidad para desentrañar las interacciones o pueden generar un incremento dramático en tu habilidad para desentrañar las interacciones.

Estos capítulos de modelos son aún más dogmáticos que el resto del libro. Yo enfoco el modelamiento desde una perspectiva diferente a la mayoría, y hay relativamente poco espacio dedicado a ello. El modelamiento realmente requiere un libro completo, así que recomiendo que leas alguno de estos 3 libros:

- *Statistical Modeling: A Fresh Approach* by Danny Kaplan, http://project-mosaic-books.com/?page_id=13. Este libro provee una introducción suave al modelado, donde desarrollas tu intuición, herramientas matemáticas, y habilidades de R en paralelo. El libro reemplaza al curso tradicional de "Introducción a la Estadística", proporcionando un plan de estudios actualizado y relevante para ciencia de datos.

- *An Introduction to Statistical Learning* by Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani, http://www-bcf.usc.edu/~gareth/ISL/ (Disponible en línea gratis). Este libro presenta una moderna familia de técnicas de modelamiento colectivamente conocidas como aprendizaje estadístico. Para una más profunda comprensión de la matemática detrás de los modelos, lee el clásico *Elements of Statistical Learning* por Trevor Hastie, Robert Tibshirani, y Jerome Friedman, http://statweb.stanford.edu/~tibs/ElemStatLearn/ (También disponible en línea gratis).

- *Applied Predictive Modeling* por Max Kuhn and Kjell Johnson, http://appliedpredictivemodeling.com. Este libro es un compañero del paquete **caret** y provee herramientas prácticas para lidiar con desafíos de modelado predictivo.

« 23 Modelos: conceptos básicos

25 Muchos modelos »

25 Muchos modelos

25.1 Introducción

En este capítulo vas a aprender tres ideas poderosas que te van a ayudar a trabajar fácilmente con un gran número de modelos:

1. Usar muchos modelos simples para entender mejor conjuntos de datos complejos.

2. Usar columnas-lista (*list-columns*) para almacenar estructuras de datos arbitrarias en un *data frame*. Esto, por ejemplo, te permitirá tener una columna que contenga modelos lineales.

3. Usar el paquete **broom**, de David Robinson, para transformar modelos en datos ordenados. Esta es una técnica poderosa para trabajar con un gran número de modelos porque una vez que tienes datos ordenados, puedes aplicar todas las técnicas que has aprendido anteriormente en el libro.

Empezaremos entrando de lleno en un ejemplo motivador usando datos sobre la esperanza de vida alrededor del mundo. Es un conjunto de datos pequeño pero que ilustra cuán importante puede ser modelar para mejorar tus visualizaciones. Utilizaremos un número grande de modelos simples para separar algunas de las señales más fuertes y así poder ver las señales sutiles que permanecen. También veremos cómo las medidas de resumen de los modelos nos pueden ayudar a encontrar datos atípicos y tendencias inusuales.

Las siguientes secciones ahondarán en más detalles acerca de cada una de estas técnicas:

1. En columnas-lista aprenderás más acerca de la estructura de datos *columna-lista* y por qué es válido poner listas en *data frames*.

2. En creando columnas-lista aprenderás las tres maneras principales en las que crearás columnas-lista.

3. En simplificando columnas-lista aprenderás cómo convertir columnas-lista de vuelta a vectores atómicos regulares (o conjuntos de vectores atómicos) para que puedas trabajar con ellos más fácilmente.

4. En haciendo datos ordenados con broom aprenderás sobre el conjunto de herramientas completo provisto por **broom** (*escoba*, en inglés) y verás cómo puede ser aplicado a otros tipos de estructuras de datos.

Este capítulo es en cierta medida aspiracional: si este libro es tu primera introducción a R, este capítulo probablemente será un desafío. Requiere que tengas profundamente internalizadas ideas acerca de modelado, estructuras de datos e iteración. Así que no te preocupes si no lo entiendes — solo aparta este capítulo por un par de meses, y vuelve cuando quieras ejercitar tu cerebro.

25.1.1 Prerrequisitos

Trabajar con muchos modelos requiere muchos de los paquetes del **tidyverse** (para exploración de datos, manejo y programación) y **modelr** para facilitar el modelado.

```
library(modelr)
library(tidyverse)
```

Copy

25.2 gapminder

Como motivación para entender el poder de muchos modelos simples, vamos a mirar los datos de "gapminder". Estos datos fueron popularizados por Hans Rosling, un doctor y estadístico sueco. Si nunca has escuchado de él, para de leer este capítulo ahora mismo y revisa algunos de sus videos! Es un fantástico presentador de datos e ilustra cómo puedes usar datos para presentar una historia convincente. Un buen lugar para empezar es este video corto filmado en conjunto con la BBC: https://www.youtube.com/watch?v=jbkSRLYSojo.

Los datos de gapminder resumen la progresión de países a través del tiempo, mirando estadísticos como esperanza de vida y PIB. Los datos son de fácil acceso en R gracias a Jenny Bryan, quien creó el paquete **gapminder**. Utilizaremos la versión en español contenida en el paquete **datos**, que incorpora este dataset en el objeto `paises`.

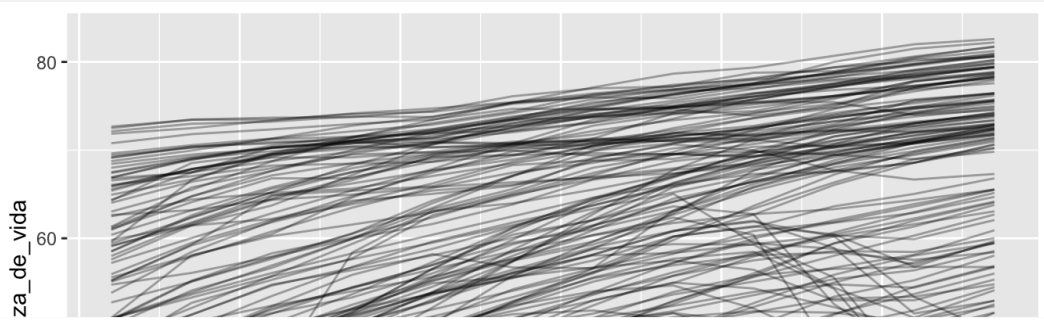
```
library(datos)
paises
#> # A tibble: 1,704 x 6
#>   pais       continente  anio esperanza_de_vida poblacion pib_per_capita
#>   <fct>      <fct>      <int>             <dbl>     <int>          <dbl>
#> 1 Afganistán Asia        1952              28.8   8425333           779.
#> 2 Afganistán Asia        1957              30.3   9240934           821.
#> 3 Afganistán Asia        1962              32.0  10267083           853.
#> 4 Afganistán Asia        1967              34.0  11537966           836.
#> 5 Afganistán Asia        1972              36.1  13079460           740.
#> 6 Afganistán Asia        1977              38.4  14880372           786.
#> # … with 1,698 more rows
```

Copy

En este caso de estudio, nos enfocaremos solo en tres variables para responder la pregunta "¿Cómo la esperanza de vida (`esperanza_de_vida`) cambia a través del tiempo (`anio`) para cada país (`pais`)?". Un buen lugar para empezar es con un gráfico:

```
paises %>%
  ggplot(aes(anio, esperanza_de_vida, group = pais)) +
  geom_line(alpha = 1 / 3)
```

Copy

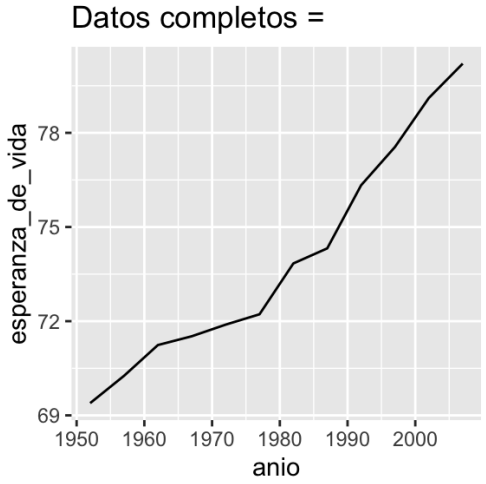


Es un conjunto de datos pequeño: solo tiene ~1,700 observaciones y 3 variables. ¡Pero aun así es difícil ver qué está pasando! En general, pareciera que la esperanza de vida ha estado mejorando en forma constante. Sin embargo, si miras de cerca, puedes notar que algunos países no siguen este patrón. ¿Cómo podemos hacer que esos países se vean más fácilmente?

Una forma es usar el mismo enfoque que en el último capítulo: hay una señal fuerte (un crecimiento lineal general) que hace difícil ver tendencias más sutiles. Separaremos estos factores estimando un modelo con una tendencia lineal. El modelo captura el crecimiento estable en el tiempo y los residuos mostrarán lo que queda fuera.

Ya sabes cómo hacer eso si tenemos un solo país:

```
nz <- filter(paises, pais == "Nueva Zelanda")
nz %>%
  ggplot(aes(anio, esperanza_de_vida)) +
  geom_line() +
  ggtitle("Datos completos = ")

nz_mod <- lm(esperanza_de_vida ~ anio, data = nz)
nz %>%
  add_predictions(nz_mod) %>%
  ggplot(aes(anio, pred)) +
  geom_line() +
  ggtitle("Tendencia lineal + ")

nz %>%
  add_residuals(nz_mod) %>%
  ggplot(aes(anio, resid)) +
  geom_hline(yintercept = 0, colour = "white", size = 3) +
  geom_line() +
  ggtitle("Patrón restante")
```



¿Cómo podemos ajustar fácilmente ese modelo para cada país?

25.2.1 **Datos anidados**

Te puedes imaginar copiando y pegando ese código múltiples veces; sin embargo, ¡ya has aprendido una mejor forma! Extrae el código en común con una función y repítelo usando una función map del paquete **purrr**. Este problema se estructura un poco diferente respecto a lo que has visto antes. En lugar de repetir una acción por cada variable, bnhgqueremos repetirla para cada país, es decir, un subconjunto de filas. Para hacer esto, necesitamos una nueva estructura de datos: el ***data frame* anidado** (*nested data frame*). Para crear un *data frame* anidado empezamos con un *data frame* agrupado, y lo "anidamos":

```
por_pais <- paises %>%
  group_by(pais, continente) %>%
  nest()

por_pais
#> # A tibble: 142 x 3
#> # Groups:   pais, continente [142]
#>   pais       continente data
#>   <fct>      <fct>      <list>
#> 1 Afganistán Asia       <tibble [12 × 4]>
#> 2 Albania    Europa     <tibble [12 × 4]>
#> 3 Argelia    África     <tibble [12 × 4]>
#> 4 Angola     África     <tibble [12 × 4]>
#> 5 Argentina  Américas   <tibble [12 × 4]>
#> 6 Australia  Oceanía    <tibble [12 × 4]>
#> # … with 136 more rows
```

(Estamos haciendo un poco de trampa agrupando por `continente` y `pais` al mismo tiempo. Dado el `pais`, `continente` es fijo, así que no agrega ningún grupo más, pero es una forma fácil de llevarnos una variable adicional para el camino.)

Esto crea un *data frame* que tiene una fila por grupo (por país), y una columna bastante inusual: `data`. `data` es una lista de *data frames* (o *tibbles*, para ser precisos). Esto parece una idea un poco loca: ¡tenemos un *data frame* con una columna que es una lista de otros *data frames*! Explicaré brevemente por qué pienso que es una buena idea.

La columna `data` es un poco difícil de examinar porque es una lista moderadamente complicada y todavía estamos trabajando para tener buenas herramientas para explorar estos objetos. Desafortunadamente, usar `str()` no es recomendable porque usualmente producirá un *output* (salida de código) muy extenso. Pero si extraes un solo elemento de la columna `data` verás que contiene todos los datos para ese país (en este caso, Afganistán).

```
por_pais$data[[1]]
#> # A tibble: 12 x 4
#>    anio esperanza_de_vida poblacion pib_per_capita
#>   <int>             <dbl>     <int>          <dbl>
#> 1  1952              28.8   8425333           779.
#> 2  1957              30.3   9240934           821.
#> 3  1962              32.0  10267083           853.
#> 4  1967              34.0  11537966           836.
#> 5  1972              36.1  13079460           740.
#> 6  1977              38.4  14880372           786.
#> # … with 6 more rows
```

Copy

Nota la diferencia entre un *data frame* agrupado estándar y un *data frame* anidado: en un *data frame* agrupado cada fila es una observación; en un *data frame* anidado, cada fila es un grupo. Otra forma de pensar en un conjunto de datos anidado es que ahora tenemos una meta-observación: una fila que representa todo el transcurso de tiempo para un país, en lugar de solo un punto en el tiempo.

25.2.2 Columnas-lista

Ahora que tenemos nuestro *data frame* anidado, estamos en una buena posición para ajustar algunos modelos. Tenemos una función para ajustar modelos:

```
modelo_pais <- function(df) {
  lm(esperanza_de_vida ~ anio, data = df)
}
```

Copy

Y queremos aplicarlo a cada *data frame*. Los *data frames* están en una lista, así que podemos usar `purrr::map()` para aplicar `modelo_pais` a cada elemento:

```
modelos <- map(por_pais$data, modelo_pais)
```

Copy

Sin embargo, en lugar de dejar la lista de modelos como un objeto suelto, lo mejor sería almacenarlo como una columna en el *data frame* `por_pais`. Almacenar objetos relacionados en columnas es una parte clave del valor de los *data frames*, y por eso creemos que las columnas-lista son tan buena idea. En el transcurso de nuestro trabajo con estos países vamos a tener muchas listas en las que tenemos un elemento por país. ¿Por qué no almacenarlos todos juntos en un *data frame*?

En otras palabras, en lugar de crear un nuevo objeto en el entorno global, vamos a crear una nueva variable en el *data frame* `por_pais`. Ese es un trabajo para `dplyr::mutate()`:

```
por_pais <- por_pais %>%
  mutate(modelo = map(data, modelo_pais))
por_pais
#> # A tibble: 142 x 4
#> # Groups:   pais, continente [142]
#>   pais       continente data              modelo
#>   <fct>      <fct>      <list>            <list>
#> 1 Afganistán Asia       <tibble [12 × 4]> <lm>
#> 2 Albania    Europa     <tibble [12 × 4]> <lm>
#> 3 Argelia    África     <tibble [12 × 4]> <lm>
#> 4 Angola     África     <tibble [12 × 4]> <lm>
#> 5 Argentina  Américas   <tibble [12 × 4]> <lm>
#> 6 Australia  Oceanía    <tibble [12 × 4]> <lm>
#> # … with 136 more rows
```

Esto tiene una gran ventaja: como todos los objetos relacionados están almacenados juntos, no necesitas manualmente mantenerlos sincronizados cuando filtras o reordenas. La semántica del *data frame* se ocupa de esto por ti:

```
por_pais %>%
  filter(continente == "Europa")
#> # A tibble: 30 x 4
#> # Groups:   pais, continente [30]
#>   pais                 continente data              modelo
#>   <fct>                <fct>      <list>            <list>
#> 1 Albania              Europa     <tibble [12 × 4]> <lm>
#> 2 Austria              Europa     <tibble [12 × 4]> <lm>
#> 3 Bélgica              Europa     <tibble [12 × 4]> <lm>
#> 4 Bosnia y Herzegovina Europa     <tibble [12 × 4]> <lm>
#> 5 Bulgaria             Europa     <tibble [12 × 4]> <lm>
#> 6 Croacia              Europa     <tibble [12 × 4]> <lm>
#> # … with 24 more rows
por_pais %>%
  arrange(continente, pais)
#> # A tibble: 142 x 4
#> # Groups:   pais, continente [142]
#>   pais         continente data              modelo
#>   <fct>        <fct>      <list>            <list>
#> 1 Argelia      África     <tibble [12 × 4]> <lm>
#> 2 Angola       África     <tibble [12 × 4]> <lm>
#> 3 Benin        África     <tibble [12 × 4]> <lm>
#> 4 Botswana     África     <tibble [12 × 4]> <lm>
#> 5 Burkina Faso África     <tibble [12 × 4]> <lm>
#> 6 Burundi      África     <tibble [12 × 4]> <lm>
#> # … with 136 more rows
```

Si tu lista de *data frames* y lista de modelos fueran objetos separados, tendrías que acordarte de que cuando reordenas o seleccionas un subconjunto de un vector, necesitas reordenar o seleccionar el subconjunto de todos los demás para mantenerlos sincronizados. Si te olvidas, tu código va a seguir funcionando, ¡pero va a devolver la respuesta equivocada!

25.2.3 **Desanidando**

Previamente calculamos los residuos de un único modelo con un conjunto de datos también único. Ahora tenemos 142 *data frames* y 142 modelos. Para calcular los residuos, necesitamos llamar a la función `add_residuals()` (*adicionar residuos*, en inglés) con cada par modelo-datos:

```
por_pais <- por_pais %>%
  mutate(
    residuos = map2(data, modelo, add_residuals)
  )
por_pais
#> # A tibble: 142 x 5
#> # Groups:   pais, continente [142]
#>   pais       continente data              modelo residuos
#>   <fct>      <fct>      <list>            <list> <list>
#> 1 Afganistán Asia       <tibble [12 × 4]> <lm>   <tibble [12 × 5]>
#> 2 Albania    Europa     <tibble [12 × 4]> <lm>   <tibble [12 × 5]>
#> 3 Argelia    África     <tibble [12 × 4]> <lm>   <tibble [12 × 5]>
#> 4 Angola     África     <tibble [12 × 4]> <lm>   <tibble [12 × 5]>
#> 5 Argentina  Américas   <tibble [12 × 4]> <lm>   <tibble [12 × 5]>
#> 6 Australia  Oceanía    <tibble [12 × 4]> <lm>   <tibble [12 × 5]>
#> # … with 136 more rows
```

¿Pero cómo puedes graficar una lista de *data frames*? En lugar de luchar para contestar esa pregunta, transformemos la lista de *data frames* de vuelta en un *data frame* regular. Previamente usamos `nest()` (*anidar*) para transformar un *data frame* regular en uno anidado; ahora desanidaremos con `unnest()`:

```
residuos <- unnest(por_pais, residuos)
residuos
#> # A tibble: 1,704 x 9
#> # Groups:   pais, continente [142]
#>   pais  continente data  modelo  anio esperanza_de_vi… poblacion pib_per_capita
#>   <fct> <fct>      <lis> <list> <int>            <dbl>     <int>          <dbl>
#> 1 Afga… Asia       <tib… <lm>    1952             28.8   8425333           779.
#> 2 Afga… Asia       <tib… <lm>    1957             30.3   9240934           821.
#> 3 Afga… Asia       <tib… <lm>    1962             32.0  10267083           853.
#> 4 Afga… Asia       <tib… <lm>    1967             34.0  11537966           836.
#> 5 Afga… Asia       <tib… <lm>    1972             36.1  13079460           740.
#> 6 Afga… Asia       <tib… <lm>    1977             38.4  14880372           786.
#> # … with 1,698 more rows, and 1 more variable: resid <dbl>
```

Nota que cada columna regular está repetida una vez por cada fila en la columna anidada.

