

aprendeR

Andrea Fernández Conde
2 de julio de 2017

Índice general

| | |
|---|-----------|
| 1. Introducción | 1 |
| 2. R: lo básico | 5 |
| Instalación | 5 |
| Editores | 6 |
| RStudio | 7 |
| ESS | 7 |
| El espacio de trabajo (Workspace) | 7 |
| Directorio de trabajo | 7 |
| Ejemplos básicos | 9 |
| Comandos útiles | 10 |
| Paquetes (<i>libraries</i>) | 11 |
| CRAN | 11 |
| Github | 12 |
| Otras fuentes | 12 |
| Paquetes recomendados | 12 |
| Scripting | 14 |
| rmarkdown | 15 |
| Encabezado y formatos | 15 |
| Knitr chunks | 16 |
| Ayuda y documentación | 16 |
| Optimizando | 19 |
| Material adicional | 19 |
| 3. Estructuras y funciones | 21 |
| Objetos | 21 |
| Funciones | 22 |
| Componentes de una función | 22 |
| El ambiente | 23 |
| Reglas de visibilidad (scoping) | 25 |
| Estructuras de datos | 26 |
| Clases atómicas (atomic classes) | 27 |
| Vectores | 28 |

| | |
|--|-----------|
| Matrices | 34 |
| Listas | 36 |
| Factores (factor) | 40 |
| Data frames | 45 |
| Estructuras de datos fuera de R-básico | 49 |
| Data tables | 49 |
| Tibbles | 51 |
| Objetos importantes | 53 |
| Infinito | 53 |
| No es un número | 53 |
| Valores perdidos (missing values) | 54 |
| Estructuras de control | 54 |
| If | 54 |
| For | 55 |
| Whiles | 56 |
| Repeat - Break | 57 |
| Next | 57 |
| Material adicional | 58 |
| | |
| 4. Vectorización, la familia apply y otros | 59 |
| Subconjuntos de diferentes estructuras de datos | 59 |
| Operadores para extraer subconjuntos | 60 |
| Asignar a un subconjunto | 64 |
| Operadores lógicos | 66 |
| Aplicaciones | 68 |
| Buscarv o buscarh | 68 |
| Muestras aleatorias | 69 |
| Expande bases | 70 |
| Which, intersect y union | 71 |
| Generación de datos a partir de distribuciones | 71 |
| Split-apply-combine | 73 |
| apply | 75 |
| lapply | 76 |
| sapply | 77 |
| mapply | 78 |
| tapply | 79 |
| by | 80 |
| replicate | 81 |
| ¿Puede ser más fácil? | 83 |
| Material adicional | 83 |
| | |
| 5. Herramientas básicas para un proyecto de datos | 85 |
| Importación de datos | 86 |
| rds | 88 |
| separado por * | 88 |

| | |
|---|------------|
| csv (archivo separado por comas) | 89 |
| Microsoft Excel | 90 |
| dbf | 90 |
| IBM SPSS | 91 |
| Stata | 91 |
| SAS | 92 |
| Google Spreadsheet | 92 |
| Google bigquery | 95 |
| Heroku Postgres | 96 |
| rdata | 101 |
| Transformación de datos | 101 |
| Tareas comunes en la manipulación de datos | 102 |
| Extracción de subconjuntos de observaciones | 105 |
| Extracción de subconjuntos de variables (<i>select</i>) | 109 |
| Creación de resúmenes de datos | 111 |
| Creación de nuevas variables | 113 |
| Combinación de conjuntos de datos (<i>joins</i>) | 116 |
| Agrupar datos | 119 |
| Reorganizar datos | 123 |
| Datos limpios | 125 |
| Datos limpios | 128 |
| De sucio a limpio | 129 |
| Visualización | 142 |
| La gramática de las gráficas | 144 |
| ¿Qué es? | 144 |
| Base plotting vs. ggplot | 144 |
| ggplot | 146 |
| Características importantes | 148 |
| Las capas | 148 |
| Componentes de una capa | 149 |
| Material adicional | 156 |
| 6. Conclusión | 157 |
| | |
| A. Markdown | 159 |
| Encabezados | 159 |
| Nivel 1 | 160 |
| Nivel 2 | 160 |
| Lineas horizontales | 160 |
| Énfasis | 160 |
| Bloques | 161 |
| Listas | 161 |
| Links | 162 |

| | |
|---|------------|
| Imágenes | 162 |
| Tablas | 163 |
| HTML | 164 |
| Código | 165 |
| Párrafos | 165 |
| B. Packrat | 167 |
| El directorio del proyecto | 168 |
| Instalación | 168 |
| Inicializarlo | 168 |
| Aregar, remover y actualizar paquetes | 169 |
| Otras cosas importantes | 169 |
| Ligas utiles | 169 |
| Bibliografía | 171 |

Índice de figuras

| | |
|---|-----|
| 1.1. Descargas anuales del espejo de RStudio de paquetes de R de 2012 a 2016 y descargas de R para 2015 y 2016 (en millones) | 2 |
| 4.1. Ejemplificación del split-apply-combine Vaidyanathan (2014, Split-Apply-Combine) | 73 |
| 5.1. Modelo de las herramientas que se necesitan en un proyecto de datos según G. Grolemund y H. Wickham (2016, Introducción) | 85 |
| 5.2. Importación en el análisis de datos G. Grolemund y H. Wickham (2016, Introducción) | 86 |
| 5.3. Transformación de datos G. Grolemund y H. Wickham (2016, Introducción) | 101 |
| 5.4. Joins en el lenguaje SQL (Moffat 2009) | 117 |
| 5.5. Limpieza de datos G. Grolemund y H. Wickham (2016, Introducción) | 125 |
| 5.6. Típica presentación de datos | 126 |
| 5.7. Mismos datos que en 5.6 pero traspuestos | 127 |
| 5.8. Observaciones son filas, variables columnas | 128 |
| 5.9. Ejemplificación de datos limpios (Hadley Wickham y Garrett Grolemund 2016, sección Data Tidying) | 129 |
| 5.10. Datos de pensiones del IMSS e ISSSTE 2000 a 2016 (<i>Cuarto informe de gobierno, 2015-2016. Anexo estadístico.</i> 2016, p. 218-219) | 140 |
| 5.11. Visualización en el análisis de datos G. Grolemund y H. Wickham (2016, Introducción) | 143 |

Índice de cuadros

| | |
|--|-----|
| 3.1. Clases atómicas. | 27 |
| 5.2. Acciones y verbos comunes en la manipulación de datos (datawrangling).. | 103 |
| 5.3. Casos de tuberculosis para México del 2000 al 2010 para hombres con diagnóstico por lesiones de pulmón. | 130 |
| 5.4. Valores de variables en una sola variable. | 131 |
| 5.5. Mediciones de IMC en sujetos. | 132 |
| 5.6. Muestra de datos limpios para experimentos IMC. | 132 |
| 5.7. Cada columna es una variable. | 134 |
| 5.8. Mediciones de temperatura max y min. | 135 |
| 5.9. Paso 1. Juntar las columnas, limpiar dias, crear fecha. | 136 |
| 5.10. Paso 2. Enviar a columnas las mediciones de temperaturas. | 136 |

Capítulo 1

Introducción

R inicia a principios de los noventas en la Universidad de Auckland en Nueva Zelanda. Ross Ihaka, profesor del departamento de estadística, pensaba que debía existir una alternativa superior para el análisis de datos realizado por los alumnos, que utilizaban lo que él llamaba *programas viejos y cuchos*. Robert Gentleman le sugiere a Ross escribir un software cuya ambición inicial era poder enseñar sus cursos de licenciatura de primer año. Así, en 1991 generan una estructura básica a través de la cual sus estudiantes podían hacer análisis de datos y producir modelos gráficos de la información. Lo bautizan R por sus iniciales (Ingenio 2016).

Ross y Robert no comercializan el software sino que lo ponen a disposición de otros interesados. Ross ha expresado que R cambió su opinión acerca de la humanidad pues es el resultado del trabajo de muchos que no reciben ingresos o reconocimiento por el mismo (Ingenio 2016). En 1996, presentan R en un paper introductorio (Ihaka y Gentleman 1996).

A partir de entonces, R ha crecido en forma importante. Entre los contribuidores actuales más relevantes se encuentra Hadley Wickham, alumno de licenciatura en el departamento de estadística de la Universidad de Auckland cuando R se encontraba en desarrollo. En la gráfica siguiente, se muestran las descargas anuales de paquetes de R del 2012 al 2016 del espejo de RStudio¹.

En el 2016 R fue descargado 670,705 veces. El aumento en la popularidad de R no es el único elemento por el cuál R es un lenguaje valioso. Sin embargo, el que sea un lenguaje comúnmente enseñado en universidades y utilizado en empresas, lo convierte en una habilidad con considerable valor de mercado.

En la encuesta de Stackoverflow, R se encuentra en el lugar séptimo de los mejores pagados para los desarrolladores cuya ocupación es matemáticas, superando a Python y a SQL (Stackoverflow 2016, Top paying tech per occupation, mathematics). En cuanto a las tecnologías más populares por tipo de desarrollador que declara dedicarse a matemáticas y datos, R está

¹Estos números representan únicamente una fracción de las descargas de R en el mundo pues existen múltiples espejos del software de donde es posible realizar la descarga. Los datos son tomados de Csardi (2016)