Ahora que tenemos un *data frame* regular, podemos graficar los residuos:

```
residuos %>%
  ggplot(aes(anio, resid)) +
  geom_line(aes(group = pais), alpha = 1 / 3) +
  geom_smooth(se = FALSE)
#> `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```

Separar facetas por continente es particularmente revelador:

```
residuos %>%
  ggplot(aes(anio, resid, group = pais)) +
  geom_line(alpha = 1 / 3) +
  facet_wrap(~continente)
```

Parece que hemos perdido algunos patrones suaves. También hay algo interesante pasando en África: vemos algunos residuos muy grandes, lo que sugiere que nuestro modelo no está ajustando muy bien. Exploraremos más eso en la próxima sección, atacando el problema desde un ángulo un poco diferente.

25.2.4 **Calidad del modelo**

En lugar de examinar los residuos del modelo, podríamos examinar algunas medidas generales de la calidad del modelo. Aprendiste cómo calcular algunas medidas específicas en el capítulo anterior. Aquí mostraremos un enfoque diferente usando el paquete **broom**. El paquete **broom** provee un conjunto de

funciones generales para transformar modelos en datos ordenados. Aquí utilizaremos `broom::glance()` (*vistazo*, en inglés) para extraer algunas métricas de la calidad del modelo. Si lo aplicamos a un modelo, obtenemos un *data frame* con una única fila:

```
broom::glance(nz_mod)
#> # A tibble: 1 x 12
#>   r.squared adj.r.squared sigma statistic p.value    df logLik   AIC   BIC
#>       <dbl>         <dbl> <dbl>     <dbl>   <dbl> <dbl>  <dbl> <dbl> <dbl>
#> 1     0.954         0.949 0.804      205. 5.41e-8     1  -13.3  32.6  34.1
#> # … with 3 more variables: deviance <dbl>, df.residual <int>, nobs <int>
```

Podemos usar `mutate()` y `unnest()` para crear un *data frame* con una fila por cada país:

```
por_pais %>%
  mutate(glance = map(modelo, broom::glance)) %>%
  unnest(glance)
#> # A tibble: 142 x 17
#> # Groups:   pais, continente [142]
#>   pais  continente data  modelo residuos r.squared adj.r.squared sigma statistic
#>   <fct> <fct>      <lis> <list> <list>       <dbl>         <dbl> <dbl>     <dbl>
#> 1 Afga… Asia       <tib… <lm>   <tibble…     0.948         0.942 1.22      181.
#> 2 Alba… Europa     <tib… <lm>   <tibble…     0.911         0.902 1.98      102.
#> 3 Arge… África     <tib… <lm>   <tibble…     0.985         0.984 1.32      662.
#> 4 Ango… África     <tib… <lm>   <tibble…     0.888         0.877 1.41       79.1
#> 5 Arge… Américas   <tib… <lm>   <tibble…     0.996         0.995 0.292    2246.
#> 6 Aust… Oceanía    <tib… <lm>   <tibble…     0.980         0.978 0.621     481.
#> # … with 136 more rows, and 8 more variables: p.value <dbl>, df <dbl>,
#> #   logLik <dbl>, AIC <dbl>, BIC <dbl>, deviance <dbl>, df.residual <int>,
#> #   nobs <int>
```

Este no es exactamente el *output* que queremos, porque aún incluye todas las columnas que son una lista. Este es el comportamiento por defecto cuando `unnest()` trabaja sobre *data frames* con una única fila. Para suprimir esas columnas usamos `.drop = TRUE` (`drop` = *descartar*, en inglés):

```
glance <- por_pais %>%
  mutate(glance = map(modelo, broom::glance)) %>%
  unnest(glance, .drop = TRUE)
#> Warning: The `.drop` argument of `unnest()` is deprecated as of tidyr 1.0.0.
#> All list-columns are now preserved.
#> This warning is displayed once every 8 hours.
#> Call `lifecycle::last_warnings()` to see where this warning was generated.
glance
#> # A tibble: 142 x 17
#> # Groups:   pais, continente [142]
#>   pais  continente data  modelo residuos r.squared adj.r.squared sigma statistic
#>   <fct> <fct>      <lis> <list> <list>       <dbl>         <dbl> <dbl>     <dbl>
#> 1 Afga… Asia       <tib… <lm>   <tibble…     0.948         0.942 1.22      181.
#> 2 Alba… Europa     <tib… <lm>   <tibble…     0.911         0.902 1.98      102.
#> 3 Arge… África     <tib… <lm>   <tibble…     0.985         0.984 1.32      662.
#> 4 Ango… África     <tib… <lm>   <tibble…     0.888         0.877 1.41       79.1
#> 5 Arge… Américas   <tib… <lm>   <tibble…     0.996         0.995 0.292    2246.
#> 6 Aust… Oceanía    <tib… <lm>   <tibble…     0.980         0.978 0.621     481.
#> # … with 136 more rows, and 8 more variables: p.value <dbl>, df <dbl>,
#> #   logLik <dbl>, AIC <dbl>, BIC <dbl>, deviance <dbl>, df.residual <int>,
#> #   nobs <int>
```

(Presta atención a las variables que no se imprimieron: hay mucha información útil allí).

Con este *data frame* podemos empezar a buscar modelos que no se ajustan bien:

```
glance %>%
  arrange(r.squared)
#> # A tibble: 142 x 17
#> # Groups:   pais, continente [142]
#>   pais  continente data  modelo residuos r.squared adj.r.squared sigma statistic
#>   <fct> <fct>      <lis> <list> <list>       <dbl>         <dbl> <dbl>     <dbl>
#> 1 Ruan… África     <tib… <lm>   <tibble…    0.0172      -0.0811   6.56     0.175
#> 2 Bots… África     <tib… <lm>   <tibble…    0.0340      -0.0626   6.11     0.352
#> 3 Zimb… África     <tib… <lm>   <tibble…    0.0562      -0.0381   7.21     0.596
#> 4 Zamb… África     <tib… <lm>   <tibble…    0.0598      -0.0342   4.53     0.636
#> 5 Swaz… África     <tib… <lm>   <tibble…    0.0682      -0.0250   6.64     0.732
#> 6 Leso… África     <tib… <lm>   <tibble…    0.0849      -0.00666  5.93     0.927
#> # … with 136 more rows, and 8 more variables: p.value <dbl>, df <dbl>,
#> #   logLik <dbl>, AIC <dbl>, BIC <dbl>, deviance <dbl>, df.residual <int>,
#> #   nobs <int>
```

Copy

Los peores modelos parecieran estar todos en África. Vamos a chequear esto con un gráfico. Tenemos un número relativamente chico de observaciones y una variable discreta, así que usar `geom_jitter()` es efectivo:

```
glance %>%
  ggplot(aes(continente, r.squared)) +
  geom_jitter(width = 0.5)
```

Copy

Podríamos quitar los países con un R^2 particularmente malo y graficar los datos:

```
mal_ajuste <- filter(glance, r.squared < 0.25)

paises %>%
  semi_join(mal_ajuste, by = "pais") %>%
  ggplot(aes(anio, esperanza_de_vida, colour = pais)) +
  geom_line()
```

Copy

Vemos dos efectos principales aquí: las tragedias de la epidemia de VIH/SIDA y el genocidio de Ruanda.

25.2.5 Ejercicios

1. Una tendencia lineal parece ser demasiado simple para la tendencia general. ¿Puedes hacerlo mejor con un polinomio cuadrático? ¿Cómo puedes interpretar el coeficiente del término cuadrático? (Pista: puedes querer transformar `year` para que tenga media cero.)

2. Explora otros métodos para visualizar la distribución del R^2 por continente. Puedes querer probar el paquete **ggbeeswarm**, que provee métodos similares para evitar superposiciones como jitter, pero usa métodos determinísticos.

3. Para crear el último gráfico (mostrando los datos para los países con los peores ajustes del modelo), precisamos dos pasos: creamos un *data frame* con una fila por país y después hicimos un *semi-join* al conjunto de datos original. Es posible evitar este *join* si usamos `unnest()` en lugar de `unnest(.drop = TRUE)`. ¿Cómo?

25.3 Columnas-lista

Ahora que has visto un flujo de trabajo básico para manejar muchos modelos, vamos a sumergirnos en algunos detalles. En esta sección, exploraremos en más detalle la estructura de datos columna-lista. Solo recientemente hemos comenzado a apreciar realmente la idea de la columna-lista. Esta estructura está implícita en la definición de *data frame*: un *data frame* es una lista nombrada de vectores de igual largo. Una lista es un vector, así que siempre ha sido legítimo usar una lista como una columna de un *data frame*. Sin embargo, R base no hace las cosas fáciles para crear columnas-lista, y `data.frame()` trata a la lista como una lista de columnas:

```
data.frame(x = list(1:3, 3:5))
#>   x.1.3 x.3.5
#> 1     1     3
#> 2     2     4
#> 3     3     5
```

Puedes prevenir que `data.frame()` haga esto con `I()`, pero el resultado no se imprime particularmente bien:

```
data.frame(
  x = I(list(1:3, 3:5)),
  y = c("1, 2", "3, 4, 5")
)
#>         x       y
#> 1 1, 2, 3    1, 2
#> 2 3, 4, 5 3, 4, 5
```

Tibble mitiga este problema siendo más *perezoso* (`tibble()` no modifica sus *inputs*) y proporcionando un mejor método de impresión:

```
tibble(
  x = list(1:3, 3:5),
  y = c("1, 2", "3, 4, 5")
)
#> # A tibble: 2 x 2
#>   x         y
#>   <list>    <chr>
#> 1 <int [3]> 1, 2
#> 2 <int [3]> 3, 4, 5
```

Es incluso más fácil con `tribble()`, ya que automáticamente puede interpretar que necesitas una lista:

```
tribble(
  ~x, ~y,
  1:3, "1, 2",
  3:5, "3, 4, 5"
)
#> # A tibble: 2 x 2
#>   x         y
#>   <list>    <chr>
#> 1 <int [3]> 1, 2
#> 2 <int [3]> 3, 4, 5
```

Las columnas-lista son usualmente más útiles como estructuras de datos intermedias. Es difícil trabajar con ellas directamente, porque la mayoría de las funciones de R trabaja con vectores atómicos o *data frames*, pero la ventaja de mantener ítems relacionados juntos en un *data frame* hace que valga la pena un poco de molestia.

Generalmente hay tres partes en un *pipeline* efectivo de columnas-lista:

1. Creas la columna-lista usando alguna de estas opciones: `nest()` o `summarise()` + `list()` o `mutate()` + una función map, como se describe en Creando columnas-lista.

2. Creas otra columna-lista intermedia transformando columnas lista existentes con `map()`, `map2()` o `pmap()`. Por ejemplo, en el caso de estudio de arriba, creamos una columna-lista de modelos transformando una columna-lista de *data frames*.

3. Simplificas la columna-lista de vuelta en un *data frame* o vector atómico, como se describe en Simplificando columnas-lista.

25.4 Creando columnas-lista

Típicamente, no tendrás que crear columnas-lista con `tibble()`, sino a partir de columnas regulares usando uno de estos tres métodos:

1. Con `tidyr::nest()` para convertir un *data frame* agrupado en uno anidado en el que tengas columnas-lista de *data frames*.

2. Con `mutate()` y funciones vectorizadas que retornan una lista.

3. Con `summarise()` y funciones de resumen que retornan múltiples resultados.

Alternativamente, podrías crearlas a partir de una lista nombrada, usando `tibble::enframe()`.

Generalmente, cuando creas columnas-lista, debes asegurarte de que sean homogéneas: cada elemento debe contener el mismo tipo de cosa. No hay chequeos para asegurarte de que sea así, pero si usas **purrr** y recuerdas lo que aprendiste sobre funciones de tipo estable (*type-stable functions*), encontrarás que eso pasa naturalmente.

25.4.1 Con anidación

`nest()` crea un *data frame* anidado, que es un *data frame* con una columna-lista de *data frames*. En un *data frame* anidado cada fila es una meta-observación: las otras columnas son variables que definen la observación (como país y continente arriba), y la columna-lista de *data frames* tiene las observaciones individuales que construyen la meta-observación.

Hay dos formas de usar `nest()`. Hasta ahora has visto cómo usarlo con un *data frame* agrupado. Cuando se aplica a un *data frame* agrupado, `nest()` mantiene las columnas que agrupan tal cual, y envuelve todo lo demás en la columna-lista:

```
paises %>%
  group_by(pais, continente) %>%
  nest()
#> # A tibble: 142 x 3
#> # Groups:   pais, continente [142]
#>   pais       continente data
#>   <fct>      <fct>      <list>
#> 1 Afganistán Asia       <tibble [12 × 4]>
#> 2 Albania    Europa     <tibble [12 × 4]>
#> 3 Argelia    África     <tibble [12 × 4]>
#> 4 Angola     África     <tibble [12 × 4]>
#> 5 Argentina  Américas   <tibble [12 × 4]>
#> 6 Australia  Oceanía    <tibble [12 × 4]>
#> # … with 136 more rows
```

También lo puedes usar en un *data frame* no agrupado, especificando cuáles columnas quieres anidar:

```
paises %>%
  nest(anio:pib_per_capita)
#> Warning: All elements of `...` must be named.
#> Did you want `data = c(anio, esperanza_de_vida, poblacion, pib_per_capita)`?
#> # A tibble: 142 x 3
#>   pais       continente data
#>   <fct>      <fct>      <list>
#> 1 Afganistán Asia       <tibble [12 × 4]>
#> 2 Albania    Europa     <tibble [12 × 4]>
#> 3 Argelia    África     <tibble [12 × 4]>
#> 4 Angola     África     <tibble [12 × 4]>
#> 5 Argentina  Américas   <tibble [12 × 4]>
#> 6 Australia  Oceanía    <tibble [12 × 4]>
#> # … with 136 more rows
```

25.4.2 A partir de funciones vectorizadas

Algunas funciones útiles toman un vector atómico y retornan una lista. Por ejemplo, en cadenas de caracteres aprendiste sobre `stringr::str_split()`, que toma un vector de caracteres y retorna una lista de vectores de caracteres. Si lo usas dentro de `mutate`, obtendrás una columna-lista:

```r
df <- tribble(
  ~x1,
  "a,b,c",
  "d,e,f,g"
)

df %>%
  mutate(x2 = stringr::str_split(x1, ","))
#> # A tibble: 2 x 2
#>   x1      x2
#>   <chr>   <list>
#> 1 a,b,c   <chr [3]>
#> 2 d,e,f,g <chr [4]>
```

`unnest()` sabe cómo manejar estas listas de vectores:

```r
df %>%
  mutate(x2 = stringr::str_split(x1, ",")) %>%
  unnest()
#> Warning: `cols` is now required when using unnest().
#> Please use `cols = c(x2)`
#> # A tibble: 7 x 2
#>   x1      x2
#>   <chr>   <chr>
#> 1 a,b,c   a
#> 2 a,b,c   b
#> 3 a,b,c   c
#> 4 d,e,f,g d
#> 5 d,e,f,g e
#> 6 d,e,f,g f
#> # … with 1 more row
```

(Si usas mucho este patrón, asegúrate de chequear `tidyr::separate_rows()`, que es un *wrapper* alrededor de este patrón común).

Otro ejemplo de este patrón es usar `map()`, `map2()`, `pmap()` de **purrr**. Por ejemplo, podríamos tomar el ejemplo final de [Invocando distintas functions] y reescribirlo usando `mutate()`:

```r
sim <- tribble(
  ~f, ~params,
  "runif", list(min = -1, max = 1),
  "rnorm", list(sd = 5),
  "rpois", list(lambda = 10)
)

sim %>%
  mutate(sims = invoke_map(f, params, n = 10))
#> # A tibble: 3 x 3
#>   f     params           sims
#>   <chr> <list>           <list>
#> 1 runif <named list [2]> <dbl [10]>
#> 2 rnorm <named list [1]> <dbl [10]>
#> 3 rpois <named list [1]> <int [10]>
```

Nota que técnicamente `sim` no es homogéneo porque contiene tanto vectores de dobles como vectores de enteros. Sin embargo, es probable que esto no cause muchos problemas porque ambos vectores son numéricos.

25.4.3 A partir de medidas de resumen con más de un valor

Una restricción de `summarise()` es que solo funciona con funciones de resumen que retornan un único valor. Eso significa que no puedes usarlo con funciones como `quantile()`, que retorna un vector de largo arbitrario:

```
mtautos %>%
  group_by(cilindros) %>%
  summarise(q = quantile(millas))
#> `summarise()` has grouped output by 'cilindros'. You can override using the `.groups`
argument.
#> # A tibble: 15 x 2
#> # Groups:   cilindros [3]
#>   cilindros     q
#>       <dbl> <dbl>
#> 1         4  21.4
#> 2         4  22.8
#> 3         4  26
#> 4         4  30.4
#> 5         4  33.9
#> 6         6  17.8
#> # … with 9 more rows
```

Sin embargo, ¡puedes envolver el resultado en una lista! Esto obedece el contrato de `summarise()`, porque cada resumen ahora es una lista (un vector) de largo 1.

```
mtautos %>%
  group_by(cilindros) %>%
  summarise(q = list(quantile(millas)))
#> # A tibble: 3 x 2
#>   cilindros q
#> *     <dbl> <list>
#> 1         4 <dbl [5]>
#> 2         6 <dbl [5]>
#> 3         8 <dbl [5]>
```

Para producir resultados útiles con `unnest`, también necesitarás capturar las probabilidades:

```
probs <- c(0.01, 0.25, 0.5, 0.75, 0.99)
mtautos %>%
  group_by(cilindros) %>%
  summarise(p = list(probs), q = list(quantile(millas, probs))) %>%
  unnest()
#> Warning: `cols` is now required when using unnest().
#> Please use `cols = c(p, q)`
#> # A tibble: 15 x 3
#>   cilindros     p     q
#>       <dbl> <dbl> <dbl>
#> 1         4  0.01  21.4
#> 2         4  0.25  22.8
#> 3         4  0.5   26
#> 4         4  0.75  30.4
#> 5         4  0.99  33.8
#> 6         6  0.01  17.8
#> # … with 9 more rows
```

25.4.4 A partir de una lista nombrada

Los *data frames* con columnas-lista proveen una solución a un problema común: ¿qué haces si quieres iterar sobre el contenido de una lista y también sobre sus elementos? En lugar de tratar de juntar todo en un único objeto, usualmente es más fácil hacer un *data frame*: una columna puede contener los elementos y otra columna la lista. Una forma fácil de crear un *data frame* como este desde una lista es `tibble::enframe()`.

```
x <- list(
  a = 1:5,
  b = 3:4,
  c = 5:6
)

df <- enframe(x)
df
#> # A tibble: 3 x 2
#>   name  value
#>   <chr> <list>
#> 1 a     <int [5]>
#> 2 b     <int [2]>
#> 3 c     <int [2]>
```

La ventaja de esta estructura es que se generaliza de una forma relativamente sencilla - los nombres son útiles si tienes un vector de caracteres con los metadatos, pero no ayudan para otros tipos de datos o para múltiples vectores.

Ahora, si quieres iterar sobre los nombres y valores en paralelo, puedes usar `map2()`:

```
df %>%
  mutate(
    smry = map2_chr(name, value, ~ stringr::str_c(.x, ": ", .y[1]))
  )
#> # A tibble: 3 x 3
#>   name  value     smry
#>   <chr> <list>    <chr>
#> 1 a     <int [5]> a: 1
#> 2 b     <int [2]> b: 3
#> 3 c     <int [2]> c: 5
```

25.4.5 Ejercicios

1. Lista todas las funciones en las que puedas pensar que tomen como *input* un vector atómico y retornen una lista.

2. Piensa en funciones de resumen útiles que, como `quantile()`, retornen múltiples valores.

3. ¿Qué es lo que falta en el siguiente *data frame*? ¿Cómo `quantile()` retorna eso que falta? ¿Por qué eso no es tan útil aquí?

```
mtautos %>%
  group_by(cilindros) %>%
  summarise(q = list(quantile(millas))) %>%
  unnest()
#> Warning: `cols` is now required when using unnest().
#> Please use `cols = c(q)`
#> # A tibble: 15 x 2
#>   cilindros     q
#>       <dbl> <dbl>
#> 1         4  21.4
#> 2         4  22.8
#> 3         4  26
#> 4         4  30.4
#> 5         4  33.9
#> 6         6  17.8
#> # … with 9 more rows
```

1. ¿Qué hace este código? ¿Por qué podría ser útil?

```
mtautos %>%
  group_by(cilindros) %>%
  summarise_each(funs(list))
```

25.5 Simplificando columnas-lista

Para aplicar las técnicas de manipulación de datos y visualización que has aprendido en este libro, necesitarás simplificar la columna-lista de vuelta a una columna regular (un vector atómico) o conjunto de columnas. La técnica que usarás para volver a una estructura más sencilla depende de si quieres un único valor por elemento, o múltiples valores.

1. Si quieres un único valor, usa `mutate()` con `map_lgl()`, `map_int()`, `map_dbl()`, y `map_chr()` para crear un vector atómico.

2. Si quieres varios valores, usa `unnest()` para convertir columnas-lista de vuelta a columnas regulares, repitiendo las filas tantas veces como sea necesario.

Estas técnicas están descritas con más detalle abajo.

25.5.1 Lista a vector

Si puedes reducir tu columna lista a un vector atómico entonces será una columna regular. Por ejemplo, siempre puedes resumir un objeto con su tipo y largo, por lo que este código funcionará sin importar cuál tipo de columna-lista tengas:

```
df <- tribble(
  ~x,
  letters[1:5],
  1:3,
  runif(5)
)

df %>% mutate(
  tipo = map_chr(x, typeof),
  largo = map_int(x, length)
)
#> # A tibble: 3 x 3
#>   x         tipo      largo
#>   <list>    <chr>     <int>
#> 1 <chr [5]> character     5
#> 2 <int [3]> integer       3
#> 3 <dbl [5]> double        5
```

Esta es la misma información básica que obtienes del método de impresión por defecto de *tbl* , solo que ahora lo puedes usar para filtrar. Es una técnica útil si tienes listas heterogéneas y quieres remover las partes que no te sirven.

No te olvides de los atajos de `map_*()` - puedes usar `map_chr(x, "manzana")` para extraer la cadena de caracteres almacenada en `manzana` para cada elemento de `x`. Esto es útil para separar listas anidadas en columnas regulares. Usa el argumento `.null` para proveer un valor para usar si el elemento es un valor faltante (en lugar de retornar `NULL`):

```
df <- tribble(
  ~x,
  list(a = 1, b = 2),
  list(a = 2, c = 4)
)
df %>% mutate(
  a = map_dbl(x, "a"),
  b = map_dbl(x, "b", .null = NA_real_)
)
#> # A tibble: 2 x 3
#>   x                   a     b
#>   <list>          <dbl> <dbl>
#> 1 <named list [2]>     1     2
#> 2 <named list [2]>     2    NA
```

25.5.2 Desanidando

`unnest()` trabaja repitiendo la columna regular una vez para cada elemento de la columna-lista. Por ejemplo, en el siguiente ejemplo sencillo repetimos la primera fila 4 veces (porque el primer elemento de `y` tiene largo cuatro) y la segunda fila una vez:

```
tibble(x = 1:2, y = list(1:4, 1)) %>% unnest(y)
#> # A tibble: 5 x 2
#>       x     y
#>   <int> <dbl>
#> 1     1     1
#> 2     1     2
#> 3     1     3
#> 4     1     4
#> 5     2     1
```

Esto significa que no puedes simultáneamente desanidar dos columnas que contengan un número diferente de elementos:

```
# Funciona, porque y y z tienen el mismo número de elementos en
# cada fila
df1 <- tribble(
  ~x, ~y, ~z,
  1, c("a", "b"), 1:2,
  2, "c", 3
)
df1
#> # A tibble: 2 x 3
#>       x y         z
#>   <dbl> <list>    <list>
#> 1     1 <chr [2]> <int [2]>
#> 2     2 <chr [1]> <dbl [1]>
df1 %>% unnest(y, z)
#> Warning: unnest() has a new interface. See ?unnest for details.
#> Try `df %>% unnest(c(y, z))`, with `mutate()` if needed
#> # A tibble: 3 x 3
#>       x y         z
#>   <dbl> <chr> <dbl>
#> 1     1 a         1
#> 2     1 b         2
#> 3     2 c         3

# No funciona porque y y z tienen un número diferente de elementos
df2 <- tribble(
  ~x, ~y, ~z,
  1, "a", 1:2,
  2, c("b", "c"), 3
)
df2
#> # A tibble: 2 x 3
#>       x y         z
#>   <dbl> <list>    <list>
#> 1     1 <chr [1]> <int [2]>
#> 2     2 <chr [2]> <dbl [1]>
df2 %>% unnest(y, z)
#> Warning: unnest() has a new interface. See ?unnest for details.
#> Try `df %>% unnest(c(y, z))`, with `mutate()` if needed
#> # A tibble: 4 x 3
#>       x y         z
#>   <dbl> <chr> <dbl>
#> 1     1 a         1
#> 2     1 a         2
#> 3     2 b         3
#> 4     2 c         3
```

El mismo principio aplica al desanidar columnas-lista de *data frames*. Puedes desanidar múltiples columnas-lista siempre que todos los *data frames* de cada fila tengan la misma cantidad de filas.

25.5.3 Ejercicios

1. ¿Por qué podría ser útil la función `lengths()` para crear columnas de vectores atómicos a partir de columnas-lista?

2. Lista los tipos de vectores más comúnes que se encuentran en un *data frame*. ¿Qué hace que las listas sean diferentes?

25.6 Haciendo datos ordenados con broom

El paquete **broom** provee tres herramientas generales para transformar modelos en *data frames* ordenados:

1. `broom::glance(modelo)` retorna una fila para cada modelo. Cada columna tiene una medida de resumen del modelo: o bien una medida de la calidad del modelo, o bien complejidad, o una combinación de ambos.

2. `broom::tidy(modelo)` retorna una fila por cada coeficiente en el modelo. Cada columna brinda información acerca de la estimación o su variabilidad.

3. `broom::augment(modelo, data)` retorna una fila por cada fila en `data`, agregando valores adicionales como residuos, y estadísticos de influencia.

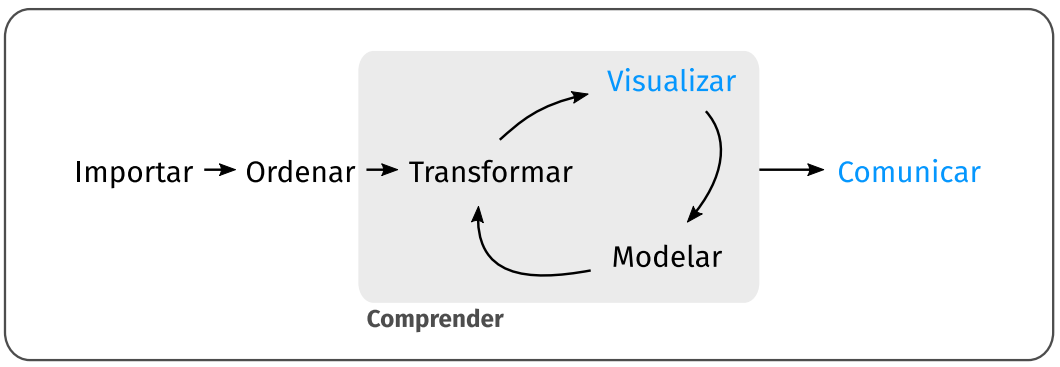
« 24 Construcción de modelos

26 Introducción »

26 Introducción

Hasta aquí, hemos aprendido a usar las herramientas para importar tus datos en R, ordenarlos de una manera conveniente para el análisis, y luego interpretarlos a través de su transformación, visualización y modelado. Sin embargo, no importa lo bien que esté hecho tu análisis si no puedes explicarlo de manera sencilla a otros: es decir, que es necesario comunicar tus resultados.

La comunicación de resultados es el tema de los siguientes cuatro capítulos.



- En [el capítulo: R Markdown], aprenderás sobre dicho paquete, el cual es una herramienta para integrar texto, código y resultados. Puedes usarlo en modo notebook, es decir, en un entorno interactivo de ejecución de código para la comunicación de analista-a-analista, y en modo reporte para la comunicación de analista-a-tomadores-de-decisión. Gracias al potencial de los formatos de R Markdown, incluso puedes usar el mismo documento para ambos propósitos.

- En [el capítulo: Gráficos para la comunicación], aprenderás cómo convertir tus gráficos exploratorios en gráficos explicativos, los cuales ayudarán a quien ve tu análisis por primera vez a comprender de qué se trata de manera fácil y sencilla.

- En [el capítulo: Formatos de R Markdown], aprenderás un poco sobre la gran variedad de salidas que puedes generar usando la librería R Markdown, incluyendo dashboards (tableros de control), sitios web, y libros.

- Terminaremos con [el flujo de trabajo de R Markdown], donde aprenderás sobre “analysis notebook”, en otras palabras, aprenderás sobre el modo notebook para realizar el análisis y registrar sistemáticamente tus logros y fallas para que puedas aprender de ellos.

Desafortunadamente, estos capítulos se enfocan principalmente en la parte técnica de la comunicación, y no en los verdaderos grandes problemas de comunicar tus pensamientos a otras personas. Sin embargo, existe una gran cantidad de excelentes libros que abordan esta problemática, cuyas referencias estarán disponibles al final de cada capítulo.

« 25 Muchos modelos

27 R Markdown »

View source

Edit this page

27 R Markdown

27.1 Introducción

R Markdown provee un marco de escritura para ciencia de datos, que combina tu código, sus resultados y tus comentarios en prosa. Los documentos de R Markdown son completamente reproducibles y soportan docenas de formatos de salida tales como PDFs, archivos de Word, presentaciones y más.

Los archivos R Markdown están diseñados para ser usados de tres maneras:

1. Para comunicarse con quienes toman decisiones, que desean enfocarse en las conclusiones, no en el código que subyace al análisis.
2. Para colaborar con otras personas que hacen ciencia de datosdatos (¡incluyendo a tu yo futuro!), quienes están interesados tanto en tus conclusiones como en el modo en el que llegaste a ellas (es decir, el código).
3. Como un ambiente en el cual *hacer* ciencia de datos, como si fuera un *notebook* de laboratorio moderno donde puedes capturar no solo que hiciste, sino también lo que estabas pensando cuando lo hacías.

R Markdown integra una cantidad de paquetes de R y herramientas externas. Esto implica que la ayuda, en general, no está disponible a través de `?`. En su lugar, a lo largo de este capítulo y cuando utilices R Markdown en el futuro, mantén estos recursos cerca:

- Hoja de referencia de R Markdown : *Help* > *Cheatsheets* > *R Markdown Cheat Sheet*
- Guía de referencia R Markdown : *Help* > *Cheatsheets* > *R Markdown Reference Guide*

Ambas hojas también se encuentran disponibles en https://rstudio.com/resources/cheatsheets/>.

27.1.1 Prerrequisitos

Si bien necesitas el paquete **rmarkdown**, no necesitas cargarlo o instalarlo explícitamente, ya que RStudio hace ambas acciones de forma automática cuando es necesario.

27.2 Elementos básicos de R Markdown

Este es un archivo R Markdown, un archivo de texto plano que tiene la extensión `.Rmd`:

````
---
title: "Tamaño de los diamantes"
date: 2016-08-25
output: html_document
---

```{r setup, include = FALSE}
library(datos)
library(ggplot2)
library(dplyr)

pequenios <- diamantes %>%
  filter(quilate <= 2.5)
```

Tenemos datos respecto de `r nrow(diamantes)` diamantes. Únicamente
`r nrow(diamantes) - nrow(pequenios)` son mayores a 2,5 quilates.
La distribución de los diamantes pequeños se muestra a continuación:

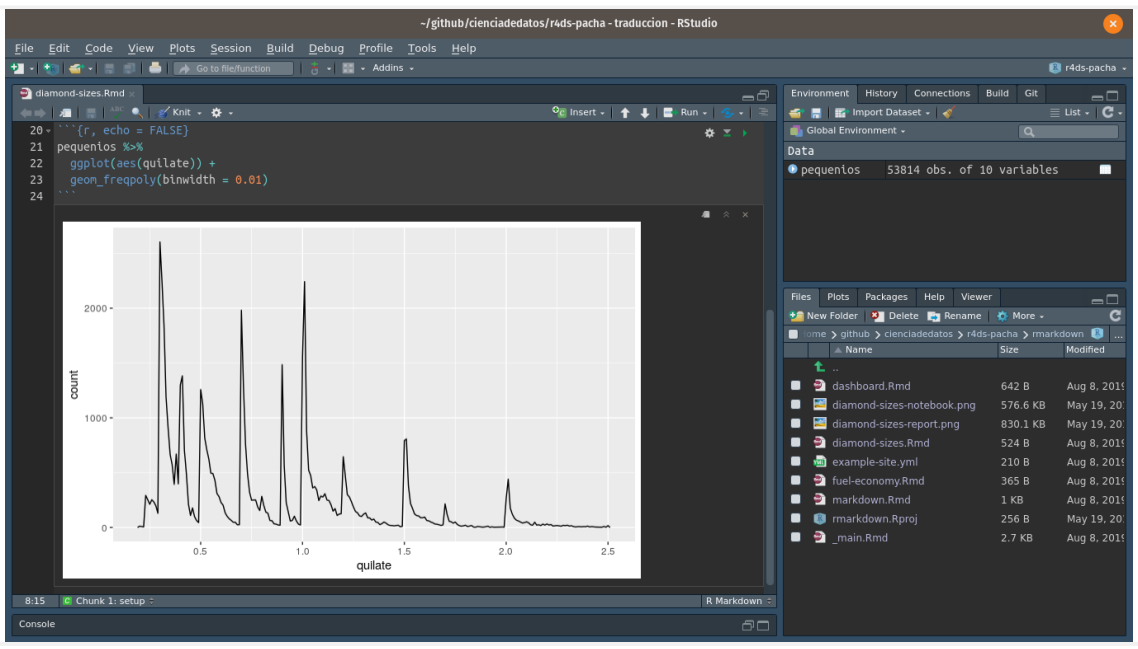
```{r, echo = FALSE}
pequenios %>%
  ggplot(aes(quilate)) +
  geom_freqpoly(binwidth = 0.01)
```
````

Copy

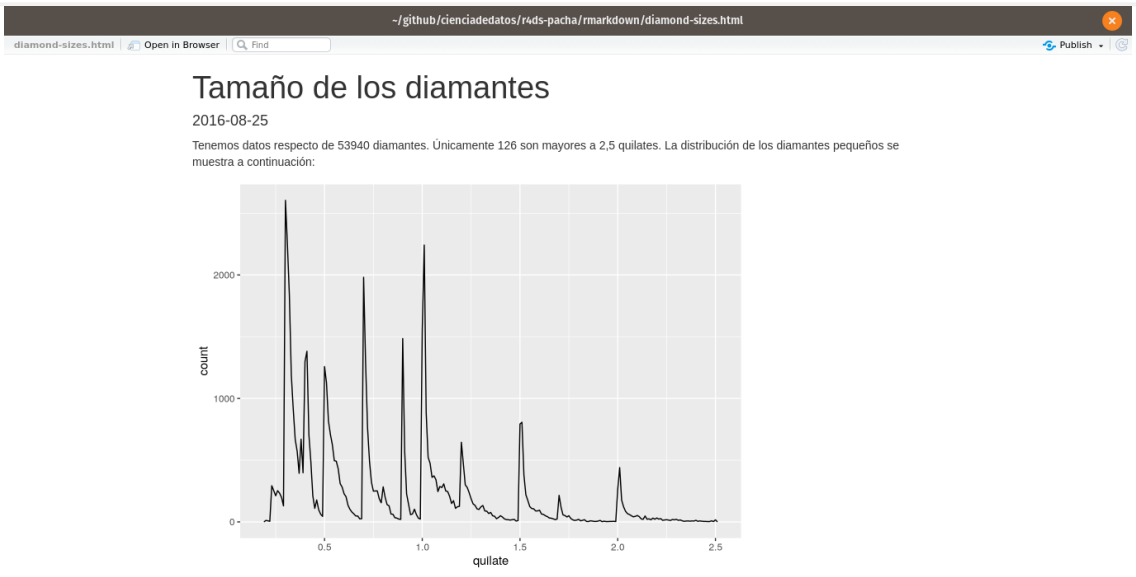
Contiene tres tipos importantes de contenido:

1. Un encabezado YAML (opcional) rodeado de `---`
2. **Bloques** de código de R rodeados de ```` ``` ````.
3. Texto mezclado con formateos de texto simple como `# Encabezado` e `_itálicas_`.

Cuando abres un archivo `.Rmd`, obtienes una interfaz de *notebook* donde el código y el *output* están intercalados. Puedes ejecutar cada bloque de código haciendo clic en el ícono ejecutar (se parece a un botón de reproducir en la parte superior del bloque de código), o presionando Cmd/Ctrl + Shift + Enter. RStudio ejecuta el código y muestra los resultados incrustados en el código:



Para producir un reporte completo que contenga todo el texto, código y resultados, haz clic en “Knit” o presionar Cmd/Ctrl + Shift + K. Puede hacerse también de manera programática con `rmarkdown::render("1-example.Rmd")`. Esto mostrará el reporte en el panel *viewer* y creará un archivo HTML independiente que puedes compartir con otras personas.



Cuando haces *knit* el documento (*knit* significa tejer en inglés), R Markdown envía el .Rmd a *knitr* (http://yihui.name/knitr/) que ejecuta todos los bloques de código y crea un nuevo documento markdown (.md) que incluye el código y su output. El archivo markdown generado por *knitr* es procesado entonces por pandoc (http://pandoc.org/) que es el responsable de crear el archivo terminado. La ventaja de este flujo de trabajo en dos pasos es que puedes crear un muy amplio rango de formatos de salida, como aprenderás en Formatos de R markdown.

Para comenzar con tu propio archivo `.Rmd`, selecciona *File* > *New File* > *R Markdown...* en la barra de menú. RStudio iniciará un asistente que puedes usar para pre-rellenar tu archivo con contenido útil que te recuerda cómo funcionan las principales características de R Markdown.

Las siguientes secciones profundizan en los tres componentes de un documento de R Markdown en más detalle: el texto Markdown, los bloques de código y el encabezado YAML.

27.2.1 Ejercicios

1. Crea un nuevo *notebook* usando *File* > *New File* > *R Notebook*. Lee las instrucciones. Practica ejecutando los bloques. Verifica que puedes modificar el código, re-ejecútalo, y observa la salida modificada.

2. Crea un nuevo documento R Markdown con *File* > *New File* > *R Markdown...* Haz clic en el icono apropiado de *Knit*. Haz *Knit* usando el atajo de teclado apropiado. Verifica que puedes modificar el *input* y la actualización del *output*.

3. Compara y contrasta el *notebook* de R con los archivos de R markdown que has creado antes. ¿Cómo son similares los outputs? ¿Cómo son diferentes? ¿Cómo son similares los inputs? ¿En qué se diferencian? ¿Qué ocurre si copias el encabezado YAML de uno al otro?

4. Crea un nuevo documento R Markdown para cada uno de los tres formatos incorporados: HTML, PDF and Word. Haz *knit* en cada uno de estos tres documentos. ¿Como difiere el output? ¿Cómo difiere el input? (Puedes necesitar instalar LaTeX para poder compilar el output en PDF— RStudio te preguntará si esto es necesario).

27.3 Formateo de texto con Markdown

La prosa en los archivos `.Rmd` está escrita en Markdown, una colección simple de convenciones para dar formato a archivos de texto plano. Markdown está diseñado para ser fácil de leer y fácil de escribir. Es también muy fácil de aprender. La siguiente guía muestra cómo usar el Markdown de Pandoc, una versión ligeramente extendida de Markdown que R Markdown comprende.