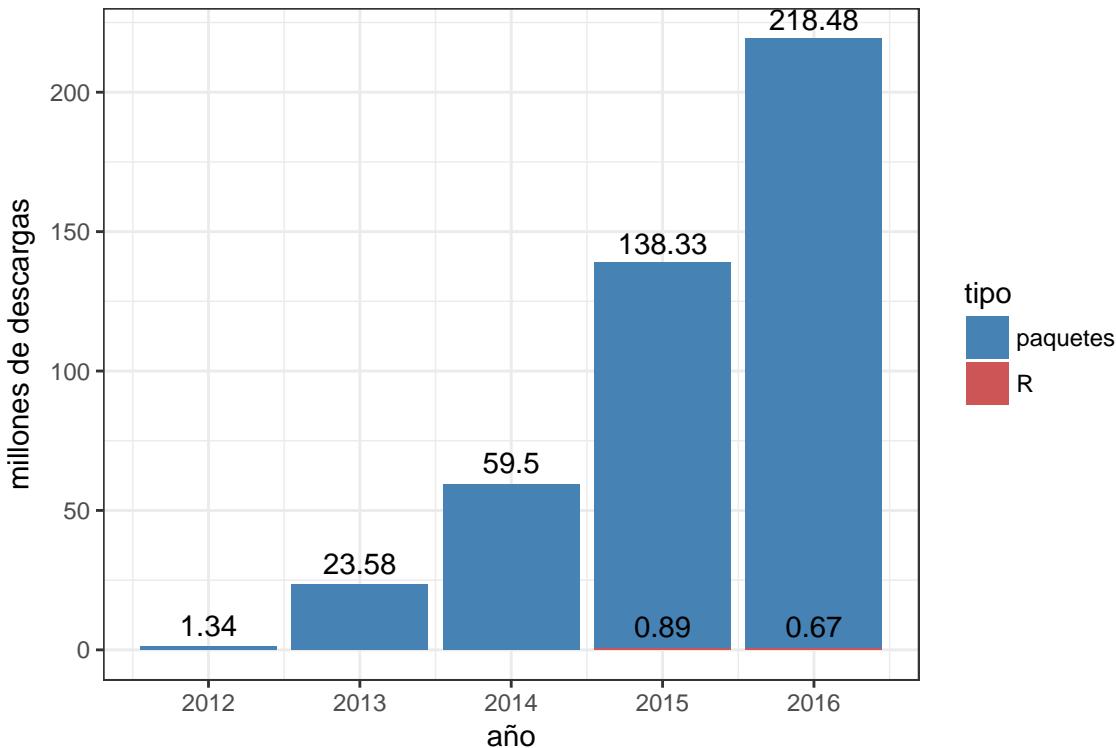


Figura 1.1: Descargas anuales del espejo de RStudio de paquetes de R de 2012 a 2016 y descargas de R para 2015 y 2016 (en millones).

en el sexto lugar, el primer lugar lo tiene **python**, seguido de **SQL** (Stackoverflow 2016, Most Popular Technologies per Dev Type, Math and Data).

Actualmente, **R**, **python** y **SQL** se encuentran entre las herramientas más populares tanto entre desarrolladores como empresas, aunque no son las únicas. La decisión de aprender alguno de estos lenguajes depende de muchos factores, entre ellos cuán natural resulta la interacción individual con cada cuál, el lenguaje preferido en el grupo de trabajo particular y el tipo de análisis que se requiere realizar en el día a día. Escapa del objetivo de este manual el realizar una comparación exhaustiva de tecnologías pero se recomienda tener en cuenta que cada herramienta tiene una especialidad específica y, particularmente en un ambiente de producción, es necesario tener esto en consideración.

R es un excelente lenguaje para aprender ciencia de datos; de hecho en CRAN (2016) se describe a **R** como un proyecto para estadística computacional. Esto lo convierte en un lenguaje único pues fue construido por estadísticos y diseñado para realizar análisis de datos.

Su uso generalizado en la comunidad estadística tiene la ventaja de que casi cualquier prueba o técnica estadística puede ser encontrada en algún paquete de **R** (Labs 2016). Además, existe una documentación extensa y estandarizada que facilita su uso.

Aunque el material para aprender **R** es amplio y hay una comunidad mundial muy activa que constantemente produce nuevos recursos, existen pocas referencias que faciliten iniciar su aprendizaje para hispanoparlantes. En general, la documentación, listas de distribución,

libros y tutoriales están escritos en inglés.

Este manual tiene como objetivo guiar a principiantes en programación que tienen una formación previa como analistas de datos. El enfoque principal es el de facilitar ejemplos que permitan al analista traducir la manipulación de datos que ya saben realizar en otro ambiente a R.

El manual se estructura como sigue: en el capítulo 2, se introducen elementos básicos para poder iniciar el trabajo en R. Se especifica cómo instalar el software, se recomienda utilizar un editor especializado, así como paquetes útiles para diferentes tareas. En particular, se explica cómo guardar código de manera que otras personas puedan ejecutarlo y cómo realizar documentos reproducibles. Por último, se explica cómo accesar a la ayuda y documentación, así como la forma en la que puede optimizarse su funcionamiento. Este capítulo actúa más como una referencia general para poder realizar el trabajo en el ambiente.

En el capítulo 3, se introducen las funciones, las estructuras de datos y las estructuras de control disponibles en el lenguaje. El capítulo 4, explica como operar los objetos y estructuras detallados en el capítulo anterior, proporcionando múltiples ejemplos y ejercicios para familiarizar al lector con el lenguaje.

El capítulo 5, detalla las herramientas básicas para poder realizar un proyecto de datos en R. Las herramientas que se desarrollan en este capítulo permiten iterar sobre parte del ciclo de un proyecto de datos: importación de datos al ambiente, manipulación, limpieza y visualización de los mismos. Éstas herramientas permiten operar sobre los objetos introducidos en el capítulo 3 en una forma eficiente, fácil de aprender, fácil de leer y que permite que el usuario realice manipulaciones de datos complejas que le permitirán, a su vez, utilizar todas las herramientas de modelado que R posee que necesitan como insumo datos limpios y preparados en una forma específica.

Cada capítulo incluye ejercicios y respuestas a los mismos; al final se recomienda material adicional para repasar los conceptos estudiados. El material se encuentra disponible electrónicamente en <https://github.com/animalito/aprendeR>. Para facilitar el aprendizaje, se recomienda descargar los materiales o clonar el repositorio, esto permite revisar el material y el código desde el ambiente local evitando copiar y pegar el mismo para su ejecución.

Capítulo 2

R: lo básico

En este capítulo se revisarán elementos básicos para poder iniciar el trabajo en R. Primero que nada, se proporcionan instrucciones de instalación según el sistema operativo utilizado. Posteriormente, se recomiendan editores que facilitan la edición de código y documentos en R, particularmente `Rstudio` o `Emacs` combinado con `ESS`. Posteriormente, se describe brevemente el espacio de trabajo y se dan algunos ejemplos que ilustran la interacción con la consola.

Se cubren algunos temas útiles para el trabajo continuo en R: se recomiendan paquetes que complementan a la instalación básica de R y que son particularmente útiles para el análisis de datos; se describe qué es un `script` y un documento de R; se explica cómo obtener ayuda y accesar a la documentación.

Por último, se recomienda realizar instalaciones adicionales que permiten optimizar el trabajo de álgebra lineal que soporta los distintos métodos implementados en R.

Instalación

Para los usuarios de Linux, se pueden correr los siguientes comandos en la consola para instalar R compilándolo (Escalante 2015). Ésta es la mejor opción pues se aprovecharán todas las características de su máquina.

```
#!/bin/bash
while true; do
  read -p "Do you wish to Compile R? y/n " yn
  case $yn in
    [Yy]* ) sudo apt-get update;
              sudo apt-get upgrade -y;
              sudo apt-get install -y build-essential libpq-dev liblapack3 libblas3 \
              libmysql++-dev sqlite3 fort77 gnuplot-x11 texinfo liblapack-dev \
              texi2html libglpk-dev libgeos-dev libgdal1-dev libproj-dev;
```

```

sudo apt install -y gfortran autoconf automake bzip2-doc cdbs \
debhelper dh-strip-nondeterminism dh-translations gettext intltool \
intltool-debian libarchive-zip-perl libasprintf-dev libbz2-dev \
libfile-stripnondeterminism-perl libgettextpo-dev libgettextpo0 \
liblzma-dev libmail-sendmail-perl libncurses5-dev libpcre3-dev \
libpcre32-3 libpcrecpp0v5 libreadline-dev libreadline6-dev \
libsystime-long-perl libtinfo-dev libunistring0 m4 po-debconf \
python-scour xorg-dev libcairo2-dev libgtk2.0-dev;
sudo apt-get -y build-dep r-base;
mkdir -p $HOME/src;
cd $HOME/src;
wget -c http://cran.r-project.org/src/base/R-latest.tar.gz;
tar zxvf R-latest.tar.gz && rm R-latest.tar.gz;
cd "$(ls -dt R-* | head -1 )";
./configure --enable-memory-profiling --enable-R-shlib \
--with-blas --with-lapack --with-tcltk --with-cairo \
--with-libpng --with-jpeglib --with-libtiff;
make;
sudo make install;
break;;
[Nn]* ) sudo apt-key adv --keyserver keyserver.ubuntu.com \
--recv-keys E084DAB9;
ubuntu_codename=`lsb_release -cs` ;
sudo chmod ugo+r /etc/apt/sources.list;
echo \
"deb http://cran.r-project.org/bin/linux/ubuntu $ubuntu_codename/" \
>> '/etc/apt/sources.list';
sudo apt-get update;
sudo apt-get install -y --no-install-recommends r-base r-base-dev;
exit;;
* ) echo "Please answer yes or no.";;
esac
done

```

Para descargar e instalar R en su versión precompilada, seguir las instrucciones de este link para el sistema operativo que estén utilizando.

Editores

Hay muchísimos, en particular se mencionarán dos.

RStudio

Puedes descargar RStudio siguiendo las instrucciones para cada sistema operativo. RStudio es un IDE (integrated development environment) para R que incluye consola, editor de texto, memoria de gráficos, vista de objetos en el ambiente y otras herramientas útiles para desarrollar (RStudio Team 2016). En su versión más reciente, también autocompleta código y depura (*debugging*) “al vuelo”, es decir, al mismo tiempo que se escribe, señala potenciales errores de código.

Hay que tener cuidad con el uso de la memoria RAM de este editor pues utiliza muchos recursos de la computadora y -cuando están usando una gran cantidad de datos o procesos muy pesados- RStudio suele detenerse fácilmente. Buenas prácticas en general: guardar seguido, seguir un flujo de trabajo (*workflow*) aunado a controlador de versiones (o algún tipo de respaldo) y, sobretodo, crear las funciones, lógica, algoritmos, con una muestra de los datos.

ESS

Emacs speaks statistics es el add-on favorito para los usuarios de `emacs` y R (Rossini y col. 2004). Soporta la edición de scripts para R, S-plus, SAS, Stata, OPenBUGS/JAGS. Para los que además ya están acostumbrados al enorme poder de Emacs, ésta será la mejor opción.

El editor interactivo es muy bueno y casi no consume memoria.

El espacio de trabajo (Workspace)

El *espacio de trabajo* es el ambiente actual de trabajo en R. Incluye todos los objetos definidos por el usuario (vectores, matrices, funciones, dataframes, listas).