```
Formato de texto
------------------------------------------------------------

*cursiva*   ó _cursiva_
**negrita**   __negrita__
`code`
superscript^2^ y subscript~2~

Encabezados
------------------------------------------------------------

# Encabezado de primer nivel

## Encabezado de segundo nivel

### Encabezado de tercer nivel

Listas
------------------------------------------------------------

*   Elemento 1 en lista no enumerada

*   Elemento 2

    * Elemento 2a

    * Elemento 2b

1.  Elemento 1 en lista enumerada

1.  Elemento 2. La numeración se incrementa automáticamente en el output.

Enlaces e imágenes
------------------------------------------------------------

<http://ejemplo.com>

[texto del enlace](http://ejemplo.com)

![pie de página opcional](ruta/de/la/imagen.png)

Tablas
------------------------------------------------------------

Primer encabezado     | Segundo encabezado
--------------------- | ---------------------
Contenido de la celda | Contenido de la celda
Contenido de la celda | Contenido de la celda
```

Copy

La mejor manera de aprender estas convenciones es simplemente probar. Tomará unos días, pero pronto se convertirán en algo natural y no necesitarás pensar en ellas. Si te olvidas, puedes tener una útil hoja de referencia con *Help* > *Markdown Quick Reference*.

27.3.1 **Ejercicios**

1. Practica lo que has aprendido crando un CV breve. El título debería ser tu nombre, y deberías incluir encabezados para (por lo menos) educación o empleo. Cada una de las secciones debería incluir una lista con viñetas de trabajos/ títulos obtenidos. Resalta el año en negrita.

2. Usando la referencia rápida de R Markdown, descubre como:

3. Agregar una nota al pie.

4. Agregar una linea horizontal.

5. Agregar una cita en bloque.

6. Copia y pega los contenidos de `diamond-sizes.Rmd` desde https://github.com/hadley/r4ds/tree/master/rmarkdown a un documento local de R Markdown. Revisa que puedes ejecutarlo, agrega texto después del polígono de frecuencias que describa sus características más llamativas.

27.4 Bloques de código

Para ejecutar código dentro de un documento R Markdown, necesitas insertar un bloque o *chunk*, en inglés. Hay tres maneras para hacerlo:

1. Con el atajo de teclado: Cmd/Ctrl + Alt + I
2. Con el ícono "*Insert*" en la barra de edición
3. Tipeando manualmente los delimitadores de bloque ```` ```{r} ```` y ```` ``` ````.

Obviamente, nuestra recomendación es que aprendas a usar el atajo de teclado. A largo plazo, te ahorrará mucho tiempo.

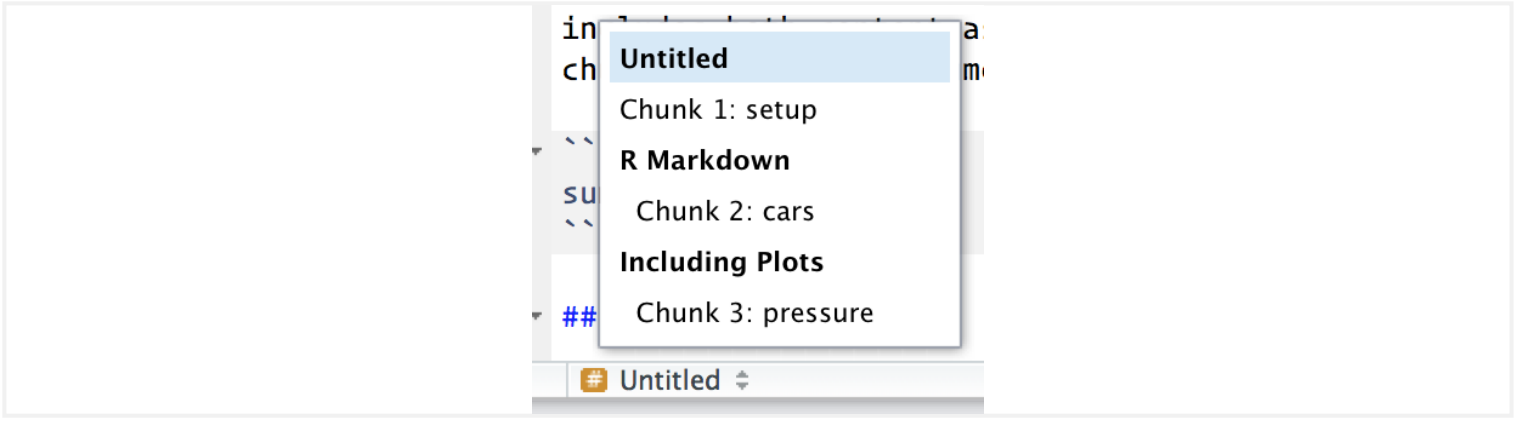
Puedes continuar ejecutando el código usando el atajo de teclado que para este momento (¡esperamos!) ya conoces y amas : Cmd/Ctrl + Enter. Sin embargo, los bloques de código tienen otro atajo de teclado: Cmd/Ctrl + Shift + Enter, que ejecuta todo el código en el bloque. Piensa el bloque como una función. Un bloque debería ser relativamente autónomo y enfocado en torno a una sola tarea.

Las siguientes secciones describen el encabezado de bloque, que consiste en ```` ```{r ````, seguido por un nombre opcional para el bloque, seguido luego por opciones separadas por comas y un `}`. Luego viene tu código de R. El término del bloque se indica con un ```` ``` ```` final.

27.4.1 Nombres de los bloques

Los bloques puede tener opcionalmente nombres: ```` ```{r nombre} ````. Esto presenta tres ventajas:

1. Puedes navegar más fácilmente a bloques específicos usando el navegador de código desplegable abajo a la izquierda en el editor de *script*:



1. Los gráficos producidos por los bloques tendrán nombres útiles que hace que sean más fáciles de utilizar en otra parte. Más sobre esto en la sección sobre otras opciones importantes.
2. Puedes crear redes de bloques guarados en el caché para evitar re-ejecutar cómputos costosos en cada ejecución. Más sobre esto más adelante.

Hay un nombre de bloque que tiene comportamiento especial: `setup`. Cuando te encuentras en modo *notebook*, el bloque llamado setup se ejecutará automáticamente una vez, antes de ejecutar cualquier otro código.

27.4.2 Opciones de los bloques

La salida de los bloques puede personalizarse con **options**, que son argumentos suministrados en el encabezado del bloque. Knitr provee casi 60 opciones que puedes usar para personalizar tus bloques de código. Aquí cubriremos las opciones de bloques más importantes que usarás más frecuentemente. Puedes ver la lista completa en http://yihui.name/knitr/options/.

El conjunto más importante de opciones controla si tu bloque de código es ejecutado y qué resultados estarán insertos en el reporte final:

- `eval = FALSE` evita que el código sea evaluado. (Y, obviamente, si el código no es ejecutado no se generaran resultados). Esto es útil para mostrar códigos de ejemplo, o para deshabilitar un gran bloque de código sin comentar cada línea.

- `include = FALSE` ejecuta el código, pero no muestra el código o los resultados en el documento final. Usa esto para código de configuración que no quieres que abarrote tu reporte.

- `echo = FALSE` evita que se vea el código, pero sí muestra los resultados en el archivo final. Utiliza esto cuando quieres escribir reportes enfocados a personas que no quieren ver el código subyacente de R.

- `message = FALSE` o `warning = FALSE` evita que aparezcan mensajes o advertencias en el archivo final.

- `results = 'hide'` oculta el *output* impreso; `fig.show = 'hide'` oculta gráficos.

- `error = TRUE` causa que el *render* continúe incluso si el código devuelve un error. Esto es algo que raramente quieres incluir en la versión final de tu reporte, pero puede ser muy útil si necesitas depurar exactamente qué ocurre dentro de tu `.Rmd`. Es también útil si estás enseñando R y quieres incluir deliberadamente un error. Por defecto, `error = FALSE` provoca que el *knitting* falle si hay incluso un error en el documento.

La siguiente tabla resume qué tipos de *output* suprime cada opción:

Opción	Ejecuta	Muestra	Output	Gráficos	Mensajes	Advertencias
eval = FALSE	-		-	-	-	-
include = FALSE		-	-	-	-	-
echo = FALSE		-				
results = "hide"			-			
fig.show = "hide"				-		
message = FALSE					-	
warning = FALSE						-

27.4.3 Tablas

Por defecto, R Markdown imprime data frames y matrices tal como se ven en la consola:

```
mtcars[1:5, ]
#>                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
#> Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
#> Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
#> Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
#> Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
#> Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
```

Si prefieres que los datos tengan formato adicional, puedes usar la función `knitr::kable`. El siguiente código genera una Tabla 27.1.

```
knitr::kable(
  mtcars[1:5, ],
  caption = "Un kable de knitr."
)
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3	2

Table 27.1: Un kable de knitr.

Lee la documentación para `?knitr::kable` para ver los otros modos en los que puedes personalizar la tabla. Para una mayor personalización, considera los paquetes **xtable**, **stargazer**, **pander**, **tables** y **ascii**. Cada uno provee un set de herramientas para generar tablas con formato a partir código de R.

Hay también una gran cantidad de opciones para controlar cómo las figuras están embebidas o incrustadas. Aprenderás sobre esto en la sección guardando tus gráficos.

27.4.4 Caching

Normalmente, cada *knit* de un documento empieza desde una sesión limpia. Esto es genial para la reproducibilidad, porque se asegura que has capturado cada cómputo importante en el código. Sin embargo, puede ser doloroso si tienes cómputos que toman mucho tiempo. La solución es `cache = TRUE`. Cuando está configurada, esto guarda el output del bloque en un archivo con un nombre especial en el disco. En ejecuciones siguientes, *knitr* revisará si el código ha cambiado, y si no ha hecho, reutilizará los resultados del caché.

El sistema de caché debe ser usado con cuidado, porque por defecto está solo basado en el código, no en sus dependencias. Por ejemplo, aquí el bloque `datos_procesados` depende del bloque `datos_crudos`:

````
```{r datos_crudos}
datos_crudos <- readr::read_csv("un_archivo_muy_grande.csv")
```

```{r datos_procesados, cache = TRUE}
datos_procesados <- datos_crudos %>%
  filter(!is.na(variable_important)) %>%
  mutate(nueva_variable = transformacion_complicada(x, y, z))
```
````

Hacer *caching* en el bloque `datos_procesados` significa que se re-ejecutará si el pipeline de *dplyr* se modifica, pero no re-ejecutará si cambia la llamada a `read_csv()`. Puedes evitar este problema con la opción de bloque `dependson` (*depende de*, en inglés):

````
```{r datos_procesados, cache = TRUE, dependson = "datos_crudos"}
datos_procesados <- datos_crudos %>%
  filter(!is.na(variable_important)) %>%
  mutate(nueva_variable = transformacion_complicada(x, y, z))
```
````

`dependson` debiese incluir un vector de caracteres de *cada* bloque del que depende el bloque cacheado. *Knitr* actualizará los resultados para el bloque cacheado cada vez que detecta que una de sus dependencias ha cambiado.

Ten en cuenta que los bloques de código no se actualizarán si `un_archivo_muy_grande.csv` cambia, porque el caché de *knitr* solo hace seguimiento de los cambios dentro del archivo `.Rmd`. Si quieres seguir también los cambios hechos en ese archivo, puedes usar la opción `cache.extra`. Esta es una expresión arbitraria de R que invalidará el *caché* cada vez que cambie. Una buena función a usar es `file.info()`: genera mucha información sobre el archivo incluyendo cuándo fue su última modificación. Puedes escribir entonces:

````
```{r datos_crudos, cache.extra = file.info("un_archivo_muy_grande.csv")}
datos_crudos <- readr::read_csv("un_archivo_muy_grande.csv")
```
````

A medida que tus estrategias de *cacheo* se vuelvan progresivamente más complicadas, es una buena idea limpiar regularmente todos tus *cachés* con `knitr::clean_cache()`.

Hemos utilizado el consejo de David Robinson para nombrar estos bloques: cada bloque debe nombrarse según el objeto principal que crea. Esto hace mucho más fácil entender la especificación `dependson`.

27.4.5 Opciones globales

A medida que trabajes más con *knitr*, descubrirás que algunas de las opciones de bloque por defecto no se ajustan a tus necesidades y querrás cambiarlas. Puedes hacer esto incluyendo `knitr::opts_chunk$set()` en un bloque de código. Por ejemplo, cuando escribimos libros y tutoriales seteamos:

```
knitr::opts_chunk$set(
  comment = "#>",
  collapse = TRUE
)
```

Esto utiliza nuestro formato preferido de comentarios y se asegura que el código y el *output* se mantengan entrelazados. Por otro lado, si preparas un reporte, puedes fijar:

```
knitr::opts_chunk$set(
  echo = FALSE
)
```

Esto ocultará por defectoel código, así que solo mostrará los bloques que deliberadamente has elegido mostrar (con `echo = TRUE`). Puedes considerar fijar `message = FALSE` y `warning = FALSE`, pero eso puede hacer más díficil la tarea de depurar problemas, porque no verías ningún mensajes en el documento final.

27.4.6 **Código dentro de una línea**

Hay otro modo de incluir código R en un documento R Markdown: directamente en el texto, con: `` `r ` ``. Esto puede ser muy útil si mencionas propiedades de tus datos en el texto. Por ejemplo, en el documento de ejemplo que utilizamos al comienzo del capítulo teníamos:

> Tenemos datos sobre `` `r nrow(diamonds)` `` diamantes. Solo `` `r nrow(diamonds) - nrow(smaller)` `` son de más de 2.5 quilates. La distribución de los restantes se muestra a continuación:

Cuando hacemos *knit*, los resultados de estos cómputos están insertos en el texto:

> Tenemos datos de 53940 diamantes. Solo 126 son de más de 2.5 quilates. La distribución de los restantes se muestra a continuación:

Cuando insertas números en el texto, la función `format()` es tu amiga. Esta permite establecer el número de dígitos (`digits`) para que no imprimas con un grado ridículo de precisión y el separador de miles (`big.mark`) para hacer que los números sean más fáciles de leer. Siempre combinamos estos en una función de ayuda:

```
coma <- function(x) format(x, digits = 2, big.mark = ",")
coma(3452345)
#> [1] "3,452,345"
coma(.12358124331)
#> [1] "0.12"
```

27.4.7 **Ejercicios**

1. Incluye una sección que explore cómo los tamaños de diamantes varían por corte, color y claridad. Asume que escribes un reporte para alguien que no conoce R, por lo que en lugar de fijar `echo = FALSE` en cada bloque, fija una opción global.

2. Descarga `diamond-sizes.Rmd` de https://github.com/cienciadedatos/r4ds/blob/traduccion/rmarkdown. Agrega una sección que describa los 20 diamantes mas grandes, incluyendo una tabla que muestre sus atributos más importantes.

3. Modifica `diamonds-sizes.Rmd` para usar `coma()` para producir un formato de *output* ordenado. También incluye el porcentaje de diamantes que son mayores a 2.5 quilates.

4. Define una red de bloques donde `d` dependa de `c` y de `b`, y tanto `b` y `c` dependan de `a`. Haz que cada bloque imprima `lubridate::now()`, fija `cache = TRUE` y verifica que estás comprendiendo cómo se almacena en *cache*.

27.5 Solucionando problemas

Solucionar problemas en documentos de R Markdown puede ser muy desafiante, ya que no te encuentras usando R en un ambiente de R interactivo. Es por eso que necesitarás aprender algunos trucos nuevos. La primera cosa que siempre debes intentar es recrear el problema en una sesión interactiva. Reinicia R, ejecuta todos los bloques de código (ya sea desde el menú Code, desde el menú desplegable bajo Run o con el atajo del teclado Ctrl + Alt + R). Si tienes suerte, eso recreará el problema y podrás descubrir lo que está ocurriendo interactivamente.

Si eso no ayuda, debe haber algo diferente entre tu ambiente interactivo y el ambiente de R Markdown. Tendrás que explorar sistemáticamente las opciones. La diferencia más común es el directorio de trabajo: el directorio de trabajo de R Markdown es el directorio en el que se encuentra. Revisa que el directorio de trabajo es el que esperas incluyendo `getwd()` en un bloque.

A continuación, piensa en todas las cosas que podrían causar el error. Necesitarás revisar sistemáticamente que tu sesión de R y tu sesión de R Markdown sean la misma. La manera más fácil de hacer esto es fijar `error = TRUE` en el bloque que causa problemas, y luego usa `print()` y `str()` para revisar que la configuración es la esperada.

27.6 Encabezado YAML

Puedes controlar otras configuraciones de "documento completo" haciendo ajustes a los parámetros del encabezado YAML. Estarás preguntándote que significa YAML: es la sigla en inglés de la frase *"yet another markup language"*, que significa *"otro lenguaje de marcado más"*. Este lenguaje de marcado está diseñado para representar datos jerárquicos de modo tal que sea fácil de escribir y leer para humanos. R Markdown lo utiliza para controlar muchos detalles del *output*. Aquí discutiremos dos: parámetros del documento y bibliografías.

27.6.1 Parámetros

Los documentos R Markdown pueden incluir uno o mas parámetros cuyos valores pueden ser fijados cuando se renderiza el reporte. Los parámetros son útiles cuando quieres re-renderizar el mismo reporte con valores distintos para varios inputs clave. Por ejemplo, podrías querer producir reportes de venta por ramas, resultados de un examen por alumno, resúmenes demográficos por país. Para declarar uno o más parámetros, utiliza el campo `params` .

Este ejemplo utiliza el parámetro `my_class` para determinar que clase de auto mostrar:

Copy

```
---
output: html_document
params:
  mi_clase: "suv"
---

```{r setup, include = FALSE}
library(datos)
library(ggplot2)
library(dplyr)

clase <- millas %>% filter(clase == params$mi_clase)
```

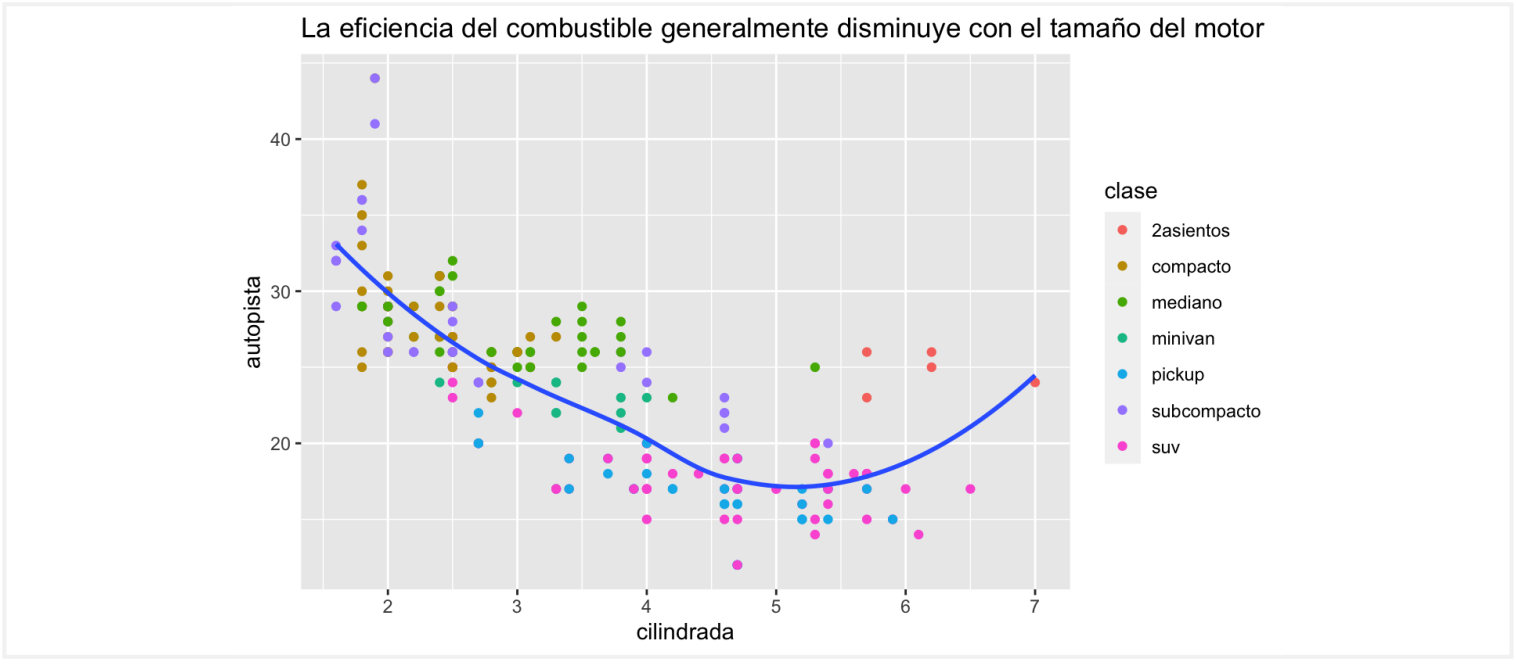
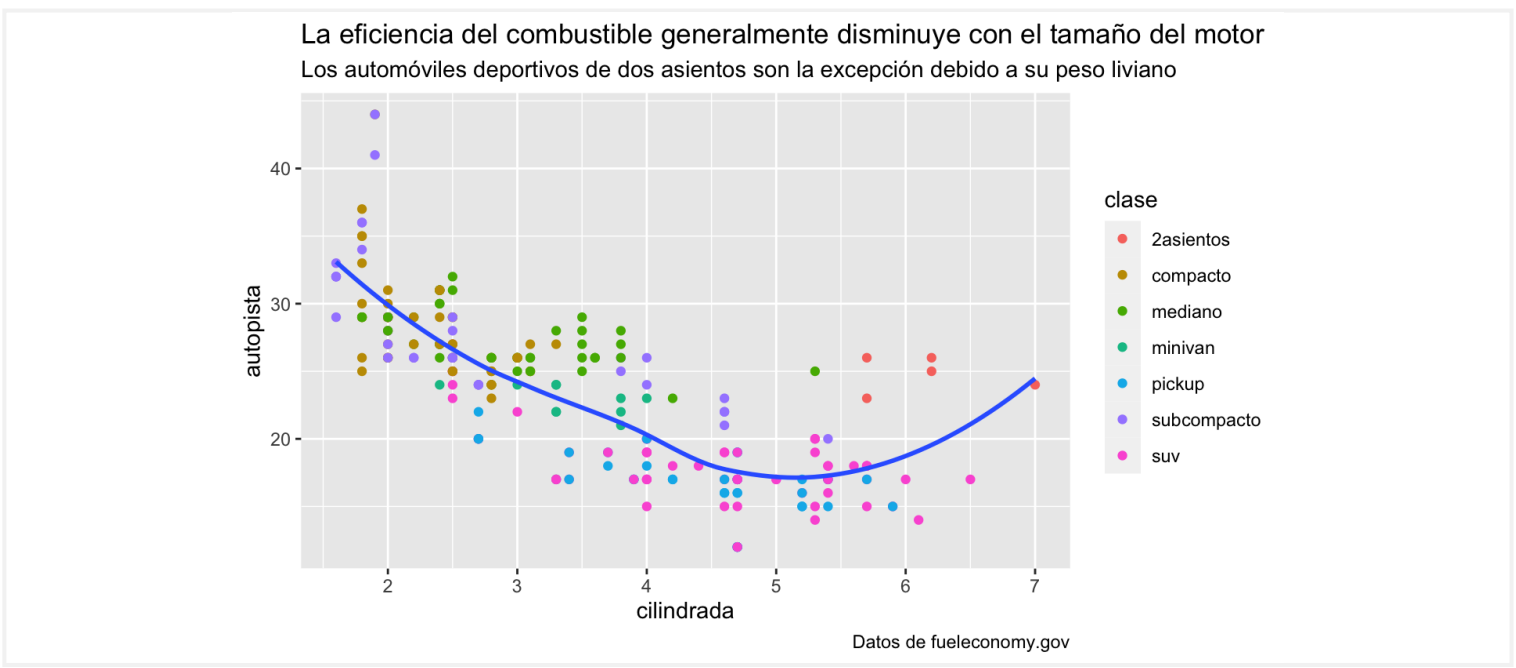
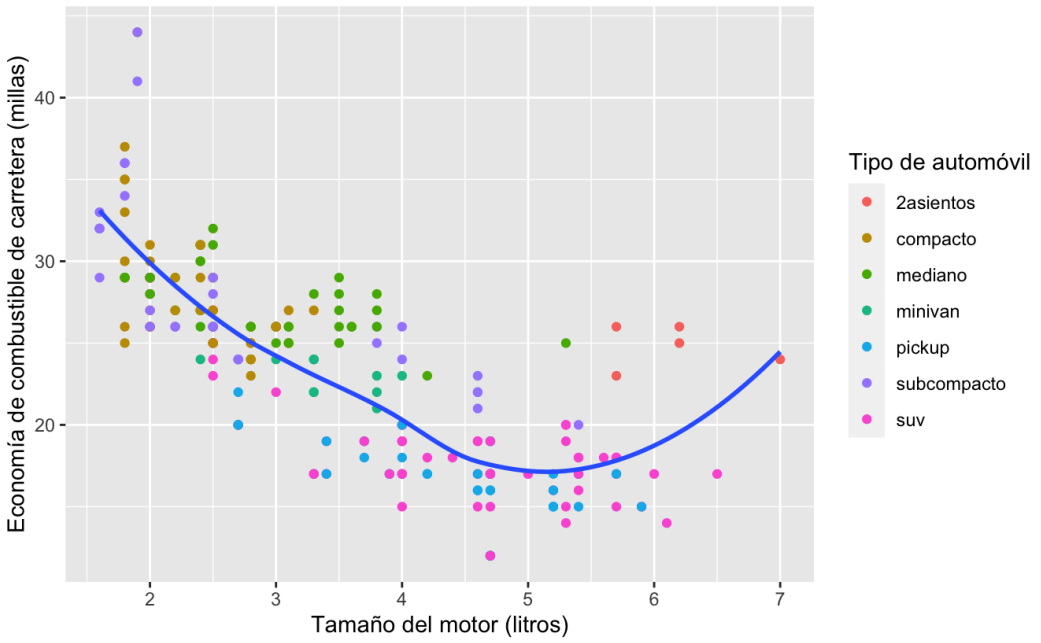
Economía de combustible en vehículos `r params$mi_clase`

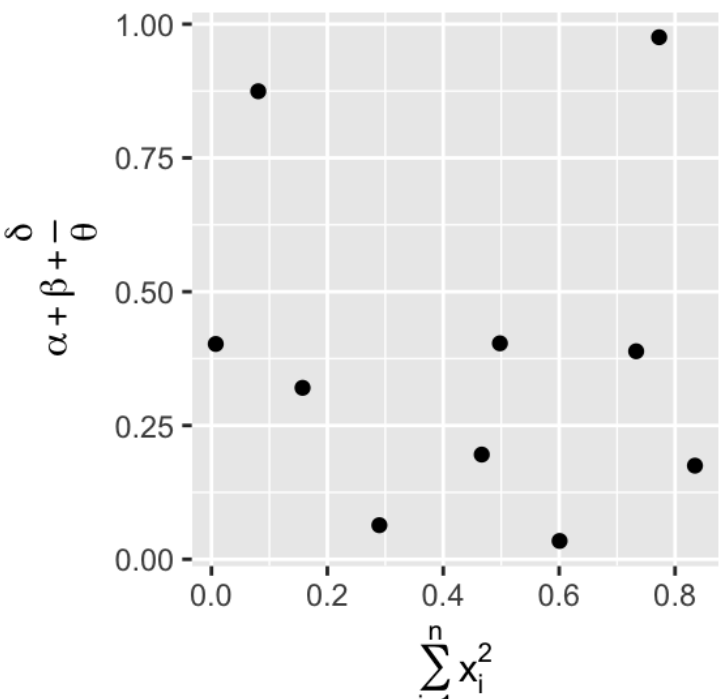
```{r, message = FALSE}
ggplot(clase, aes(motor, autopista)) +
  geom_point() +
  geom_smooth(se = FALSE)
```
```

Como puedes ver, los parámetros están disponibles dentro de los bloques de código como una lista de solo lectura llamada `params` .

Puedes escribir vectores atómicos directamente en el encabezado YAML. Puedes también ejecutar expresiones arbitrarias de R agregando `!r` antes del valor del parámetro. Esta es una buena manera de especificar parámetros de fecha/hora.

```
params:
 start: !r lubridate::ymd("2015-01-01")
 snapshot: !r lubridate::ymd_hms("2015-01-01 12:30:00")
```

Copy

En RStudio, puedes hacer clic en la opción "Knit with Parameters" en el menú desplegable *Knit* para fijar parámetros, renderizar y previsualizar en un solo paso amigable. Puedes personalizar el diálogo fijando otras opciones en el encabezado. Ver para más detalles http://rmarkdown.rstudio.com/developer_parameterized_reports.html#parameter_user_interfaces (en inglés).

De manera alternativa, si necesitas producir varios reportes parametrizados, puedes ejecutar `rmarkdown::render()` con una lista de `params`:

```
rmarkdown::render("fuel-economy.Rmd", params = list(mi_clase = "suv"))
```

Copy

Esto es particularmente poderoso en conjunto con `purrr:pwalk()`. El siguiente ejemplo crea un reporte para cada valor de `clase` que se encuentra en `millas`. Primero creamos un data frame que tiene una fila para cada clase, dando el nombre de archivo (`filename`) del reporte y los `params`:

```
reportes <- tibble(
  clase = unique(millas$clase),
  filename = stringr::str_c("economia-combustible-", clase, ".html"),
  params = purrr::map(clase, ~ list(mi_clase = .))
)
reportes
#> # A tibble: 7 x 3
#>   clase     filename                             params
#>   <chr>     <chr>                                <list>
#> 1 compacto  economia-combustible-compacto.html  <named list [1]>
#> 2 mediano   economia-combustible-mediano.html   <named list [1]>
#> 3 suv       economia-combustible-suv.html       <named list [1]>
#> 4 2asientos economia-combustible-2asientos.html <named list [1]>
#> 5 minivan   economia-combustible-minivan.html   <named list [1]>
#> 6 pickup    economia-combustible-pickup.html    <named list [1]>
#> # … with 1 more row
```

Copy

Entonces unimos los nombres de las columnas con los nombres de los argumentos de `render()`, y utilizamos la función `pwalk()` (*parallel walk*) del paquete **purrr** para invocar `render()` una vez en cada fila:

```
reportes %>%
  select(output_file = filename, params) %>%
  purrr::pwalk(rmarkdown::render, input = "fuel-economy.Rmd")
```

Copy

27.6.2 Bibliografías y citas

Pandoc puede generar automáticamente citas y bibliografía en varios estilos. Para usar esta característica, especifica un archivo de bibliografía usando el campo `bibliography` en el encabezado de tu archivo. El campo debe incluir una ruta del directorio que contiene tu archivo .Rmd al archivo que contiene el archivo de bibliografía:

```
bibliography: rmarkdown.bib
```

Copy

Puedes usar muchos formatos comunes de biliografía incluyendo BibLaTeX, BibTeX, endnote, medline.

Para crear una cita dentro de tu archivo .Rmd, usa una clave compuesta de '@' + el identificador de la cita del archivo de la bibliografía. Después, ubica esta cita entre corchetes. Aquí hay algunos ejemplos:

```
Multiple citas se separan con un `;`: Bla bla[@smith04; @doe99].

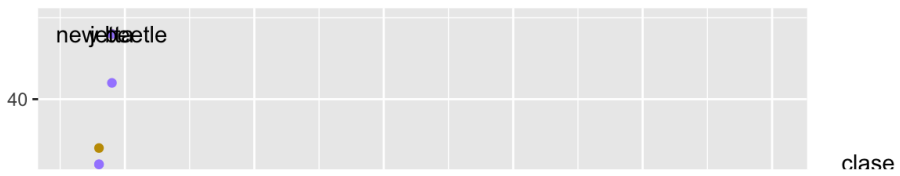
Puedes incluir comentarios arbritarios dentro de los corchetes:
Bla bla [ver @doe99, pp. 33-35; también @smith04, ch. 1].

Remover los corchetes para crear una cita dentro del texto: @smith04
dice bla, o @smith04 [p. 33] dice bla.

Agrega un signo `-` antes de la cita para eliminar el nombre del autor:

Smith dice bla [-@smith04].
```

Cuando R Markdown *renderice* tu archivo, construirá y agregará una bibliografía al final del documento. La bibliografía contendrá cada una de las referencias citadas de tu archivo de bibliografía, pero no contendrá un encabezado de sección. Como resultado, es una práctica común finalizar el archivo con un encabezado de sección para la bibliografía, tales como `# Referencias` or `# Bibliografía`.

Puedes cambiar el estilo de tus citas y bibliografía referenciando un archivo CSL (sigla de "citation style language", es decir, lenguaje de estilo de citas) en el campo `csl`:

```
bibliography: rmarkdown.bib
csl: apa.csl
```

Tal y como en el campo de bilbiografía, tu archivo csl debería contener una ruta al archivo. Aquí asumimos que el archivo csl está en el mismo directorio que el archivo .Rmd. Un buen lugar para encontrar archivos CSL para estilos de bibliografía comunes es http://github.com/citation-style-language/styles.

27.7 Aprendiendo más

R Markdown es todavía relativamente reciente y sigue creciendo con rapidez. El mejor lugar para estar al tanto de las innovaciones es el sitio oficial de R Markdown: http://rmarkdown.rstudio.com.

Hay dos tópicos importantes que no hemos mencionado aquí: colaboraciones y los detalles de comunicar de manera precisa tus ideas a otros seres humanos. La colaboración es una parte vital de la ciencia de datos moderna y puedes hacer tu vida mucho más fácil si usas herramientas de control de versión, tales como Git y GitHub. Recomendamos dos recursos gratuitos que te enseñarán Git:

1. "Happy Git with R": una introducción amigable a Git y GitHub para personas que usan R, de Jenny Bryan. El libro esta disponible de manera libre en línea en: http://happygitwithr.com

2. El capitulo "Git and GitHub" de *R Packages*, de Hadley Wickham. Puedes también leerlo online: https://r-pkgs.org/git.html.