Una sesión de R inicia cuando abres la consola. Al terminar el trabajo se puede guardar la imagen del espacio de trabajo tal cual está, de manera que sea posible continuar *desde donde te quedaste* (Kabacoff 2015, p. 11).

Directorio de trabajo

El directorio de trabajo (*working directory*) es el directorio en tu computadora en el que estás trabajando en ese momento. Cuando se le pide a R que abra un archivo o guarde ciertos datos, R lo hará a partir del directorio de trabajo que le hayas fijado.

Para saber en qué directorio te encuentras, se usa el comando `getwd()`.

Uso la mnemotécnica del inglés: *get working directory* \equiv `getwd`. Notarás como muchas funciones tienen un nombre que acorta lo que hacen.

`getwd()`

```
## [1] "/home/animalito/study/aprendeR/01_programacion_basica"
```

Para especificar el directorio de trabajo, se utiliza el comando `setwd()` (*set working directory*) en la consola. Y volvemos a

```
setwd("/home/animalito/study/")
getwd()
```



Ejercicios

1. Abre tu consola de R y escribe `setwd()`.
2. Utiliza la tecla `tab` para autocompletar las posibles rutas desde donde quiera que estés.
3. Escoge alguna (nuevamente usando la tecla tab para moverte entre las opciones). Si esto no funciona, teclea textualmente alguna de las rutas que ves.
4. Cierra la doble comilla y el paréntesis.
5. Teclea enter.
6. Debes encontrarte en la ruta elegida cuando tecleas `getwd()`.

Con lo que acabamos de hacer, R buscará archivos o guardará archivos en la carpeta que obtuviste con el comando `getwd()`. En R también es posible navegar a partir de el directorio de trabajo. Como siempre,

- `../un_archivo.R` le indica a R que busque una carpeta arriba del actual directorio de trabajo por el archivo `un_archivo.R`.
- `datos/otro_archivo.R` hace que se busque en el directorio de trabajo, dentro de la carpeta `datos` por el archivo `otro_archivo.R`

Rutas relativas vs. Rutas absolutas

El resultado que se muestra aquí al usar el comando `getwd()` depende de la computadora en la que se esta trabajando debido a que es una *ruta absoluta*. Nota como es diferente la ruta que obtienes al correr el comando en tu consola de R. Eso es porque se trata de una ruta absoluta, es decir, es tal que da la ruta (*path*) completo al directorio en cuestión. Puedes accesar todos los directorios o archivos usando su ruta absoluta.

En investigación reproducible (*reproducible research*), en investigación colaborativa o incluso cuando trabajas en varias computadoras es una buena idea usar rutas relativas en lugar de absolutas. Esto hace que el código sea menos dependiente de una estructura de archivos o computadora en particular (Gandrud 2013, p. 67).

En general, es *buenas prácticas* configurar el código de un proyecto con rutas relativas. En R en particular, cuando guardas un Rmarkdown y lo corres desde la línea de comandos (o lo tejes desde RStudio), la ruta que está fija -como si hubieras usado el comando `setwd()` es en donde vive ese archivo, es decir, el directorio en donde está guardado el mismo.

Desde cualquier script puedes llamar a otros usando este tipo de ruta como en el ejemplo anterior.

Ejemplos básicos

La consola permite hacer operaciones sobre números o caracteres (cuando tiene sentido).

```
# Potencias, sumas, multiplicaciones
2^3 + 67 * 4 - (45 + 5)
```

```
## [1] 226
```

```
# Comparaciones
56 > 78
```

```
## [1] FALSE
```

```
34 <= 34
```

```
## [1] TRUE
```

```
234 < 345
```

```
## [1] TRUE
```

```
"hola" == "hola"
```

```
## [1] TRUE
```

```
"buu" != "yay"
```

```
## [1] TRUE
```

```
# módulo
```

```
10 %% 4
```

```
## [1] 2
```

Estas operaciones también pueden ser realizadas entre vectores¹.

```
# Creamos un vector con entradas del -1 al 12 y lo asignamos a la variable x
x <- -1:12
```

¹Revisaremos más adelante con detalle la definición de vectores en la sección 3.

```
# Lo vemos
x

## [1] -1 0 1 2 3 4 5 6 7 8 9 10 11 12

# Le sumamos 1 a todas las entradas
x + 1

## [1] 0 1 2 3 4 5 6 7 8 9 10 11 12 13

# Multiplicamos por 2 cada entrada y le sumamos 3
2 * x + 3

## [1] 1 3 5 7 9 11 13 15 17 19 21 23 25 27

# Sacamos el módulo de cada entrada
x %% 5

## [1] 4 0 1 2 3 4 0 1 2 3 4 0 1 2
```

Comandos útiles

Para enlistar los objetos que están en el espacio de trabajo

```
ls()
```

```
## [1] "misdatos" "mydata"    "x"
```

Para eliminar todos los objetos en un directorio de trabajo (*workspace*)

```
rm(list = ls()) # se puede borrar solo uno, por ejemplo, nombrándolo
ls()
```

```
## character(0)
```

También se puede utilizar/guardar la historia de comandos utilizados

```
history()
history(max.show = 5)
history(max.show = Inf) # Muestra toda la historia

# Se puede salvar la historia de comandos a un archivo
savehistory(file = "mihistoria") # Por default, R ya hace esto
# en un archivo ".Rhistory"

# Cargar al espacio de trabajo actual (current workspace) una
# historia de comandos en particular
loadhistory(file = "mihistoria")
```

Es posible también guardar el ambiente de trabajo (*workspace*) -en forma completa- en un archivo con el comando `save.image()` a un archivo con extensión *.RData*. Puedes guardar una lista de objetos específica a un archivo *.RData*. Por ejemplo:

```
x <- 1:12
y <- 3:45
save(x, y, file = "ejemplo.RData") #la extensión puede ser arbitraria.
```

Después puedo cargar ese archivo. Prueba hacer:

```
rm(list = ls()) # limpiamos workspace
load(file = "ejemplo.RData") #la extensión puede ser arbitraria.
ls()
```

Nota como los objetos preservan el nombre con el que fueron guardados.

Paquetes (*libraries*)

R puede hacer muchos análisis estadísticos y de datos. Las diferentes capacidades están organizadas en paquetes o librerías. Con la instalación estándar resumida en la sección B, se instalan también los paquetes más comunes (también llamado el *base* o R-básico). Para obtener una lista de todos los paquetes instalados se puede utilizar el comando `library()` en la consola o en un script.

Existen una gran cantidad de paquetes disponibles además de los incluidos por omisión (*default*).

CRAN

Comprehensive R Archive Network (CRAN 2016) es una colección de sitios que contienen exactamente el mismo material, es decir, son espejos (*mirrors*) de las distribuciones de R, las extensiones, la documentación y los binarios. El master de CRAN está en Wirtschaftsuniversität Wien en Austria. Éste se “espeja” (*mirrors*) en forma diaria a muchos sitios alrededor del mundo. En la lista de espejos se puede ver que para México están disponibles el espejo del ITAM, del Colegio de Postgraduados (Texcoco) y Jellyfish Foundation (CRAN 2016).

Los espejos son importantes pues, cada vez que busquen instalar paquetes, se les preguntará qué espejo quieren utilizar para la sesión en cuestión. Del espejo que selecciones, será del cuál R *bajará* el binario y la documentación.

Del CRAN es que se obtiene la última versión oficial de R. Diariamente se actualizan los espejos. Para más detalles consultar el FAQ.

Para contribuir un paquete en CRAN se deben seguir las instrucciones aquí.

Github

Git es un controlador de versiones muy popular para desarrollar software. Cuando se combina con GitHub se puede compartir el código con el resto de la comunidad. Éste controlador de versiones es el más popular entre los que contribuyen a R. Muchos problemas a los que uno se enfrenta alguien ya los desarrolló y no necesariamente publicó el paquete en CRAN. Para instalar algún paquete desde GitHub, se pueden seguir las instrucciones siguientes

```
install.packages("devtools")
devtools::install_github("username/packagename")
```

Donde `username` es el usuario de Github y `packagename` es el nombre del repositorio que contiene el paquete. Cuidado, no todo repositorio en GitHub es un paquete. Para más información ver el capítulo Git and GitHub en Hadley Wickham (2015b).

Otras fuentes

Otros lugares en donde es común que se publiquen paquetes es en Bioconductor un proyecto de software para la comprensión de datos del genoma humano.

Paquetes recomendados

Hay muchísimas librerías y lo recomendable es, dado un problema y un modelo para resolverlo, revisar si alguien ya implementó el método en algunas de las fuentes de paquetes mencionadas antes.

Para mantener orden en los paquetes descargados puede ser útil utilizar el Rinker y Kurkiewicz (2015) pues provee de herramientas para instalar paquetes en una forma un poco más sencilla que usando la función `install.packages`.

En particular, la función `p_load` permite instalar, cargar y actualizar uno o varios paquetes.

Si queremos instalar varios paquetes usando las herramientas del R básico (`base`) (R Core Team 2016a) haríamos algo como (ejemplo tomado de Rinker y Kurkiewicz 2015, en la viñeta de introducción al paquete):

```
packs <- c("XML", "devtools", "RCurl", "fakePackage", "SPSSemulate")
success <- suppressWarnings(sapply(packs, require, character.only = TRUE))
install.packages(names(success)[!success])
sapply(names(success)[!success], require, character.only = TRUE)
```

Con `pacman::p_load` la tarea se reduce a:

```
pacman::p_load(XML, devtools, RCurl, fakePackage, SPSSemulate)
```

Nota como se puede llamar a una función por su nombre `p_load` una vez que ya cargamos el paquete en el cuál esa función está guardada con el comando `library(pacman)` o podemos llamarla directamente utilizando la convención `paquete::funcion`, en este caso, `pacman::p_load`.

Para instalar `pacman` escribe:

```
install.packages("pacman")
```

Algunos paquetes se encuentran en desarrollo. En particular, si se encuentran en `github` pueden descargarse usando la función `pacman::p_install_gh('usuario/repositorio')`.