Tampoco hemos hablado acerca de qué es lo que realmente deberías escribir para poder comunicar claramente los resultados de tu análisis. Para mejorar tu escritura, recomendamos leer cualquiera de estos libros (en inglés): *Style: Lessons in Clarity and Grace* de Joseph M. Williams & Joseph Bizup o *The Sense of Structure: Writing from the Reader's Perspective* de George Gopen. Ambos libros te ayudarán a entender la estructura de oraciones y párrafos, y te darán las herramientas para hacer más clara tu escritura. (Estos libros son bastante caros si se compran nuevos, pero dado que son usados en muchas clases de inglés es posible encontrar copias baratas de segunda mano). George Gopen también ha escrito varios artículos cortos en inglés sobre escritura en https://www.georgegopen.com/the-litigation-articles.html. Están dirigidos a abogados, pero casi todo también se aplica en el contexto de la ciencia de datos.

« 26 Introducción

28 Comunicar con gráficos »

28 Comunicar con gráficos

28.1 Introducción

En [análisis de datos exploratorios] aprendiste a usar gráficos como herramientas de *exploración*. Cuando haces gráficos exploratorios, sabes incluso antes de mirar, qué variables mostrará el gráfico. Hiciste cada gráfico con un propósito, lo miraste rápidamente y luego pasaste al siguiente. En el transcurso de la mayoría de los análisis, producirás decenas o cientos de gráficos, muchos de los cuales se desecharán inmediatamente.

Ahora que comprendes tus datos, debes *comunicar* tu conocimiento a los demás. Es probable que tu audiencia no comparta tus conocimientos previos y no esté profundamente involucrada en los datos. Para ayudar a otros a construir rápidamente un buen modelo mental de los datos, deberás invertir un esfuerzo considerable para que tus gráficos se expliquen por sí solos . En este capítulo, aprenderás algunas de las herramientas que proporciona **ggplot2** para hacerlo.

Este capítulo se centra en las herramientas necesarias para crear buenos gráficos. Supongo que sabes lo que quieres y solo te falta saber cómo hacerlo. Por esa razón, recomiendo combinar este capítulo con un buen libro de visualización general. Me gusta especialmente *The Truthful Art*, de Albert Cairo. No enseña la mecánica de crear visualizaciones, sino que se enfoca en lo que necesitas pensar para crear gráficos efectivos.

28.1.1 Prerrequisitos

En este capítulo, nos centraremos una vez más en ggplot2. También usaremos un poco el paquete **dplyr** para la manipulación de datos y algunos paquetes como **ggrepel** y **viridis** que extienden las funciones de **ggplot2**. En lugar de cargar esas extensiones aquí, nos referiremos a sus funciones de forma explícita, utilizando la notación `::` . Esto ayudará a aclarar qué funciones están integradas en ggplot2 y cuáles vienen de otros paquetes. No olvides que deberás instalar esos paquetes con `install.packages()` si aún no los tienes.

```
library(tidyverse)
library(datos)
```

Copy

28.2 Etiquetas

El punto de inicio más sencillo para convertir un gráfico exploratorio en un gráfico expositivo es con buenas etiquetas. Agrega etiquetas con la función `labs()` . Este ejemplo agrega un título al gráfico:

```
ggplot(millas, aes(cilindrada, autopista)) +
  geom_point(aes(color = clase)) +
  geom_smooth(se = FALSE) +
  labs(title = "La eficiencia del combustible generalmente disminuye con el tamaño del motor")
```

Copy



El propósito del título de un gráfico es resumir el hallazgo principal. Evita títulos que simplemente describen el gráfico, por ejemplo "Diagrama de dispersión del desplazamiento del motor frente al ahorro de combustible".

Si necesitas agregar más texto, hay otras dos etiquetas útiles que puedes usar en **ggplot2** versión 2.2.0 y superiores (que deberían estar disponibles para cuando estés leyendo este libro):

- el `subtítulo`, del inglés *subtitle*, agrega detalles adicionales en una fuente más pequeña debajo del título.
- la `leyenda`, del inglés *caption*, agrega texto en la parte inferior derecha del gráfico, suele usarse para describir la fuente de los datos.

```
ggplot(millas, aes(cilindrada, autopista)) +
  geom_point(aes(color = clase)) +
  geom_smooth(se = FALSE) +
  labs(
    title = "La eficiencia del combustible generalmente disminuye con el tamaño del motor",
    subtitle = "Los automóviles deportivos de dos asientos son la excepción debido a su peso liviano",
    caption = "Datos de fueleconomy.gov"
  )
```

Copy



También puedes usar `labs()` para reemplazar los títulos de ejes y leyendas. Por lo general, es una buena idea reemplazar los nombres cortos de las variables con descripciones más detalladas e incluir las unidades.

```
ggplot(millas, aes(cilindrada, autopista)) +
  geom_point(aes(colour = clase)) +
  geom_smooth(se = FALSE) +
  labs(
    x = "Tamaño del motor (litros)",
    y = "Economía de combustible de carretera (millas)",
    colour = "Tipo de automóvil"
  )
```

Copy



Es posible usar ecuaciones matemáticas en lugar de cadenas de texto. Simplemente cambia `""` por `quote()` y lee acerca de las opciones disponibles en `?plotmath`:

```
df <- tibble(
  x = runif(10),
  y = runif(10)
)

ggplot(df, aes(x, y)) +
  geom_point() +
  labels(
    x = quote(sum(x[i]^2, i == 1, n)),
    y = quote(alpha + beta + frac(delta, theta))
  )
```



28.2.1 Ejercicios

1. Crea un gráfico partiendo de los datos de economía de combustible con etiquetas para `title`, `subtitle`, `caption`, `x`, `y` y `color` personalizadas.

2. La función `geom_smooth()` es un poco engañosa porque `autopista` está sesgada positivamente para motores grandes, debido a la inclusión de autos deportivos livianos con motores grandes. Usa tus herramientas de modelado para ajustar y mostrar un modelo mejor.

3. Elige un gráfico exploratorio que hayas creado en el último mes y agrégale títulos informativos para volverlo más fácil de comprender para otros.

28.3 Anotaciones

Además de etiquetar las partes principales de tu gráfico, a menudo es útil etiquetar observaciones individuales o grupos de observaciones. La primera herramienta que tienes a tu disposición es `geom_text()`. La función `geom_text()` es similar a `geom_point()`, pero tiene una estética adicional: `label`. Esto hace posible agregar etiquetas textuales a tus gráficos.

Hay dos posibles fuentes de etiquetas. En primer lugar, es posible tener un tibble que proporcione las etiquetas. El siguiente gráfico no es en sí terriblemente útil, pero si lo es su enfoque: filtrar el auto más eficiente de cada clase con **dplyr**, y luego etiquetarlo en el gráfico:

```
mejor_de_su_clase <- millas %>%
  group_by(clase) %>%
  filter(row_number(desc(autopista)) == 1)

ggplot(millas, aes(cilindrada, autopista)) +
  geom_point(aes(colour = clase)) +
  geom_text(aes(label = modelo), data = mejor_de_su_clase)
```

Copy



Esto es difícil de leer porque las etiquetas se superponen entre sí y con los puntos. Podemos mejorar ligeramente las cosas cambiando por `geom_label()`, que dibuja un rectángulo detrás del texto. También usamos el parámetro `nudge_y` para mover las etiquetas ligeramente por encima de los puntos correspondientes:

```
ggplot(millas, aes(cilindrada, autopista)) +
  geom_point(aes(colour = clase)) +
  geom_label(aes(label = modelo), data = mejor_de_su_clase, nudge_y = 2, alpha = 0.5)
```

Copy

Esto ayuda un poco, pero si te fijas bien en la esquina superior izquierda verás que hay dos etiquetas prácticamente una encima de la otra. Esto sucede porque el kilometraje y el desplazamiento para los mejores automóviles en las categorías de compactos y subcompactos son exactamente los mismos. No hay forma de que podamos solucionar esto aplicando la misma transformación para cada etiqueta. En cambio, podemos usar el paquete **ggrepel** de Kamil Slowikowski. Este paquete es muy útil ya que ajusta automáticamente las etiquetas para que no se superpongan:

```
ggplot(millas, aes(cilindrada, autopista)) +
  geom_point(aes(colour = clase)) +
  geom_point(size = 3, shape = 1, data = mejor_de_su_clase) +
  ggrepel::geom_label_repel(aes(label = modelo), data = mejor_de_su_clase)
```

Copy

Ten en cuenta otra técnica muy práctica utilizada aquí: agregué una segunda capa de puntos grandes y huecos para resaltar los puntos que etiqueté.

A veces puedes usar la misma idea para reemplazar la leyenda con etiquetas colocadas directamente en tu gráfico. Los resultados no son maravillosos para este gráfico en particular, pero tampoco son tan malos. (`theme(legend.position = "none")` desactiva la leyenda — hablaremos de ello en breve).

```
clase_promedio <- millas %>%
  group_by(clase) %>%
  summarise(
    cilindrada = median(cilindrada),
    autopista = median(autopista)
  )

ggplot(millas, aes(cilindrada, autopista, colour = clase)) +
  ggrepel::geom_label_repel(aes(label = clase),
    data = clase_promedio,
    size = 6,
    label.size = 0,
    segment.color = NA
  ) +
  geom_point() +
  theme(legend.position = "none")
```

Alternativamente puede que quieras agregar una única etiqueta al gráfico, pero de todas formas necesitarás generar un conjunto de datos. Puede ocurrir que desees ubicar la etiqueta en la esquina del gráfico, en ese caso es conveniente crear un nuevo marco de datos usando `summarise()` para calcular los valores máximos de x e y.

```
etiqueta <- millas %>%
  summarise(
    cilindrada = max(cilindrada),
    autopista = max(autopista),
    etiqueta = "El aumento del tamaño del motor está \nrelacionado con la disminución en el gasto de combustible."
  )

ggplot(millas, aes(cilindrada, autopista)) +
  geom_point() +
  geom_text(aes(label = etiqueta), data = etiqueta, vjust = "top", hjust = "right")
```

Si deseas colocar el texto exactamente en los bordes del gráfico puedes usar `+Inf` y `-Inf`. Como ya no estamos calculando las posiciones de `millas`, podemos usar `tibble()` para crear el conjunto de datos:

```
etiqueta <- millas %>%
  summarise(
    cilindrada = Inf,
    autopista = Inf,
    etiqueta = "El aumento del tamaño del motor está \nrelacionado con la disminución en el gasto de combustible."
  )

ggplot(millas, aes(cilindrada, autopista)) +
  geom_point() +
  geom_text(aes(label = etiqueta), data = etiqueta, vjust = "top", hjust = "right")
```

En estos ejemplos, separé manualmente la etiqueta en líneas usando "". Otra posibilidad es usar `stringr::str_wrap()` para agregar saltos de línea automáticamente, dado el número de caracteres que deseas por línea:

```
"El aumento del tamaño del motor está relacionado con la disminución en el gasto de combustible." %>%
  stringr::str_wrap(width = 40) %>%
  writeLines()
#> El aumento del tamaño del motor está
#> relacionado con la disminución en el
#> gasto de combustible.
```

Ten en cuenta el uso de `hjust` y `vjust` para controlar la alineación de la etiqueta. La figura 28.1 muestra las nueve combinaciones posibles.

Figure 28.1: Las nueve combinaciones posibles con `hjust` y `vjust`.

Recuerda que además de `geom_text()`, en ggplot2 tienes muchos otros geoms disponibles para ayudar a agregar notas a tu gráfico . Algunas ideas:

- Emplea `geom_hline()` y `geom_vline()` para agregar líneas de referencia. A menudo las hago gruesas (`size = 2`) y blancas (`color = white`), y las dibujo debajo de la primera capa de datos. Eso las hace fáciles de ver, sin distraer la atención de los datos.
- Emplea `geom_rect()` para dibujar un rectángulo alrededor de los puntos de interés. Los límites del rectángulo están definidos por las estéticas `xmin` , `xmax` , `ymin` , `ymax` .
- Emplea `geom_segment()` con el argumento `arrow` para destacar un punto en particular con una flecha. Usa la estética `x` e `y` para definir la ubicación inicial, y `xend` y `yend` para definir la ubicación final.

¡El único límite es tu imaginación! (y tu paciencia para posicionar las anotaciones de forma estéticamente agradable)

28.3.1 Ejercicios

1. Usa las infinitas posiciones que permite `geom_text()` para colocar texto en cada una de las cuatro esquinas del gráfico.
2. Lee la documentación de la función `annotate()` . ¿Cómo puedes usarla para agregar una etiqueta de texto a un gráfico sin tener que crear un tibble?
3. ¿Cómo interactúan las etiquetas producidas por `geom_text()` con la separación en facetas? ¿Cómo puedes agregar una etiqueta a una sola faceta? ¿Cómo puedes poner una etiqueta diferente en cada faceta? (Sugerencia: piensa en los datos subyacentes).
4. ¿Qué argumentos para `geom_label()` controlan la apariencia de la caja que se ve atrás?
5. ¿Cuáles son los cuatro argumentos de `arrow()` ? ¿Cómo funcionan? Crea una serie de gráficos que demuestren las opciones más importantes.

28.4 Escalas

La tercera forma en que puedes mejorar tu gráfico para comunicar es ajustar las escalas. Las escalas controla el mapeo de los valores de los datos a cosas que puedes percibir. Normalmente, **ggplot2** agrega escalas automáticamente. Por ejemplo, cuando tipeas:

```
ggplot(millas, aes(cilindrada, autopista)) +
  geom_point(aes(colour = clase))
```

Copy

ggplot2 agrega automáticamente escalas predeterminadas detrás de escena:

```
ggplot(millas, aes(cilindrada, autopista)) +
  geom_point(aes(colour = clase)) +
  scale_x_continuous() +
  scale_y_continuous() +
  scale_colour_discrete()
```

Copy

Ten en cuenta que los nombres de las escalas comienzan siempre igual: `scale_` seguido del nombre de la estética, luego `_` y finalmente el nombre de la escala. Las escalas predeterminadas se nombran según el tipo de variable con la que se alinean: continua, discreta, fecha y hora (datetime) o fecha. Hay muchas escalas no predeterminadas que aprenderás a continuación.

Las escalas predeterminadas se han elegido cuidadosamente para ser adecuadas para una gama amplia de valores. Sin embargo, es posible que desees sobrescribir los valores predeterminados por dos razones:

- Es posible que desees modificar algunos de los parámetros de la escala predeterminada. Esto te permite hacer cosas como cambiar los intervalos de valores en los ejes o las etiquetas de cada valor visible.

- Es posible que desees reemplazar la escala por completo, y utilizar un algoritmo completamente diferente. Por lo general tu opción será mejor que la predeterminada ya que sabes más acerca de los datos.

28.4.1 **Marcas de los ejes y leyendas**

Hay dos argumentos principales que afectan la apariencia de las marcas, del inglés *ticks*, en los ejes y las leyendas: `breaks` y `labels` , del inglés *quiebre* y *etiqueta* respectivamente. Los `breaks` controlan la posición de las marcas en los ejes o los valores asociados con las leyendas. Las `labels` controlan la etiqueta de texto asociada con cada marca/leyenda. El uso más común de los `breaks` es redefinir la opción predeterminada:

```
ggplot(millas, aes(cilindrada, autopista)) +
  geom_point() +
  scale_y_continuous(breaks = seq(15, 40, by = 5))
```

Copy

Puedes usar `labels` de la misma manera (un vector de caracteres de la misma longitud que `breaks`), o puedes establecerlas como `NULL` del inglés *nulo*, para suprimir las etiquetas por completo. Esto es útil para mapas, o para publicar gráficos donde no puedes compartir los números absolutos.

```
ggplot(millas, aes(cilindrada, autopista)) +
  geom_point() +
  scale_x_continuous(labels = NULL) +
  scale_y_continuous(labels = NULL)
```

Copy

También puedes usar `breaks` y `labels` para controlar la apariencia de las leyendas. En conjunto, los ejes y las leyendas se llaman **guías**. Los ejes se usan para la estética de x e y; las leyendas se usan para todo lo demás.

Otro uso de los `breaks` es cuando tienes relativamente pocos puntos de datos y deseas resaltar exactamente dónde se producen las observaciones. Como ejemplo, el siguiente gráfico muestra cuándo comenzó y terminó su mandato cada presidente de los Estados Unidos.

```
presidencial %>%
  mutate(id = 33 + row_number()) %>%
  ggplot(aes(inicio, id)) +
  geom_point() +
  geom_segment(aes(xend = fin, yend = id)) +
  scale_x_date(NULL, breaks = presidencial$inicio, date_labels = "'%y")
```

Copy

Ten en cuenta que la especificación de `breaks` y `labels` para escalas en formato de fecha y fecha y hora es ligeramente diferente:

- `date_labels` toma en cuenta la especificación de formato, en la misma forma que `parse_datetime()` .

- `date_breaks` (no se muestra aquí), toma una cadena como “2 días” o “1 mes”.

28.4.2 **Diseño de leyendas**

Con mayor frecuencia utilizarás `breaks` y `labels` para ajustar los ejes. Aunque ambos también funcionan con leyendas, hay algunas otras técnicas que podrías usar.

Para controlar la posición general de la leyenda, debes usar una configuración de `theme()` del inglés *tema*. Volveremos a los *temas* al final del capítulo, pero en resumen, controlan las partes del gráfico que no son de datos. La configuración del tema `legend.position` del inglés *posición de la leyenda*, controla dónde se

dibuja la leyenda:

```
base <- ggplot(millas, aes(cilindrada, autopista)) +
  geom_point(aes(colour = clase))

base + theme(legend.position = "left")
base + theme(legend.position = "top")
base + theme(legend.position = "bottom")
base + theme(legend.position = "right") # the default
```

Copy

También puedes usar `legend.position = "none"` para suprimir por completo la visualización de la leyenda.

Para controlar la visualización de leyendas individuales, usa `guides()` junto con `guide_legend()` o `guide_colourbar()`. El siguiente ejemplo muestra dos configuraciones importantes: controlar el número de filas que usa la leyenda con `nrow`, y redefinir una de las estéticas para agrandar los puntos. Esto es particularmente útil si has usado un valor de `alfa` bajo para mostrar muchos puntos en un diagrama.

```
ggplot(millas, aes(cilindrada, autopista)) +
  geom_point(aes(colour = clase)) +
  geom_smooth(se = FALSE) +
  theme(legend.position = "bottom") +
  guides(colour = guide_legend(nrow = 1, override.aes = list(size = 4)))
#> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

Copy

28.4.3 Reemplazando una escala

En lugar de simplemente modificar un poco los detalles, puedes reemplazar la escala por completo. Hay dos tipos de escalas que es probable que desees cambiar: escalas de posición continua y escalas de color. Afortunadamente, los mismos principios se aplican a todos los demás aspectos estéticos, por lo que una vez que hayas dominado la posición y el color serás capaz de realizar rápidamente otros reemplazos de escala.

Esto es muy útil para graficar transformaciones de tu variable. A modo de ejemplo, como hemos visto en diamond prices, es más fácil ver la relación precisa entre `quilate` y `precio` si aplicamos una transformación logarítmica en base 10:

```
ggplot(diamantes, aes(quilate, precio)) +
  geom_bin2d()

ggplot(diamantes, aes(log10(quilate), log10(precio))) +
  geom_bin2d()
```

Copy

Sin embargo, la desventaja de esta transformación es que los ejes ahora están etiquetados con los valores transformados, por lo que se vuelve difícil interpretar el gráfico. En lugar de hacer la transformación en el mapeo estético, podemos hacerlo con la escala. Esto es visualmente idéntico, excepto que los ejes están etiquetados en la escala original de los datos.