A continuación, hay una lista de paquetes que se recomienda descargar o revisar para tener a la mano herramientas diversas útiles para el trabajo del científico de datos. La lista no es comprensiva pues hay un gran número de paquetes útiles.

```
# Para cargar datos al ambiente de trabajo (data load)
pacman::p_load(RODBC, RMySQL, RPostgreSQL, RSQLite, foreign, Rpostgres, haven
               , readr)
pacman::p_install_gh("hadley/readxl")
pacman::p_install_gh("rstats-db/RPostgres")

# Para manipular datos (data manipulation)
pacman::p_load(plyr, dplyr, data.table, tidyr, stringr, lubridate, gsubfn)

# Para visualizar datos (data visualization)
pacman::p_load(ggplot2, graphics, ggvis)
pacman::p_install_gh("RcppCore/Rcpp")
pacman::p_install_gh("rstats-db/DBI")
pacman::p_install_gh('ramnathv/htmlwidgets')
pacman::p_install_gh('rstudio/leaflet')
pacman::p_install_gh('bwlewis/rthreejs')
pacman::p_install_gh('htmlwidgets/sparkline')
pacman::p_load(dygraphs, DT, DiagrammeR, networkD3, googleVis)

# Para modelar (data modelling)
pacman::p_load(car, mgcv, lme4, nlme, randomForest, multcomp, vcd
               , glmnet, survival, caret)

# Para generar reportes (reports)
pacman::p_load(shiny, xtable, knitr, rmarkdown)

# Para trabajar con datos espaciales (spatial data)
pacman::p_load(sp, maptools, maps, ggmap, rgdal)

# Para trabajo con series de tiempo (time series)
```

```

pacman::p_load(zoo, quantmod)

# Para escribir código de alto rendimiento en R (High performance R code)
pacman::p_load(Rcpp, parallel)

# Trabajar con la web
pacman::p_load(XML, jsonlite, httr)

# Para escribir paquetes en R
pacman::p_load(devtools, testthat, roxygen2)

```

Packrat Cuando cambian las versiones de distintos paquetes de R, es posible que código que solía funcionar deje de hacerlo. Por esta razón, es conveniente empaquetar proyectos de código de manera que el código en un proyecto específico tenga asociados también las versiones específicas de los paquetes con los cuales fue creado. Una forma de lograr esto es utilizando `packrat`. Para mayor detalle, ver el apéndice B.

Scripting

R es un intérprete. Utiliza un ambiente basado en línea de comandos. Por ende, es necesario escribir la secuencia de comandos que se desea realizar a diferencia de otras herramientas en donde es posible utilizar el *mouse* o menús.

Aunque los comandos pueden ser ejecutados directamente en consola una única vez, también es posible guardarlos en archivos conocidos como *scripts*. Típicamente, utilizamos la extensión `.R` o `.r`. En RStudio (RStudio Team 2016), CTRL + SHIFT + N abre inmediatamente un nuevo editor en el panel superior izquierdo.

En RStudio, por ejemplo, se puede *ir editando* el script y corriendo los comandos línea por línea con CTRL + ENTER. Esto también aplica para *correr* una selección del texto editable².

Es posible también ejecutar todo el script

```
source("foo.R")
```

O con el atajo CTRL + SHIFT + S en RStudio.

Para enlistar algunos atajos (*shortcuts*) comunes en RStudio presiona ALT + SHIFT + K.

De la misma manera, si utilizas Emacs + ESS (Rossini y col. 2004), existen múltiples atajos de teclado para realizar todo mucho más eficientemente. Estudiarlos no es tiempo perdido.

²RStudio tiene muchos atajos de teclado que facilitan el trabajo.

rmarkdown

Es posible generar documentos reproducibles en R utilizando R Markdown, un *framework* que permite salvar y ejecutar código, así como generar reportes de alta calidad en múltiples formatos (Inc. 2016b). Para utilizarlo, se instala el paquete **rmarkdown** con el comando:

```
install.packages("rmarkdown")
```

Para generar un documento, se necesitan conocer únicamente algunos elementos importantes. La extensión que se suele utilizar para estos documentos es **.Rmd** o **.rmd**.

Encabezado y formatos

El primer elemento es el encabezado y se conoce como el **yaml** o **front-matter**. Se coloca en la parte superior del documento y corresponde a las opciones que ofrece **pandoc** para la generación de documentos.

Éste contiene la especificación de elementos como el título del documento, autor, fecha, entre otros. Además, se especifica el formato de salida del documento.

Para crear un documento en HTML, por ejemplo, es necesario especificar como el **output** del documento **html_output** y se ve como sigue:

```
---
title: "Un título"
author: "Un autor"
date: "Una fecha"
output:
  html_document:
    toc: yes
    toc_depth: 2
    toc_float: true
    theme: spacelab
---
```

En este ejemplo, se colocaron opciones adicionales para el documento HTML como el que incluya una tabla de contenidos (**toc: yes**), que la profundidad de dicha tabla de contenidos incluya los primeros dos niveles de encabezados (**toc_depth: 2**), que la tabla de contenidos sea flotante -que se encuentre fija en una barra a la izquierda del documento aunque se desplace el documento (**toc_float: true**) y, por último, se especifica el tema para la estética del documento (**theme: spacelab**).

Existen muchas otras opciones, mismas que puedes encontrar en la documentación (Inc. 2016a).

Es posible también especificar como salida para el documento un pdf con la opción **pdf_document**. Las opciones se encuentran también en la documentación (Inc. 2016c). Esta

opción es conveniente cuando se tiene conocimiento previo de LaTeX.

Por último, cabe mencionar la opción `word_document`, cuyas opciones se encuentran aquí (Inc. 2016e) y la opción `md_document` que compila a Markdown.

Así como es posible generar documentos, es posible crear presentaciones en HTML (ioslides o slidy) o pdf (beamer) (Inc. 2016d) y dashboards (con flexdashboards) (Allaire 2016).

En el apéndice A se detalla la sintaxis de Markdown, misma que permitirá escribir documentos.

Knitr chunks

Entre distintas líneas de texto, es posible incluir **chunks** o *pedazos* de código de R. Para iniciar un pedazo de código, se incluyen tres acentos invertidos, seguidos de la letra `r` entre llaves; se cierra un pedazo de código con tres acentos invertidos (Inc. 2016d).

```
```r
paste("Hola", "Mundo")
```

```
[1] "Hola Mundo"
```
```

Se puede incluir un pedazo de código en cualquier parte del documento y se controlan las opciones de cada pedazo, por ejemplo, incluyendo una opción para que el código no se imprima y que solo se imprima el resultado agregando `r, echo = F`:

```
```
[1] "Hola Mundo"
```
```

Las opciones se encuentran resumidas en Inc. (2016d).

Ayuda y documentación

R tiene mucha documentación. Dado que es imposible recordar todas las funciones o cómo utilizar todo lo que ya está hecho, es necesario aprender a leerla. Desde la consola se puede accesar a la misma.

Para ayuda general,

```
help.start()
```

Para la **ayuda de una función en específico**, por ejemplo, si se quiere graficar algo y sabemos que existe la función `plot` podemos consultar fácilmente la ayuda.

```
help(plot)
# o tecleando directamente
?plot
```

El segundo ejemplo se puede extender para buscar esa función en todos los paquetes que tengo instalados en mi ambiente al escribir `??plot`.

A veces, es útil ver el **cuerpo de una función**. Esta tarea no necesariamente es trivial. Para funciones generadas por el usuario, usa

```
xx <- function(x) x^2
body(xx)
```

```
## x^2
# o simplemente imprimir el objeto en donde guardamos la función
xx

## function(x) x^2
```

También funciona para algunas funciones de paquete, por ejemplo `rename`:

```
library(plyr)
body(rename)
```

```
## {
##   names(x) <- revalue(names(x), replace, warn_missing = warn_missing)
##   duplicated_names <- names(x)[duplicated(names(x))]
##   if (warn_duplicated && (length(duplicated_names) > 0L)) {
##     duplicated_names_message <- paste0(``, duplicated_names,
##                                         ``, collapse = ", ")
##     warning("The plyr::rename operation has created duplicates for the ",
##            "following name(s): (", duplicated_names_message,
##            ") ", call. = FALSE)
##   }
##   x
## }
```

Para `plot`, en cambio, al usar la función `body` se ve:

```
body(plot)
```

```
## UseMethod("plot")
```

Esto es porque `plot` es una función genérica (S3) que tiene métodos para distintas clases de objetos. En esos casos, primero debemos usar la función `methods` para enlistar los métodos que tiene esa función.

```
methods(plot)

## [1] plot.acf*           plot.data.frame*   plot.decomposed.ts*
## [4] plot.default*       plot.dendrogram*  plot.density*
## [7] plot.ecdf*          plot.factor*      plot.formula*
## [10] plot.function*      plot.hclust*     plot.histogram*
## [13] plot.HoltWinters*  plot.isoreg*     plot.lm*
## [16] plot.medpolish*    plot.mlm*        plot.ppr*
## [19] plot.prcomp*        plot.princomp*   plot.profile.nls*
## [22] plot.raster*       plot.spec*      plot.stepfun
## [25] plot.stl*          plot.table*     plot.ts
## [28] plot.tskernel*     plot.TukeyHSD*
## see '?methods' for accessing help and source code
```

Si tiene asteriscos, significa que la función para ese método en particular no viene directamente del espacio de nombres del paquete pero, de cualquier forma, lo podemos pedir usando la función `getAnywhere` para cualquiera de los métodos que se desplegaron:

```
getAnywhere(plot.density)
```

```
## A single object matching 'plot.density' was found
## It was found in the following places
##   registered S3 method for plot from namespace stats
##   namespace:stats
##   with value
##
## function (x, main = NULL, xlab = NULL, ylab = "Density", type = "l",
##         zero.line = TRUE, ...)
## {
##   if (is.null(xlab))
##     xlab <- paste("N =", x$n, " Bandwidth =", formatC(x$bw))
##   if (is.null(main))
##     main <- deparse(x$call)
##   plot.default(x, main = main, xlab = xlab, ylab = ylab, type = type,
##               ...)
##   if (zero.line)
##     abline(h = 0, lwd = 0.1, col = "gray")
##   invisible(NULL)
## }
## <bytecode: 0x3f4a998>
## <environment: namespace:stats>
```

Nota como el método `plot.density` viene del paquete `stats`³.

³Hay otro tipo de funciones en las accesas al código fuente no se pueda con los métodos descritos. Para ello, es útil revisar la sección ".old-school object-oriented programming in R"(Adler 2010, p.131-133) o las secciones dedicadas a los objetos S3 y S4 en Hadley Wickham (2014a).

La documentación normalmente se acompaña de **ejemplos**. Para *correr* los ejemplos sin necesidad de copiar y pegar, prueba

```
example(plot)
```

Para búsquedas más comprensivas, se puede buscar de otras maneras:

```
apropos("foo") # Enlista todas las funciones que contengan la cadena "foo"
RSiteSearch("foo") # Busca por la cadena "foo" en todos
# los manuales de ayuda y listas de distribución.
```

Optimizando

Es común que muy pronto nos encontramos con limitaciones al poder de cómputo y rapidez con el que R procesa los datos. Hay operaciones intensivas como, por ejemplo, la inversión de matrices (qr) o el análisis por componentes principales (svd). Incluso una selección de variables (*back/forward selection*) usando una simple regresión lineal sobre múltiples regresores puede llevar un tiempo de cómputo de horas/días o no terminar.

Una de las maneras más rápidas de mejorar el rendimiento (*performance*) de R es instalando las librerías de álgebra lineal que puede utilizar el software para hacer las operaciones más rápido.

Para mucho (demasiado) detalle al respecto, referirse a la comparación de rendimiento en Eddelbuettel (2010) o al paquete del mismo autor Eddelbuettel (2016).

Para la parte práctica de todo esto, referirse a este blog para instalar las librerías apropiadas para BLAS y Lapack (Klamer 2014). Para una comparación bastante práctica de las diferentes versiones de esas librerías, ver aquí (Nguyen 2014).

Material adicional

Práctica y paciencia son dos elementos fundamentales para tener éxito cuando se aprende un nuevo lenguaje de programación. Un proyecto interesante para aprender R es **swirl** (Sean Kross y col. 2016), un paquete en CRAN en el que se desarrolla una gama de cursos que permiten aprender interactivamente desde la consola de R. El material actualmente se encuentra en inglés.

Para instalar swirl,

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

Luego llama a la función **swirl** para activarlo

swirl()

Lo primero que te pedirá es un nombre de usuario (para que pueda guardar el avance en los cursos y no debas regresar) y dará algunas instrucciones y comandos útiles.

```
| You can exit swirl and return to the R prompt (>) at any time by pressing the Esc key. If you are already at the prompt, type bye() to exit and save your progress.
| When you exit properly, you'll see a short message letting you know you've done so.

| When you are at the R prompt (>):
| -- Typing skip() allows you to skip the current question.
| -- Typing play() lets you experiment with R on your own; swirl will ignore what you do...
| -- UNTIL you type nxt() which will regain swirl's attention.
| -- Typing bye() causes swirl to exit. Your progress will be saved.
| -- Typing main() returns you to swirl's main menu.
| -- Typing info() displays these options again.

| Let's get started!
|
```

- **skip()** para saltarte la pregunta actual
- **play()** para poder utilizar la consola en ese momento y practicar un poco más
- **nxt()** para que se pase a la siguiente pregunta
- **bye()** para salir de swirl
- **main()** para regresar al menú principal
- **info()** para recordar las instrucciones

Una vez que te da la introducción, acepta que te instale el curso de **R Programming**

```
| To begin, you must install a course. I can install a course for you from the internet, or I can send you to a web page (https://github.com/swirldev/swirl\_courses)
| which will provide course options and directions for installing courses yourself. (If you are not connected to the internet, type 0 to exit.)

1: R Programming: The basics of programming in R
2: Regression Models: The basics of regression modeling in R
3: Don't install anything for me. I'll do it myself.

Selection: 1
```

El material de este capítulo se cubre en los módulos 1 a 3 del curso **R Programming**.

Capítulo 3

Estructuras y funciones

En este capítulo se introducen los principales objetos de R. Primero, se definen brevemente. Despu s se introducen las funciones, objetos que permiten realizar acciones sobre otros objetos.

Posteriormente, se introducen las distintas estructuras de datos en R b sico. El primer bloque de construcci n son las *clases* de datos con los que R sabe trabajar, es decir, caracteres, n meros enteros, reales, complejos y booleanos.

El segundo bloque son los *vectores*. Esta es una estructura fundamental en R y est n conformados por un conjunto de elementos de una de las clases de datos. El tercero son las *matrices*, que le agregan una dimensi n a los vectores. Las *listas* son como vectores pero pueden contener un subconjunto de elementos de cualesquiera de las clases, incluida otra lista.

Los *dataframes* son listas con la restricci n que cada uno de sus elementos es un vector del mismo tama o. Esta estructura es la m s natural para un estad stico pues refiere a la forma tabular en la que se acostumbra pensar a los datos en esa disciplina. Los *tibbles* y los *datatables* extienden los dataframes, haci ndolos m s eficientes para el procesamiento de una mayor cantidad de datos.

Finalmente, se mencionan objetos adicionales -como el infinito y el objeto que representa valores perdidos- y se describen las principales estructuras de control, proporcionando ejemplos para escribirlos en R.

Objetos



En R

- Todo lo que existe es un objeto.
- Todo lo que sucede es una llamada a una funci n.

Todo lenguaje de programación provee de una forma de accesar los datos guardados en memoria. R no permite un acceso directo a la memoria de la computadora pero ofrece varias estructuras de datos especializadas para realizar esa tarea. A estas estructuras, se les da el nombre de objetos (R Core Team 2016c, ver sección 2 “objetos”). Estos objetos son referidos a través de símbolos o variables, sin embargo, los símbolos son también objetos y pueden ser manipulados de la misma manera.

Todos los objetos tienen un *tipo*, mismo que se le puede preguntar a los objetos con la función `typeof`

```
typeof("holo")
```

```
## [1] "character"
```

Esta función puede reconocer muchos tipos, entre ellos algunos de los que veremos con mayor detalle a continuación. Comenzaremos con las funciones que, regresando al cuadro anterior, *todo lo que sucede es una llamada a una función*. Posteriormente, se revisarán las estructuras de datos más básicas en R y, por último, se verán estructuras de control básicas que permitirán mezclar el uso de objetos de manera que se operen bajo ciertas condiciones lógicas.

Funciones

Hay una regla de oro en programación en general: *DRY code*¹ (acrónimo de “Don’t repeat yourself”) (Hunt y Thomas 1999). Básicamente esto se reduce a *no te repitas*. Cuando tienes las mismas líneas de código varias veces (cuando estás copiando y pegando mucho) entonces lo que necesitas es escribir una función que realice esa tarea.

En R las funciones son los *building blocks* de básicamente todo. Como todo lo demás en R, las funciones son también objetos. Cuando llamas a un objeto en R, casi siempre estas en realidad llamando a una función.

Componentes de una función

- El `body()` o cuerpo de la función es el código dentro de la misma.
- `formals()` o el listado de argumentos formales de la función, controla cómo se puede llamar a una función.
- El ambiente `environment()` determina cómo son referidas las variables dentro de la función.
- La lista de argumentos se obtiene con `args()`

¹El concepto es mucho más general que esto pero lo reduciremos a una de sus capas más bajas primero para introducirlo y, segundo, porque el concepto general cobra sentido al avanzar en las tareas de programación a las que se enfrenta el usuario.

```

# Ejecuto la función
f <- function(x) x
# Imprimo el objeto f
f

## function(x) x
# Al usar la función body, veo solo el cuerpo
body(f)

## x
# Lista sus parámetros o argumentos
formals(f)

## $x
# Veo en qué ambiente está la función
environment(f)

## <environment: R_GlobalEnv>

```

Una vez definida una función, llamarla es muy sencillo: se le proporciona un valor para los parámetros y nos regresa el resultado esperado. Una función puede regresar cualquier objeto, por ejemplo, una función o un valor. Llamamos a la función declarada arriba:

```

f(4)

## [1] 4
# Elimino la función del espacio de trabajo
rm(f)

```

Por defecto (*default*), los argumentos de una función son flojos (*lazy*), es decir, solamente son evaluados cuando se utilizan (esto es importante pues si tienes un error en una función no te darás cuenta cuando ejecutes la misma sino cuando la mandes llamar).

El ambiente

Las variables que se definen dentro de una función existen en un ambiente distinto al ambiente global de R. Si una variable **no** está definida dentro de la función, R busca en el nivel superior por esa variable.

```

x <- 2
g <- function() {
  y <- 1
  c(x, y)
}
g()

```

```
## [1] 2 1
rm(x, g)
```

Así como fuimos capaces de anidar ciclos for, también podemos anidar funciones. Esta capacidad es muy útil pero hay que tener cuidado con los ambientes y la jerarquía en los mismos.

```
myfuncion <- function() {
  print("Hola")
}
myfuncion()
```

```
## [1] "Hola"
```

Podemos generar funciones con mayor utilidad.

```
suma <- function(x, y){
  return(x + y)
}
vector <- c(1, 2, 3, 4)
sapply(vector, suma, 2)
```

```
## [1] 3 4 5 6
```

Toda función *regresa* un valor.

```
x <- 10
f <- function() {
  y <- 25
  g <- function() {
    z <- 30
    c(x = x, y = y, z = z)
  }
  g()
}
```

```
##  x  y  z
## 10 25 30
```

```
f <- function(x) {
  x * 2
}
g <- function(x) {
  x + 2
}
f(g(2))
```

```
## [1] 8
g(f(2))
```

```
## [1] 6
```

En este caso, utilizamos una función con parámetros que *recibe* cuando es llamada. También podemos generar funciones con valores predefinidos, es decir, defaults. Éstos son utilizados cuando se llama a la función *a menos que* se especifique lo contrario (es decir, se *override them*).

```
f <- function(a = 2, b = 3) {
  return(a + b)
}
f()
```

```
## [1] 5
f(4, 5)
```

```
## [1] 9
f(b = 4)
```

```
## [1] 6
```

Return

No es necesario especificar lo que regresa la función. Las funciones por default regresan el último elemento o valor computado.

Reglas de visibilidad (scoping)

Sabemos que existe la función *c* que nos permite concatenar vectores o elementos a vectores. Sin embargo, es posible asignar un valor a una variable llamada *c* y que la función *c* siga funcionando.

```
c <- 1000
c + 1
```

```
## [1] 1001
x <- c(1:4)
x
```

```
## [1] 1 2 3 4
```

Esto es debido a que R tienen espacios de nombres (*namespaces*) separados para funciones y no-funciones. Cuando R intenta concatenar los valores del 1 al 4, busca primero en el

ambiente global y, en caso de no encontrarlo, busca en los *namespaces* de cada uno de los paquetes que tiene cargados.

El orden en el que busca se puede encontrar utilizando el comando `search()`.