```
ggplot(diamantes, aes(quilate, precio)) +
  geom_bin2d() +
  scale_x_log10() +
  scale_y_log10()
```

Copy

Otra escala que se personaliza con frecuencia es el color. La escala categórica predeterminada selecciona los colores que están espaciados uniformemente alrededor del círculo cromático. Otras alternativas útiles son las escalas de ColorBrewer que han sido ajustadas manualmente para que funcionen mejor para

personas con tipos comunes de daltonismo. Los dos gráficos de abajo se ven similares, sin embargo hay suficiente diferencia en los tonos rojo y verde tal que los puntos de la derecha pueden distinguirse incluso por personas con daltonismo rojo-verde.

```
ggplot(millas, aes(cilindrada, autopista)) +
  geom_point(aes(color = traccion))

ggplot(millas, aes(cilindrada, autopista)) +
  geom_point(aes(color = traccion)) +
  scale_colour_brewer(palette = "Set1")
```

Copy

No olvides las técnicas más simples. Si solo hay unos pocos colores, puedes agregar un mapeo de forma redundante. Esto también ayudará a asegurar que tu gráfico sea interpretable en blanco y negro.

```
ggplot(millas, aes(cilindrada, autopista)) +
  geom_point(aes(color = traccion, shape = traccion)) +
  scale_colour_brewer(palette = "Set1")
```

Copy

Las escalas de ColorBrewer están documentadas en línea en http://colorbrewer2.org/ y están disponibles en R en el paquete **RColorBrewer** de Erich Neuwirth. La figura 28.2 muestra la lista completa de paletas de colores disponibles. Las paletas secuenciales (arriba) y divergentes (abajo) son particularmente útiles si sus valores categóricos están ordenados o tienen un "centro". Esto a menudo ocurre si has utilizado `cut()` para convertir una variable continua en una variable categórica.

Figure 28.2: Todas las escalas ColorBrewer.

Cuando tengas un mapeo predefinido entre valores y colores, usa `scale_colour_manual()`. Por ejemplo, si mapeamos los partidos presidenciales de Estados Unidos en color, queremos usar el mapeo estándar de color rojo para republicanos y azul para demócratas:

```
presidencial %>%
  mutate(id = 33 + row_number()) %>%
  ggplot(aes(inicio, id, colour = partido)) +
  geom_point() +
  geom_segment(aes(xend = fin, yend = id)) +
  scale_colour_manual(values = c(Republicano = "red", Demócrata = "blue"))
```

Copy

Para generar una escala de color para variables continuas , puedes usar `built in scale_colour_gradient()` o `scale_fill_gradient()`. Si tienes una escala divergente, puedes usar `scale_colour_gradient2()`. Eso tepermite dar, por ejemplo, diferentes colores a valores positivos y negativos. Esto a veces también es útil si quieres distinguir puntos por encima o por debajo de la media.

Otra opción es `scale_colour_viridis()` proporcionada por el paquete **viridis**. Es un análogo continuo de las escalas categóricas de ColorBrewer. Los diseñadores, Nathaniel Smith y Stéfan van der Walt, adaptaron cuidadosamente una paleta de color para variables continuas que tiene buenas propiedades perceptuales. Aquí hay un ejemplo de viridis:

```
df <- tibble(
  x = rnorm(10000),
  y = rnorm(10000)
)
ggplot(df, aes(x, y)) +
  geom_hex() +
  coord_fixed()

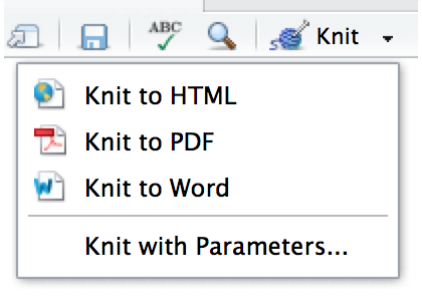
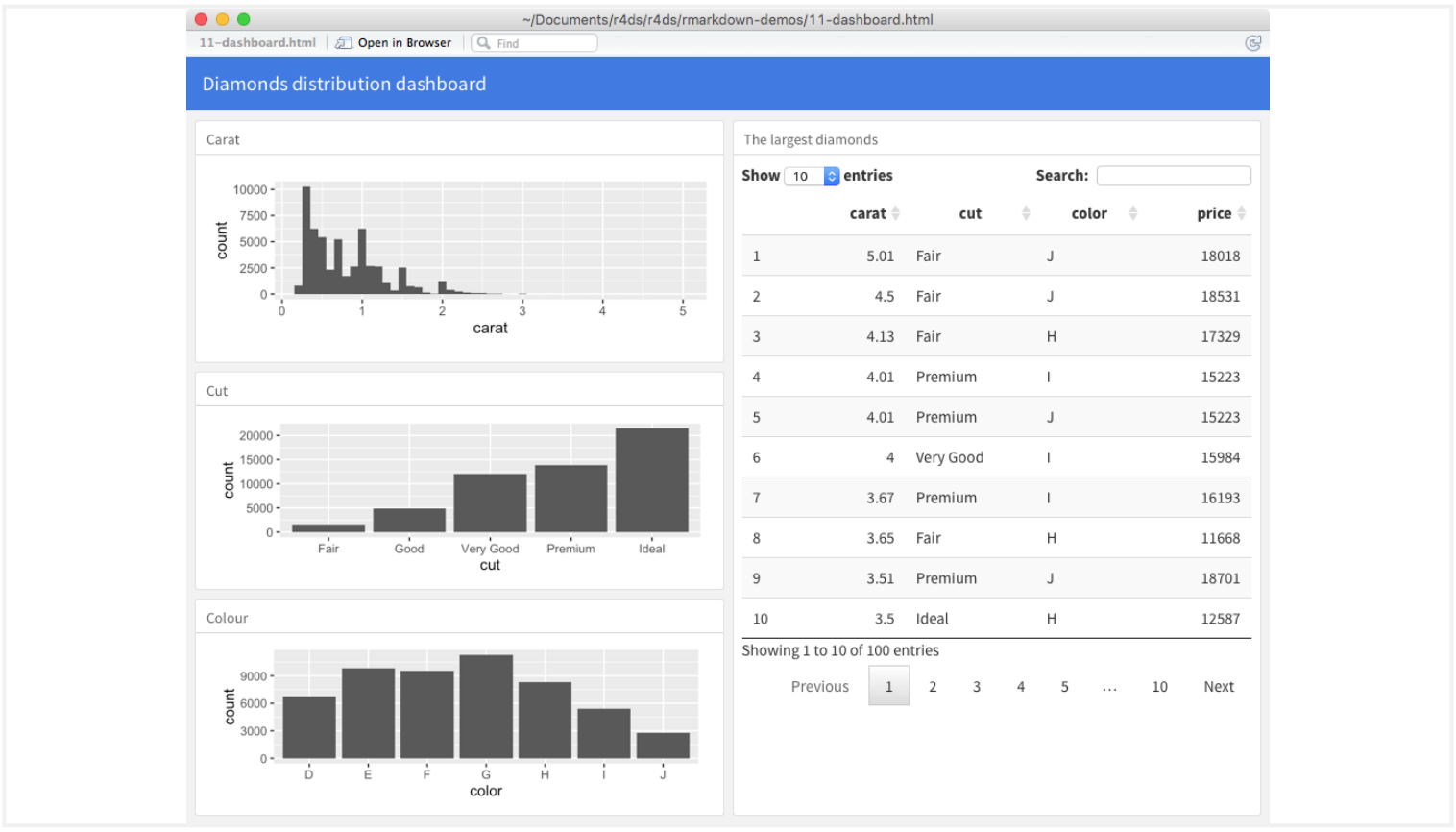
ggplot(df, aes(x, y)) +
  geom_hex() +
  viridis::scale_fill_viridis() +
  coord_fixed()
```

Copy

Ten en cuenta que todas las escalas de color vienen en dos variedades: `scale_colour_x()` y `scale_fill_x()` para la estética `color` y `fill`, respectivamente (las escalas de color se expresan tanto en inglés americano como británico).

28.4.4 Ejercicios

1. ¿Por qué el siguiente código no reemplaza la escala predeterminada?

```
ggplot(df, aes(x, y)) +
 geom_hex() +
 scale_colour_gradient(low = "white", high = "red") +
 coord_fixed()
```

Copy

2. ¿Cuál es el primer argumento para cada escala? ¿Cómo se compara con `labs()`?

3. Cambia la visualización de los términos presidenciales de las siguientes maneras:

I. Combinando las dos variantes que se muestran arriba.

II. Mejorando la visualización del eje y.

III. Etiquetando cada término con el nombre del presidente.

IV. Agregandoetiquetas informativas al gráfico.

V. Poniendo intervalos de 4 años (¡esto es más complicado de lo que parece!).

4. Utiliza `override.aes` para que la leyenda en el siguiente gráfico sea más fácil de ver:

```
ggplot(diamantes, aes(quilate, precio)) +
 geom_point(aes(colour = corte, alpha = 1 / 20))
```

Copy

28.5 Haciendo Zoom

Hay tres formas de controlar los límites de un gráfico:

1. Modificando los datos que se grafican
2. Estableciendo los límites en cada escala
3. Estableciendo `xlim` y `ylim` en `coord_cartesian()`

Para ampliar una región del gráfico, generalmente es mejor usar `coord_cartesian()`.Compara los siguientes dos gráficos:

```
ggplot(millas, mapping = aes(cilindrada, autopista)) +
  geom_point(aes(color = clase)) +
  geom_smooth() +
  coord_cartesian(xlim = c(5, 7), ylim = c(10, 30))

millas %>%
  filter(cilindrada >= 5, cilindrada <= 7, autopista >= 10, autopista <= 30) %>%
  ggplot(aes(cilindrada, autopista)) +
  geom_point(aes(color = clase)) +
  geom_smooth()
```

Copy

También puedes establecer `limits` del inglés *límites* en escalas individuales. La reducción de los límites del gráfico es equivalente a seleccionar un subconjunto de los datos. En general es más útil si deseas *expandir* los límites, por ejemplo, cuando quieres hacer coincidir escalas de diferentes gráficos. A modo de ejemplo, si extraemos dos clases de automóviles y los graficamos por separado, son difíciles de comparar ya que las tres escalas (el eje x, el eje y y la estética del color) tienen rangos diferentes.

```
suv <- millas %>% filter(clase == "suv")
compacto <- millas %>% filter(clase == "compacto")

ggplot(suv, aes(cilindrada, autopista, colour = traccion)) +
  geom_point()

ggplot(compacto, aes(cilindrada, autopista, colour = traccion)) +
  geom_point()
```

Copy

Una forma de superar este problema es compartir la escala entre varios gráficos, estableciendo una escala única a partir de los límites del conjunto de datos completo.

```
x_scale <- scale_x_continuous(limits = range(millas$cilindrada))
y_scale <- scale_y_continuous(limits = range(millas$autopista))
col_scale <- scale_colour_discrete(limits = unique(millas$traccion))

ggplot(suv, aes(cilindrada, autopista, colour = traccion)) +
  geom_point() +
  x_scale +
  y_scale +
  col_scale

ggplot(compacto, aes(cilindrada, autopista, colour = traccion)) +
  geom_point() +
  x_scale +
  y_scale +
  col_scale
```

Copy

En este caso particular podrías haber simplemente empleado la separación en facetas, pero esta técnica es más útil en general, por ejemplo, si deseas realizar gráficos en varias páginas de un informe.

28.6 Temas

Finalmente, puedes personalizar los elementos de tu gráfico que no son datos aplicando un tema:

```
ggplot(millas, aes(cilindrada, autopista)) +
  geom_point(aes(color = clase)) +
  geom_smooth(se = FALSE) +
  theme_bw()
```

Copy

ggplot2 incluye ocho temas por defecto, como se muestra en la Figura 28.3. Muchos otros están incluidos en paquetes adicionales como **ggthemes** (https://github.com/jrnold/ggthemes), de Jeffrey Arnold.

Figure 28.3: The eight themes built-in to ggplot2.

Muchas personas se preguntan por qué el tema predeterminado tiene un fondo gris. Esta fue una elección deliberada ya que el fondo gris pone los datos por delante mientras siguen siendo visibles las líneas de la cuadrícula. Las líneas blancas de la cuadrícula son visibles (lo cual es importante porque ayudan significativamente a evaluar la posición), pero tienen poco impacto visual y son fáciles de eliminar. El fondo gris le da a al gráfico un color tipográfico similar al del texto, asegurando que los gráficos encajen con el flujo de un documento sin saltar con un fondo blanco brillante. Finalmente, el fondo gris crea un campo continuo de color que asegura que el gráfico se perciba como una sola entidad visual.

También es posible controlar componentes individuales de cada tema, como el tamaño y el color de la fuente utilizada para el eje y. Desafortunadamente, este nivel de detalle está fuera del alcance de este libro, por lo que deberás leer el libro ggplot2 book para obtener todos los detalles. También puedes crear tus propios temas, si estás tratando de hacer coincidir un estilo corporativo o de revista en particular.

28.7 Guardando tus gráficos

Hay dos formas principales de obtener tus gráficos desde R: `ggsave()` y `knitr. ggsave()` guardarán el gráfico más reciente en el disco.

```
ggplot(millas, aes(cilindrada, autopista)) + geom_point()
```

Copy

```
ggsave("mi_grafico.pdf")
#> Saving 7 x 4.33 in image
```

Copy

Si no especificas `width` y `height` , del inglés el *ancho* y el *alto*, se usarán las dimensiones del dispositivo empleado para graficar. Para que el código sea reproducible, necesitarás especificarlos.

En general, sin embargo, creo que deberías armar tus informes finales utilizando R Markdown, por lo que quiero centrarme en las opciones importantes para los bloques de código que debes conocer para graficar. Puedes obtener más información sobre `ggsave()` en la documentación.

28.7.1 Redimensionar una figura

El mayor desafío de los gráficos en R Markdown es conseguir que tus figuras tengan el tamaño y la forma correctos. Hay cinco opciones principales que controlan el tamaño de la figura: `fig.width` , `fig.height` , `fig.asp` , `out.width` y `out.height` . El tamaño de la imagen es un desafío porque hay dos tamaños (el tamaño de la figura creada por R y el tamaño al que se inserta en el documento de salida) y varias formas de especificarlo (es decir, altura, ancho y relación de aspecto: elige dos de tres).

Solo uso tres de las cinco opciones:

- Encuentro estéticamente más agradable que los gráficos tengan un ancho consistente. Para hacer cumplir esto, configuro `fig.width = 6` (6 ") y `fig.asp = 0.618` (la proporción áurea) en los valores predeterminados. Luego, en bloques individuales, solo ajusto `fig.asp` .

- Controlo el tamaño de salida con `out.width` y lo configuro a un porcentaje del ancho de línea). De manera predeterminada, `out.width = "70%"` y `fig.align = "center"` . Eso le da a los gráficos cierto espacio para respirar, sin ocupar demasiado espacio.

- Para poner múltiples gráficos en una sola fila, establezco `out.width en 50%` para dos gráficos, `33%` en 3 gráficos, o `25%` en 4 gráficos, y `setfig.align = "default"` . Dependiendo de lo que intento ilustrar (por ejemplo, mostrar datos o variacionesdel gráfico), también modificaré `fig.width` cómo se explica a continuación.

Si observas que tienes que entrecerrar los ojos para leer el texto de tu gráfico, debes ajustar `fig.width`. Si `fig.width` es mayor que el tamaño de la figura en el documento final, el texto será demasiado pequeño; si `fig.width` es más pequeño, el texto será demasiado grande. A menudo necesitarás experimentar un poco para calcular la proporción correcta entre `fig.width` y el ancho asociado en tu documento. Para ilustrar el principio, los siguientes tres gráficos tienen `fig.width` de 4, 6 y 8, respectivamente:

Si deseas asegurarte que el tamaño de fuente es el mismo en todas tus figuras, al establecer `out.width`, también necesitarás ajustar `fig.width` para mantener la misma proporción en relación al `out.width` predeterminado. Por ejemplo, si tu valor predeterminado de `fig.width` es 6 y `out.width` es 0.7, cuando establezcas `out.width = "50%"` necesitarás establecer `fig.width` a 4.3 (6 * 0.5 / 0.7).

28.7.2 **Otras opciones importantes**

Al mezclar código y texto, como hago en este libro, recomiendo configurar `fig.show = "hold"` para que los gráficos se muestren después del código. Esto tiene el agradable efecto secundario de obligarte a dividir grandes bloques de código con sus explicaciones.

Para agregar un título al gráfico, usa `fig.cap`. En R Markdown esto cambiará la figura de "inline" a "floating".

Si estás produciendo resultados en formato PDF, el tipo de gráficos predeterminado es PDF. Esta es una buena configuración predeterminada porque los PDF son gráficos vectoriales de alta calidad. Sin embargo, pueden generar gráficos muy grandes y lentos si muestras miles de puntos. En ese caso, configura `dev = "png"` para forzar el uso de PNG. Son de calidad ligeramente inferior, pero serán mucho más compactos.

Es una buena idea darles nombres a los bloques de código que producen figuras, incluso si no etiquetas rutinariamente otros bloques. Etiquetar el bloque se utiliza para generar el nombre de archivo del gráfico en el disco, por lo que darle un nombre a los bloques hace que sea mucho más fácil seleccionar gráficas y reutilizarlas en otras circunstancias (por ejemplo, si deseas colocar rápidamente un solo gráfico en un correo electrónico o un tweet).

28.8 **Aprendiendo más**

El mejor lugar para aprender más es el libro de ggplot2: *ggplot2: Elegant graphics for data analysis*. Este explica con mucha más profundidad la teoría subyacente y tiene muchos más ejemplos de cómo combinar las piezas individuales para resolver problemas prácticos. Desafortunadamente, el libro no está disponible en línea de forma gratuita, aunque puede encontrar el código fuente en https://github.com/hadley/ggplot2-book.

Otro gran recurso es la guía de extensiones ggplot2 https://exts.ggplot2.tidyverse.org/gallery/. Este sitio enumera muchos de los paquetes que amplían **ggplot2** con nuevos geoms y escalas. Es un buen lugar para comenzar si tratas de hacer algo que parece difícil con **ggplot2**.

« 27 R Markdown

29 Formatos de R Markdown »

29 Formatos de R Markdown

29.1 Introducción

Hasta ahora has visto R Markdown usado para producir documentos HTML. Este capítulo muestra una breve descripción de algunos de los muchos otros tipos de documentos que puedes generar con R Markdown. Hay dos maneras de definir el output de un documento:

1. De forma permanente, modificando el encabezado YAML:

```
title: "Demo Viridis"
output: html_document
```

2. De forma transitoria, llamando `rmarkdown::render()` directamente:

```
rmarkdown::render("diamond-sizes.Rmd", output_format = "word_document")
```

   Esto es útil si quieres producir múltiples tipos de outputs programáticamente.

El botón knit de RStudio genera un archivo con el primer tipo de formato listado en el campo `output` . Puedes generar archivos en formatos adicionales haciendo clic en el menú de selección al lado del botón knit.



29.2 Opciones de salida

Cada formato de salida está asociado con una función de R. Puedes escribir `foo` o `pkg::foo` . Si omites `pkg` , por defecto se asume que es rmarkdown. Es importante conocer el nombre de la función que genera el documento de salida, porque así es como obtienes ayuda. Por ejemplo, para saber qué parámetros puedes definir con `html_document` , busca en `?rmarkdown::html_document` .

Para sobrescribir los parámetros predeterminados necesitas usar un campo de `output` extendido. Por ejemplo, si quisieras generar un `html_document` con una tabla de contenido flotante, deberías usar:

```
output:
  html_document:
    toc: true
    toc_float: true
```

Incluso puedes generar múltiples salidas suministrando una lista de formatos:

```
output:
  html_document:
    toc: true
    toc_float: true
  pdf_document: default
```

Nota la sintaxis especial si no quieres modificar ninguna de las opciones por defecto: debes agregar `default` .

29.3 **Documentos**

El capítulo anterior se enfocó en en la salida por defecto, que es `html_document`. Sin embargo, hay un número de variaciones básicas para generar diferentes tipos de documentos:

- `pdf_document` crea un PDF con LaTeX (un sistema de código abierto de composición de textos), que necesitarás instalar. RStudio te notificará si no lo tienes.
- `word_document` para documentos de Microsoft Word (`.docx`).
- `odt_document` para documentos de texto OpenDocument (`.odt`).
- `rtf_document` para documentos de Formato de Texto Enriquecido (`.rtf`).
- `md_document` para documentos Markdown. Típicamente no es muy útil en sí mismo, pero puedes usarlo si, por ejemplo, tu CMS corporativo o tu wiki de laboratorio usa markdown.
- `github_document`: esta es una versión de `md_document` específicamente diseñada para compartir en GitHub.

Recuerda que cuando generes un documento para compartirlo con responsables de la toma de decisiones, puedes desactivar la visualización predeterminada de código, definiendo las opciones globales en el fragmento de configuración (setup):

```
knitr::opts_chunk$set(echo = FALSE)
```

Copy

Otra opción para los `html_document` es hacer que los fragmentos de código estén escondidos por defecto, pero visibles con un clic:

```
output:
  html_document:
    code_folding: hide
```

Copy

29.4 **Notebooks**

Un notebook `html_notebook` es una variación de un `html_document`. Los outputs de los dos documentos son muy similares, pero tienen propósitos distintos. Un `html_document` está enfocado en la comunicación con personas encargadas de la toma de decisiones, mientras que un notebook está enfocado en colaborar con otros/as científicos/as de datos. Estos propósitos diferentes llevan a que la salida HTML sea usada de diferentes maneras. Ambas contendrán todo el output renderizado, pero el notebook también contendrá el código fuente completo. Esto significa que puedes usar el archivo `.nb.html` generado por el notebook de dos maneras:

1. Puedes verlo en un navegador web y ver el output generado. A diferencia del `html_document`, esta renderización siempre incluye una copia incrustada del código fuente que lo generó.
2. Puedes editarlo en RStudio. Cuando abras un archivo `.nb.html`, RStudio automáticamente recreará el archivo `.Rmd` que lo creó. En el futuro, también podrás incluir archivos de soporte (por ej., archivos de datos `.csv`), que serán extraídos automáticamente cuando sea necesario.

Enviar archivos `.nb.html` por correo electrónico es una manera simple de compartir los análisis con tus colegas. Pero las cosas se pondrán difíciles tan pronto como quieras hacer cambios. Si esto empieza a suceder, es un buen momento para aprender Git y GitHub. Aprender Git y Github definitivamente es doloroso al principio, pero la recompensa de la colaboración es enorme. Como se mencionó anteriormente, Git y GitHub están fuera del alcance de este libro, pero este es un consejo útil si ya los estás usando: usa las dos salidas, `html_notebook` y `github_document`:

```
output:
  html_notebook: default
  github_document: default
```

Copy

`html_notebook` te da una vista previa local y un archivo que puedes compartir por correo electrónico. `github_document` crea un archivo md mínimo que puedes ingresar en Git. Puedes revisar fácilmente cómo los resultados de tus análisis (no solo el código) cambian con el tiempo y GitHub lo renderizará muy bien en línea.

29.5 **Presentaciones**

También puedes usar R Markdown para crear presentaciones. Obtienes menos control visual que con herramientas como Keynote y PowerPoint, pero ahorrarás mucho tiempo insertando automáticamente los resultados de tu código R en una presentación. Las presentaciones funcionan dividiendo tu contenido en diapositivas, con una nueva diapositiva que comienza en cada encabezado de primer nivel (`#`) o de segundo nivel (`##`). También puedes insertar una regla horizontal (`***`) para crear una nueva diapositiva sin encabezado.

R Markdown viene con tres formatos de presentación integrados:

1. `ioslides_presentation` - Presentación HTML con ioslides.
2. `slidy_presentation` - Presentación HTML con W3C Slidy.
3. `beamer_presentation` - Presentación PDF con LaTeX Beamer.

Otros dos formatos populares son proporcionados por paquetes:

1. `revealjs::revealjs_presentation` - Presentación HTML con reveal.js. Requiere el paquete **revealjs**.
2. **rmdshower**, https://github.com/MangoTheCat/rmdshower, proporciona un wrapper para el motor de presentaciones **shower**, https://github.com/shower/shower .

29.6 **Dashboards**

Los dashboards o tableros de control son una forma útil de comunicar grandes cantidades de información de forma visual y rápida. Flexdashboard hace que sea particularmente fácil crear dashboards usando R Markdown y proporciona una convención de cómo los encabezados afectan el diseño:

- Cada encabezado de Nivel 1 (`#`) comienza una nueva página en el dashboard.
- Cada encabezado de Nivel 2 (`##`) comienza una nueva columna.
- Cada encabezado de Nivel 3 (`###`) comienza una nueva fila.

Por ejemplo, puedes producir este dashboard:



Usando este código:

```
---
title: "Dashboard de distribución de diamantes"
output: flexdashboard::flex_dashboard
---

```{r setup, include = FALSE}
library(datos)
library(ggplot2)
library(dplyr)
knitr::opts_chunk$set(fig.width = 5, fig.asp = 1 / 3)
```

Columna 1

Quilate

```{r}
ggplot(diamantes, aes(quilate)) + geom_histogram(binwidth = 0.1)
```

Corte

```{r}
ggplot(diamantes, aes(corte)) + geom_bar()
```

Color

```{r}
ggplot(diamantes, aes(color)) + geom_bar()
```

Columna 2

Diamantes más grandes

```{r}
diamantes %>%
  arrange(desc(quilate)) %>%
  head(100) %>%
  select(quilate, corte, color, precio) %>%
  DT::datatable()
```
```

Copy

Flexdashboard también proporciona herramientas simples para crear barras laterales, tabuladores, cuadros de valores y medidores. Para obtener más información (en inglés) acerca de Flexdashboard visita http://rmarkdown.rstudio.com/flexdashboard/.

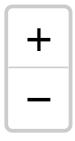
29.7 Interactividad

Cualquier formato HTML (documento, notebook, presentación o dashboard) puede contener componentes interactivos.

29.7.1 htmlwidgets

HTML es un formato interactivo y puedes aprovechar esa interactividad con **htmlwidgets**, que son funciones de R que producen visualizaciones HTML interactivas. Por ejemplo, fíjate en el mapa de **leaflet** a continuación. Si estás viendo esta página en la web, puedes arrastrar el mapa, acercar y alejar, etc. Obviamente no puedes hacer esto en un libro, por lo que RMarkdown automáticamente inserta una captura de pantalla estática.

```
library(leaflet)
leaflet() %>%
  setView(174.764, -36.877, zoom = 16) %>%
  addTiles() %>%
  addMarkers(174.764, -36.877, popup = "Maungawhau")
```



Lo bueno de htmlwidgets es que no necesitas saber nada sobre HTML o JavaScript para usarlos. Todos los detalles están incluidos en el paquete, por lo que no tienes que preocuparte por eso.

Hay muchos paquetes que proporcionan htmlwidgets, incluyendo:

- **dygraphs**, http://rstudio.github.io/dygraphs/, para visualizaciones interactivas de series de tiempo.
- **DT**, http://rstudio.github.io/DT/, para tablas interactivas.
- **threejs**, https://github.com/bwlewis/rthreejs, para plots 3D interactivos.
- **DiagrammeR**, http://rich-iannone.github.io/DiagrammeR/, para diagramas (como diagramas de flujo y diagramas simples de nodos).

Para obtener más información sobre los htmlwidgets y ver una lista más completa de los paquetes que los proporcionan, visita http://www.htmlwidgets.org/.

29.7.2 **Shiny**

Los htmlwidgets proporcionan interactividad del lado del cliente, es decir que toda la interactividad ocurre en el navegador, independientemente de R. Por un lado, esto es bueno porque puedes distribuir el archivo HTML sin ninguna conexión con R. Sin embargo, también limita fundamentalmente lo que puedes hacer a las cosas que han sido implementadas en HTML y JavaScript. Una alternativa es usar **shiny**, un paquete que te permite crear interactividad usando código R, no JavaScript.

Para llamar código Shiny desde un documento R Markdown, agrega `runtime: shiny` al encabezado:

```
title: "Applicación Web Shiny"
output: html_document
runtime: shiny
```

Luego puedes usar las funciones de "input" para agregar componentes interactivos al documento:

```
library(shiny)
textInput("nombre", "Cuál es tu nombre?")
numericInput("edad", "Cuántos años tienes?", NA, min = 0, max = 150)
```

Después puedes referirte a los valores con `input$nombre` e `input$edad` y el código que los usa se volverá a ejecutar automáticamente cada vez que los valores cambien.

No podemos mostrarte una aplicación Shiny corriendo ahora porque las interacciones de Shiny ocurren en el lado del servidor. Esto quiere decir que puedes escribir aplicaciones interactivas sin saber JavaScript, pero necesitas un servidor para correrlas. Esto introduce un problema logístico: Las aplicaciones Shiny necesitan un servidor para correr en línea. Cuando corres una aplicación Shiny en tu propia computadora, Shiny automáticamente crea un servidor Shiny para que la puedas correr, pero necesitarás un servidor Shiny público si quieres proporcionar este tipo de interactividad en línea. Ésta es la limitación fundamental de Shiny: puedes crear en Shiny todo lo que puedes crear en R, pero necesitarás que R esté corriendo en algún lugar.

Aprende más sobre Shiny en: http://shiny.rstudio.com/.

29.8 Sitios Web

Con un poco más de infraestructura puedes usar R Markdown para crear un sitio web completo:

- Coloca todos tus archivos `.Rmd` en un mismo directorio. `index.Rmd` será tu página de inicio.
- Agrega un archivo YAML llamado `_site.yml` que proporcionará la navegación para el sitio. Por ejemplo:

```
name: "mi-sitio"
navbar:
  title: "Mi Sitio"
  left:
    - text: "Inicio"
      href: index.html
    - text: "Colores Viridis"
      href: 1-example.html
    - text: "Colores Terrain"
      href: 3-inline.html
```

Copy

Ejecuta `rmarkdown::render_site()` para crear `_site`, un directorio de archivos listo para ser implementado como un sitio web estático independiente, o si usas un proyecto de RStudio para el directorio de tu sitio web, RStudio agregará una pestaña de compilación que puedes usar para crear y obtener una vista previa de tu sitio.

Lee más acerca de sitios web en: http://rmarkdown.rstudio.com/rmarkdown_websites.html.

29.9 Otros Formatos

Otros paquetes proveen incluso más formatos de salida:

- El paquete **bookdown**, https://github.com/rstudio/bookdown, hace que crear libros, como este mismo, sea fácil. Para aprender más, lee *Authoring Books with R Markdown*, de Yihui Xie, el que, por supuesto, escrito en bookdown. Visita http://www.bookdown.org para ver otros libros bookdown escritos por la comunidad de R.
- El paquete **prettydoc**, https://github.com/yixuan/prettydoc/, proporciona formatos livianos de documentos con una gama de temas atractivos.
- El paquete **rticles**, https://github.com/rstudio/rticles, compila una selección de formatos específicos para revistas científicas.

Consulta http://rmarkdown.rstudio.com/formats.html para ver una lista de más formatos. También puedes crear tu propio formato siguiendo las instrucciones en inglés de: http://rmarkdown.rstudio.com/developer_custom_formats.html.

29.10 Aprende más

Para obtener más información sobre comunicación efectiva con estos diferentes formatos, recomiendo los siguientes recursos (en inglés):

On this page

View source
Edit this page

- Para mejorar tus habilidades de presentación, recomiendo: *Presentation Patterns*, de Neal Ford, Matthew McCollough y Nathaniel Schutta. Proporciona un conjunto de patrones efectivos (de alto y bajo nivel) que puedes usar para mejorar tus presentaciones.

- Si das charlas académicas, recomiendo que leas la guía para dar charlas del grupo Leek: *Leek group guide to giving talks*.

- Nosotros no lo hemos tomado, pero hemos escuchado buenos comentarios sobre el curso en línea Public Speaking de Matt McGarrity: https://www.coursera.org/learn/public-speaking.

- Si estás creando muchos dashboards, asegúrate de leer: Information Dashboard Design: The Effective Visual Communication of Data. Te ayudará a crear dashboards realmente útiles y no solo bonitos a la vista.

- Comunicar tus ideas efectivamente a menudo se beneficia de un poco de conocimiento en diseño gráfico. El libro de diseño para no diseñadores, *The Non-Designer's Design Book*, es un buen lugar para empezar.

« 28 Comunicar con gráficos

30 Flujo de trabajo en R Markdown »

30 Flujo de trabajo en R Markdown

Anteriormente discutimos un flujo de trabajo básico para capturar tu código de R en el que trabajas interactivamente en la *consola* y luego capuras los que funciona en el *editor de script*. R Markdown une la *consola* y el *editor de script*, desdibujando los límites entre exploración interactiva y captura de código a largo plazo. Puedes iterar rápidamente dentro un bloque, editando y re-ejecutando con Cmd/Ctrl + Shift + Enter. Cuando estés conforme, puedes seguir adelante e iniciar un nuevo bloque.

R Markdown es importante también ya que integra de manera estrecha prosa y código. Esto hace que sea un gran **cuaderno de análisis**, porque permite desarrollar código y registrar tus pensamientos. Un cuaderno de análisis comparte muchos de los mismos objetivos que tiene un cuaderno de laboratorio clásico en las ciencias físicas. Puede:

- Registrar qué se hizo y por qué se hizo. Independientemente de cuán buena sea tu memoria, si no registras lo que haces llegará un momento en que habrás olvidado detalles importantes.
- Apoyar el pensamiento riguroso. Es mas probable que logres un análisis sólido si registras tus pensamientos mientras avanzas y continuas reflexionando sobre ellos. Esto también te ahorra tiempo cuando eventualmente escribas tu análisis para compartir con otros.
- Ayudar a que otras personas comprendan tu trabajo. Es raro hacer un análisis de datos por sola/o; con frecuencia trabajarás como parte de un equipo. Un cuaderno de laboratorio ayuda a que compartas no solo lo que has hecho, sino también por qué lo hiciste con tus colegas.

Muchos de estos buenos consejos sobre el uso efectivo de cuadernos de laboratorio pueden también ser aplicados a los cuadernos de análisis. Hemos extraído de nuestras propias experiencias y los consejos de Colin Purrington sobre cuadernos de laboratorio (http://colinpurrington.com/tips/lab-notebooks) para sugerir los siguientes consejos:

- Asegúrate de que cada cuaderno tenga un título descriptivo, un nombre de archivo evocativo y un primer párrafo que describa brevemente los objetivos del análisis.
- Utiliza el campo para fecha del encabezado YAML para registrar la fecha en la que comienzas a trabajar en el cuaderno:

```
date: 2016-08-23
```

  Utiliza el formato ISO8601 AAAA-MM-DD para que no haya ambiguedad. ¡Utilízalo incluso si no escribes normalmente fechas de ese modo!

- Si pasas mucho tiempo en una idea de análisis y resulta ser un callejón sin salida, ¡no la elimines! Escribe una nota breve sobre por qué falló y déjala en el cuaderno. Esto te ayudará a evitar ir por el mismo callejón sin salida cuando regreses a ese análisis en el futuro.
- Generalmente, es mejor que hagas la entrada de datos fuera de R. Pero si necesitas registrar un pequeño bloque de datos, establécelo de modo claro usando `tibble::tribble()`.
- Si descubres un error en un archivo de datos, nunca lo modifiques directamente, sino que escribe código para corregir el valor. Explica por qué lo corregiste.
- Antes de concluir el día, asgúrate de que puedes hacer *knit* en archivo (si utilizas **cacheo**, asegúrate de limpiar los cachés). Esto te permitirá corregir cualquier problema mientras el código está todavía fresco en tu mente.
- Si quieres que tu código sea reproducible a largo plazo (es decir, que puedas regresar a ejecutarlo el mes próximo o el año próximo), necesitarás registrar las versiones de los paquetes que tu código usa. Un enfoque riguroso es usar **packrat**, http://rstudio.github.io/packrat/, el cual almacena paquetes en tu directorio de proyecto, o **checkpoint**, https://github.com/RevolutionAnalytics/checkpoint, el que

reinstala paquetes disponibles en una fecha determinada. Un truco rápido es incluir un bloque que ejecute `sessionInfo()` — , esto no te permitirá recrear fácilmente tus paquetes tal y como están hoy, pero por lo menos sabrás cuales eran.

- Crearás muchos, muchos, muchos cuadernos de análisis a lo largo de tu carrera. ¿Cómo puedes organizarlos de modo tal que puedas encontrarlos otra vez en el futuro? Recomendamos almacenarlos en proyectos individuales y tener un buen esquema para nombrarlos.

« 29 Formatos de R Markdown