`search()`

```
## [1] ".GlobalEnv"      "package:knitr"    "package:rmarkdown"
## [4] "package:stats"    "package:graphics" "package:grDevices"
## [7] "package:utils"     "package:datasets" "Autoloads"
## [10] "package:base"
```

Los paquetes recién llamados acaban en la posición número 2 y todo lo demás se recorre en el orden de la lista. Nota como el *base* (que se carga por default en toda sesión) está hasta el final.

.*GlobalEnv* es el workspace del que hablamos antes. Si hay un símbolo que corresponde a tu petición entonces tomará el valor en tu workspace para poder ejecutar tu petición. Si no encuentra nada, busca en el namespace de cada uno de los paquetes que has cargado hasta el momento en el *orden* en el que los llamaste.

Esto es **muy** importante. Hay contribuidores de paquetes en todo el mundo y es muy común que utilicen el mismo nombre para implementaciones de distintas cosas y, por lo tanto, a veces nuestros resultados no son lo que esperábamos.

El orden en el que cargamos los paquetes importa:

```
## <environment: namespace:car>
## <environment: namespace:car>
## <environment: namespace:VIF>
```

La otra opción, sin quitar el paquete del ambiente, es especificar de que paquete tomarlo. En otras palabras, le pedimos explícitamente a R que busque la función en el espacio de nombres de *un* paquete en específico y que no use su búsqueda normal.

`environment(VIF::vif)`

```
## <environment: namespace:VIF>
environment(car::vif)

## <environment: namespace:car>
```

Estructuras de datos

R tiene diferentes tipos y estructuras de datos que permiten al usuario aprovechar el lenguaje. La manipulación de estos objetos es algo que se hace diario y entender cómo operarlos o cómo convertir de una a otra es muy útil.

Clases atómicas (atomic classes)

R tiene 6 clases atómicas² (R Core Team 2016c).

- **character** (caracter)
- **numeric** (números reales o decimales, a esta clase también se le llama **double**)
- **integer** (números enteros)
- **logical** (booleanos, i.e. falso-verdadero)
- **complex** (números complejos)
- **raw** (contiene bytes)

| Type | Tipo | Ejemplo |
|------------------|----------|----------------------------|
| character | Caracter | "hola", "x" |
| numeric | Numérico | 67, 45.5 |
| integer | Integer | 2L, 67L |
| logical | Lógico | TRUE, FALSE, T, F |
| complex | Complejo | 1 + 4i |
| raw | Crudo | 01 - imprime hexadecimales |

Cuadro 3.1: Clases atómicas.

Algunos comandos importantes para las clases atómicas son su tipo **typeof()**, su tamaño **length()** y sus atributos **attributes()**, es decir, sus metadatos.

```
#####
# Ejemplo 1

x <- "una cadena"
typeof(x)

## [1] "character"

length(x) # tamaño: ¿cuántas cadenas son?

## [1] 1

nchar(x) # Número de caracteres

## [1] 10

attributes(x) # Le pusimos metadatos?

## NULL

#####
# Ejemplo 2

y <- 1:10
typeof(y)
```

²Los tipos básicos son referidos como atómicos cuando es necesario excluir listas.

```

## [1] "integer"
length(y)

## [1] 10
attributes(y)

## NULL
##### Ejemplo 3

z <- c(1L, 2L, 3L) # Nota como para denotar enteros se incluye una L al final
typeof(z)

## [1] "integer"
length(z)

## [1] 3

```

Vectores

Los vectores son la estructura de datos más básica de R (Hadley Wickham 2014a). Hay dos tipos de vectores: vectores atómicos y listas.

Típicamente -en libros, blogs, manuales, cuando se mencionan vectores se refieren a los atómicos y no a las listas.

Vectores atómicos

Los vectores pueden ser pensados como celdas contiguas que contienen datos (R Core Team 2016c), es decir, elementos de alguna de las clases atómicas (`character`, `logical`, `integer`, `numeric`). Se puede crear un vector vacío con el comando `vector()` así como especificar su tamaño y su clase.

```

v <- vector()
v

## logical(0)
## Especifico clase y longitud
vector("character", length = 10)

## [1] "" "" "" "" "" "" "" "" "" ""
## Lo mismo pero usando un wrapper
character(10)

```

```
## [1] "" "" "" "" ""
## Número de tamaño 5
numeric(5)

## [1] 0 0 0 0 0
## Lógico tamaño 5
logical(5)

## [1] FALSE FALSE FALSE FALSE FALSE
```

Tipos de vectores

Realiza los siguientes ejemplos en la consola de R.

```
x <- rep(1, 5)
x
typeof(x)

xi <- c(1L, 3L, 56L, 4L)
xi
typeof(xi)

y <- c(T, F, T, F, F, T)

z <- c("a", "aba", "andrea", "b", "bueno")
```

Dijimos que la función `typeof` permitía preguntarle a un objeto qué tipo de dato es. La función `class` permite hacer una pregunta similar. La diferencia radica en el punto de vista: el primero da el tipo del objeto como un objeto en R mientras que, el segundo identifica el tipo del objeto desde el punto de vista de la programación orientada a objetos en R.

```
class(z)

## [1] "integer"
```

Otra función útil es `str` pues permite desplegar en forma compacta la estructura interna de un objeto en R:

```
str(z)

## int [1:3] 1 2 3
```

Operaciones con vectores

Aritmética: por default, se realizan componente a componente.

```

a <- c(1:5)
b <- a + 10
b

## [1] 11 12 13 14 15

c <- sqrt(b) # square root = raíz
c

## [1] 3.316625 3.464102 3.605551 3.741657 3.872983

a + c

## [1] 4.316625 5.464102 6.605551 7.741657 8.872983

10 * (a + c)

## [1] 43.16625 54.64102 66.05551 77.41657 88.72983

a^2

## [1] 1 4 9 16 25

a * c

## [1] 3.316625 6.928203 10.816654 14.966630 19.364917

```

Agregar elementos aun vector ya creado

```

a <- c(a, 7)
a

```

```
## [1] 1 2 3 4 5 7
```

Para construir datos rápido, podemos usar comandos como `rep`, `seq` o distintas distribuciones, e.g., la normal `rnorm`, uniformes `runif` o cualquiera en esta lista.

Prueba lo siguiente:

```

# Dame un vector donde el minimo sea 0, maximo 1 en intervalos de 0.25
seq(0, 1, 0.25)
# Vector con 10 unos
rep(1, 10)
# 5 realizaciones de una normal(0,1)
rnorm(5)
# De una normal(10, 5)
rnorm(5, mean = 10, sd = sqrt(5))
# De una uniforme(0,1)
runif(5)
# De una uniforme(5, 15)
runif(5, min = 5, max = 15)

```

Atributos de un vector

Cada objeto tiene atributos. Hay atributos específicos para vectores que, sin importar su clase, tienen en común. Ya revisamos algunos: tamaño (`length`), clase (`class`). También son importantes atributos como los nombres

```
calificaciones <- c(6, 5, 8, 9, 10)
names(calificaciones) <- c("Maria", "Jorge", "Miguel", "Raúl", "Carla")
attributes(calificaciones)

## $names
## [1] "Maria" "Jorge" "Miguel" "Raúl"   "Carla"
# O llamamos directo a los nombres
names(calificaciones)

## [1] "Maria" "Jorge" "Miguel" "Raúl"   "Carla"
```

Coerción

Los vectores solo permiten tener objetos del mismo tipo. Hay coerción explícita (*explicit coercion*, también llamada *cast*) utilizando `as.<nombre_clase>`.

```
as.numeric()
as.character()
as.integer()
as.logical()
```

Utilizando coerción explícita garantizamos siempre tener el resultado en cuanto a la clase del objeto.

```
c(c("a", "b", "c"), as.character(c(1, 2, 3)))
```

```
## [1] "a" "b" "c" "1" "2" "3"
```

Realizar coerción explícita implica trabajar extra y, a veces, no se puede realizar de manera directa: los datos pueden venir *sucios* con varios tipos de datos mezclados en una misma *variable*.

R mezcla distintos tipos de datos y realiza una *coerción implícita* utilizando reglas razonables. En otras palabras, R realiza una coerción explícita *por default* entre los objetos y “decide” cuál es la clase del vector.

```
# Número + caracter = caracter
c(1.7, "a")
```

```
## [1] "1.7" "a"
```

```
# Lógico + número = número
c(TRUE, 2)
```

```
## [1] 1 2
# Número + carácter = carácter
c("a", TRUE)
```

```
## [1] "a"    "TRUE"
```

En ese proceso, puede haber pérdidas de información, por ejemplo, al mezclar valores lógicos con numéricos. Se confunden valores *verdaderos* con un *uno*. Hay que tener cuidado particularmente cuando se limpian los datos:

```
c(c(T, T, T), c(1, 2, 3))
```

```
## [1] 1 1 1 1 2 3
```

Hay conversiones que no tienen sentido y generan pérdida de información total:

```
x <- c("a", "b", "c")
as.numeric(x)
```

```
## [1] NA NA NA
```

```
as.logical(x)
```

```
## [1] NA NA NA
```

Normalmente, se obtiene un mensaje de advertencia (*warning*) cuando alguna coerción puede derivar en pérdida de información (Hadley Wickham 2014a).

La última consideración importante es que para R un objeto no es igual, aunque no se pierda información, si su tipo no es el mismo

```
x <- 0:5
identical(x, as.numeric(x))
```

```
## [1] FALSE
```

En este ejemplo, cuando declaramos *x* no especificamos su clase y R decidió que era entero. Al coercionar al objeto para que fuese numérico, R no considera a los dos objetos iguales.

En general, la coersión de R es muy útil pues permite incluso comparar objetos de distintas clases si el resultado tiene sentido

```
1 < "2"
```

```
## [1] TRUE
```

Lo importante es recordar que es importante revisar las **advertencias** que R arroja a la consola y verificar que el resultado obtenido es el deseado o que la pérdida de

información no se puede evitar.

Extraer partes del vector

R tiene constructos que permite acceder a elementos individuales o subconjuntos de un vector a través de operaciones de indexación (*indexing*) (R Core Team 2016c, sección “Indexing”).

Para los vectores, es posible acceder al i-ésimo elemento usando `x[i]`.

```
x <- c(10, 20, 30, 40, 50)
names(x) <- c("a", "b", "c", "d", "e")
# Accedemos al 4to elemento
x[4]
```

```
## d
## 40
```

Además de la indexación con un entero, se puede

```
# x[i] - caso anterior
# x[[i]]
x[[4]]
```

```
## [1] 40
# x["a"] - por nombre (cuando existen)
x["a"]
```

```
## a
## 10
# Se puede extraer un subconjunto
x[1:3]
```

```
## a b c
## 10 20 30
```

[] vs. [[]]

Estas dos formas de acceder a los elementos de un vector (utilizados también en otras estructuras de datos) suelen causar confusión.

En vectores, [[casi no se utiliza, aunque son ligeramente diferentes. Como vimos en el ejemplo, [[quita los nombres o atributos y permite extraer únicamente un elemento a la vez.

Matrices

Las matrices son un tipo especial de vectores. Son un vector atómico con una dimensión adicional pues tienen filas y columnas.

```
m <- matrix(c(1, 2, 3, 4), nrow = 2, ncol = 2)
m

##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

En términos de sus atributos por *default*, la diferencia entre los vectores y las matrices es:

```
x <- c(1, 2, 3, 4)
attributes(x)
```

```
## NULL

attributes(m)

## $dim
## [1] 2 2
```

Como puedes notar, las matrices se forman *por default* usando los elementos del vector para llenar columna por columna de izquierda a derecha. Podemos simplemente “agregarle” una dimensión a un vector para construir una matriz.

```
m <- 1:10
m

## [1]  1  2  3  4  5  6  7  8  9 10
dim(m) <- c(2, 5)
m

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

También podemos pegar o concatenar vectores de la misma longitud como si fueran columnas de una matriz usando `cbind` o como si fueran filas `rbind` (`r` = row, `c` = column).

```
x <- runif(4)
y <- rnorm(4)
cbind(x, y)

##           x         y
## [1,] 0.3683017  0.7765167
## [2,] 0.9638695  0.4221273
## [3,] 0.4937018 -0.8593989
```

```
## [4,] 0.8532536 0.6414013
rbind(x, y)

## [,1]      [,2]      [,3]      [,4]
## x 0.3683017 0.9638695 0.4937018 0.8532536
## y 0.7765167 0.4221273 -0.8593989 0.6414013
```

Le agregamos atributos para accesar más fácilmente a los objetos.

```
m <- matrix(c(x, y), nrow = 4, ncol = 2, byrow = T,
            dimnames = list(paste0("row", 1:4),
                            paste0("col", 1:2)))
m

##          col1     col2
## row1  0.3683017 0.9638695
## row2  0.4937018 0.8532536
## row3  0.7765167 0.4221273
## row4 -0.8593989 0.6414013

dimnames(m)

## [[1]]
## [1] "row1" "row2" "row3" "row4"
##
## [[2]]
## [1] "col1" "col2"
```

Acceder a elementos de una matriz puede hacerse de muchas formas

```
# m[i] - quinto elemento, contando desde entrada 1,1 por columnas
m[5]

## [1] 0.9638695

# m[[i]] - quinto elemento, quitando atributos
m[[5]]

## [1] 0.9638695

# m[i, j] - mismo elemento que m[5] pero usando notacion fila, columna
m[1, 2]

## [1] 0.9638695

# m[[i, j]] - mismo elemento, quitando atributos
m[[1, 2]]

## [1] 0.9638695
```

```
# Puedo llamar por su nombre
m["row1", "col2"]

## [1] 0.9638695

# Misma forma, quitando atributos
m[["row1", "col2"]]

## [1] 0.9638695

# m[i, ] - toda la fila i-ésima
m[1, ]

##      col1      col2
## 0.3683017 0.9638695

# m[, j] - toda la columna j-ésima
m[, 2]

##      row1      row2      row3      row4
## 0.9638695 0.8532536 0.4221273 0.6414013

# Índices o nombres son equivalentes
m[1, 1] == m["row1", "col1"]

## [1] TRUE
```

[] vs. [[]]

En matrices, [[casi no se utiliza. Como vimos en el ejemplo, [[quita los nombres o atributos y permite extraer únicamente un elemento a la vez.

Listas

Tiene características muy similares a un vector pero permite que cada elemento sea de un tipo distinto. Mas aún, es posible incluir una lista como un elemento de otra lista y por eso también se les conoce como vectores recursivos (*recursive vectors*) \parencite{wickham2014advanced} sección “lists”.

Para crear una lista vacía utilizas `list()` y para coercionar un objeto a una lista usa `as.list()`.

```
x <- list(3L, 3.56, 1 + 4i, TRUE, "hola", list("genial", 1))

length(x)

## [1] 6
```

```

class(x)

## [1] "list"
class(x[[1]])

## [1] "list"
class(x[[1]]))

## [1] "integer"
y <- as.list(1:10)
length(y)

## [1] 10

```

Nota como muchas propiedades que tenían los vectores atómicos los tienen también las listas. Las listas también pueden tener nombres

```

# Lista vacía
lista <- list()

# Concatenamos un vector
lista[["numeros"]] <- c(1, 34, 45.5, 34)
# Concatenamos un objeto de datos
lista[["datos"]] <- head(iris)
# Concatenamos un número
lista <- c(lista, 3) # ¡No le tuvimos que poner nombre!

lista

## $numeros
## [1] 1.0 34.0 45.5 34.0
##
## $datos
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1        3.5         1.4        0.2  setosa
## 2          4.9        3.0         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          5.0        3.6         1.4        0.2  setosa
## 6          5.4        3.9         1.7        0.4  setosa
##
## [[3]]
## [1] 3

```

R tiene muchos datos de ejemplo que son utilizados en muchos paquetes, blogs y libros. Utiliza **help(iris)** para saber más del dataset usado arriba.

Por su propiedad recursiva, se navega diferente. Repasamos las principales maneras de extraer los elementos de la lista x declarada anteriormente:

```
# Recordamos a x
x

## [[1]]
## [1] 3
##
## [[2]]
## [1] 3.56
##
## [[3]]
## [1] 1+4i
##
## [[4]]
## [1] TRUE
##
## [[5]]
## [1] "hola"
##
## [[6]]
## [[6]][[1]]
## [1] "genial"
##
## [[6]][[2]]
## [1] 1

# x[i] - el i-ésimo elemento de la lista
x[3]

## [[1]]
## [1] 1+4i
## Nota como la clase del objeto sigue siendo lista
class(x[3])

## [1] "list"

# x[[i]] - el i-ésimo elemento de la lista
x[[3]]

## [1] 1+4i
## La clase ahora es la del objeto dentro del "espacio" 3 en la lista original
class(x[[3]])

## [1] "complex"
```

```

# Nombramos la lista
names(x) <- c("entero", "numerico", "complejo"
             , "booleano", "caracter", "lista")

# Ganamos formas de accesar los objetos
# x$a - llamamos al elemento con nombre "a"
x$entero

## [1] 3

class(x$entero) # Es equivalente a []

## [1] "integer"

# x$"a"
x$"complejo"

## [1] 1+4i

# x[["lista"]][i] - i-ésimo elemento de la lista dentro de la lista
x[["lista"]][1]

## [[1]]
## [1] "genial"

# x[[j]][i] - Mismas reglas en la lista anidada
x[[6]][1]

## [[1]]
## [1] "genial"

# x[[j]][[i]] - i-ésimo elemento en la lista del j-ésimo elemento de x
x[[6]][[1]]

## [1] "genial"

```

[] vs. [[]]

En listas, [[es fundamental para accesar correctamente los objetos y poder navegar la lista.

Como en vectores y matrices, [[quita los nombres o atributos y permite extraer únicamente un elemento a la vez. En listas, además, devuelve el objeto dentro del i-ésimo elemento. Por el contrario, [devuelve una lista.

Puedo poner listas dentro de listas, dentro de listas... Se navega en orden como en el ejemplo.

Las longitudes de los objetos en la lista se pueden pensar *por niveles*, por su propiedad

recursiva.

```
# El tamaño es del "primer nivel".
length(x)

## [1] 6

# Hay 6 elementos en x, todos de diferentes tipos
names(x)

## [1] "entero"    "numerico"   "complejo"   "booleano"   "caracter"   "lista"
# Para obtener la longitud dentro del i-ésimo elemento de la lista, debo
length(x[[6]]) # La lista anidada tiene 2 elementos

## [1] 2

# que no es lo mismo que
length(x[6]) # Donde hay un solo elemento: una lista

## [1] 1
```

Factores (factor)

Los factores son otro tipo de vectores pero que ayuda a representar datos del tipo categórico u ordinal, es decir, cuando los posibles valores de la variable tipo carácter es limitado. Por ejemplo, son útiles cuando tenemos una variable como “sexo” donde, al menos por ahora, legalmente solo puede tomar los valores *hombre* o *mujer*. Si, en cambio, se tiene un vector de nombres es conveniente dejarlo como carácter.

Un factor se guarda como un *enteros* pero con *etiquetas* encima tal que cada entero corresponde a una etiqueta (*label*).

```
y <- c("no", "si", "si", "no")
class(y)

## [1] "character"

# Debemos pedirle explícitamente que lo guarde como factor
x <- factor(c("no", "si", "si", "no"))
x

## [1] no si si no
## Levels: no si
```

Al imprimir el objeto, se observa como los niveles fueron asignados. Éstos corresponden al número de valores únicos en el vector de carácter y se asignan en orden alfabético los valores.

Los factores se despliegan *como si fueran* vectores tipo carácter y algunas operaciones son análogas:

```
table(x)
```

```
## x
## no si
## 2 2
```

La ganancia es que son más rápidas. Aunque a veces los factores se comportan como vectores tipo carácter pero *debemos* recordar que por debajo son enteros y tenemos que ser cuidadosos si los tratamos como caracteres.

Supongamos por ejemplo que tenemos un factor con valores 5, 6 o 7. Lo tenemos guardado como factor.

```
ej <- factor(c("7", "6", "5", "7", "5", "7", "6", "5", "5", "5", "6", "5"))
ej
```

```
## [1] 7 6 5 7 5 7 6 5 5 5 6 5
## Levels: 5 6 7
```

Dado que los valores son números, conceptualmente tiene sentido operarlos como tal:

```
as.integer(ej)
```

```
## [1] 3 2 1 3 1 3 2 1 1 2 1
```

Obtuvimos los enteros a los que las etiquetas originales habían sido asignados. Para recuperar los valores originales, debemos hacer

```
as.integer(as.character(ej))
```

```
## [1] 7 6 5 7 5 7 6 5 5 6 5
```

Algunos métodos que están hechos para caracteres coercionan un factor a carácter mientras que otros arrojan un error. Si usas métodos de caracteres, lo mejor es “castear” (coerción explícita) a carácter tu factor utilizando `as.character(mifactor)`. De esta manera se pierden algunas cosas pero te aseguras que las cosas funcionen como deben.

```
summary(x)
```

```
## no si
## 2 2
```

```
summary(as.character(x))
```

```
##      Length     Class      Mode
##          4 character character
```

Summary

R funciona mejor gracias a sus convenciones, es decir, porque los contribuyentes se ponen de acuerdo en seguir ciertas reglas de manera que sea más fácil utilizar los

paquetes de otros (con sus objetos y funciones).

La función `summary` es la función genérica que produce resúmenes para objetos de muchas clases. La función invoca métodos que dependen de la clase del argumento enviado (en estos ejemplos, el resumen para un factor y para un carácter respectivamente).

Los factores pueden incluir únicamente los niveles con los que fueron definidos. Por esa razón, la unión de dos factores declarados en forma independiente puede dar resultados no deseados.

```
y <- factor(c("si", "no", "tal vez"))
```

```
c(x, y)
```

```
## [1] 1 2 2 1 2 1 3
```

```
class(c(x, y))
```

```
## [1] "integer"
```

No es posible entonces recuperar el valor de las etiquetas. R hizo las operaciones posibles pero hubo pérdida de información. Para concatenar dos factores correctamente, es necesario:

```
factor(c(as.character(x), as.character(y)))
```

```
## [1] no      si      si      no      si      no      tal vez
```

```
## Levels: no si tal vez
```

En general, se recomienda incluir el valor de un nivel posible, independientemente de si se tiene o no esa respuesta. Sin embargo, el problema al concatenar persiste.

```
x <- factor(c("no", "si", "si", "no"), levels = c("no", "si", "tal vez"))
c(x, "tal vez")
```

```
## [1] "1"        "2"        "2"        "1"        "tal vez"
```

Para datos ordinales como las respuestas en una pregunta de encuesta con escala Likert³ los factores son también objetos útiles.

Veamos un ejemplo en donde tenemos 500 respuestas a la pregunta “este tutorial es muy útil”:

```
set.seed(2887)
respuestas <- sample(x = c(1:5), size = 500, replace = T
                      , prob = c(0.1, 0.15, 0.2, 0.4, 0.15))
y <- factor(
  x = respuestas,
  levels = c("1", "2", "3", "4", "5"),
  labels = c("totalmente en desacuerdo", "en desacuerdo"
```

³Se denomina así por su autor quien en (Likert 1932) describe su uso.

```

        , "ni de acuerdo ni en desacuerdo"
        , "de acuerdo", "totalmente de acuerdo"),
ordered = T)
table(y)

```

```

## y
##      totalmente en desacuerdo          en desacuerdo
##                           51                      74
## ni de acuerdo ni en desacuerdo          de acuerdo
##                           108                     194
##      totalmente de acuerdo
##                           73

```

Nota como la tabla está ordenada de izquierda a derecha con la respuesta más en desacuerdo a la más de acuerdo pues introducimos la opción `ordered = T` en la definición del factor debido a que las respuestas están en una escala ordinal.

Otras funciones útiles

En el ejemplo anterior, introducimos las funciones:

- `set.seed`: sirve para fijar la semilla con la que se generan números aleatorios. Esto es importante pues al fijarla se puede reproducir exactamente el mismo vector de respuestas cuantas veces sea necesario.
- `sample`: permite extraer muestras de un vector `x` especificando el tamaño de la muestra, si es muestreo con reemplazo y permite establecer pesos para el muestreo.

```
table(y)
```

```

## y
##      totalmente en desacuerdo          en desacuerdo
##                           51                      74
## ni de acuerdo ni en desacuerdo          de acuerdo
##                           108                     194
##      totalmente de acuerdo
##                           73

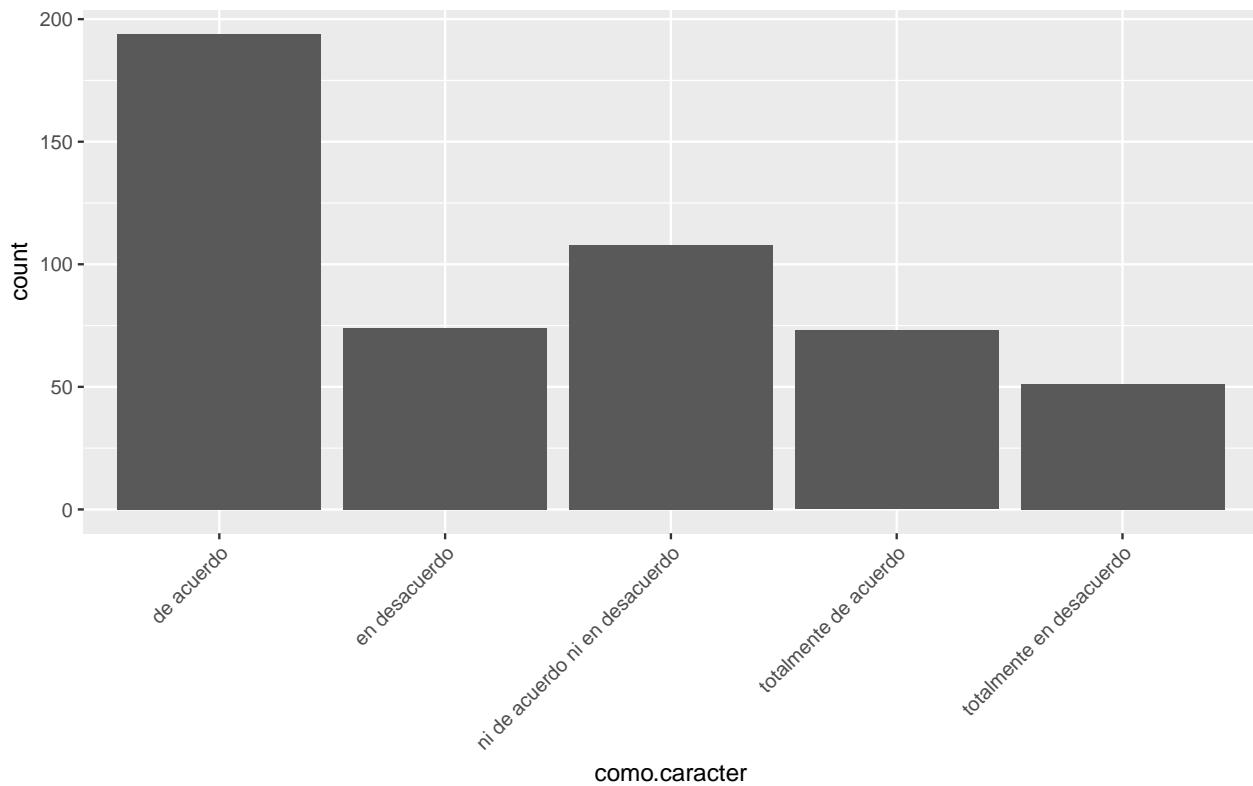
```

Por último, al utilizar factores (y más aún, declarar un orden cuando es conceptualmente pertinente) es más fácil visualizar correctamente los datos con menor desgaste. Si graficamos las respuestas como carácter recibimos:

```

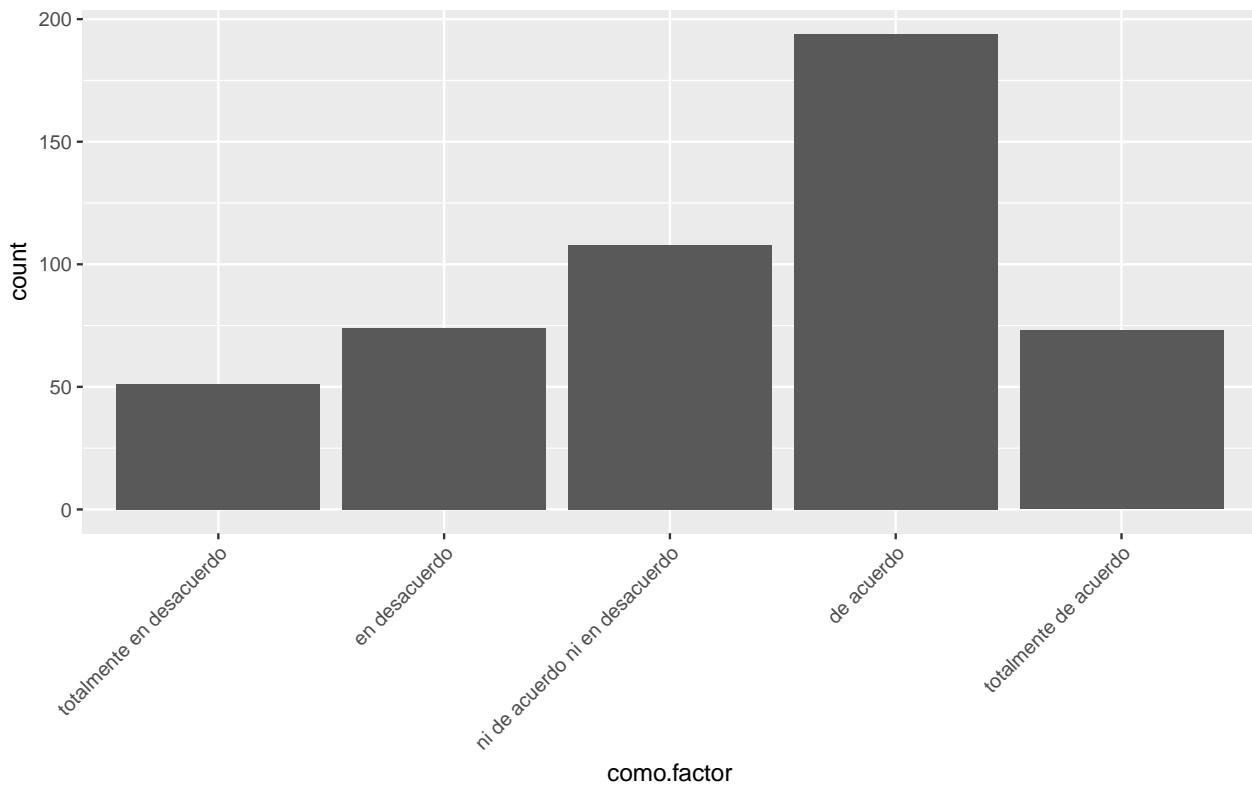
library(ggplot2)
df <- data.frame(como.caracter = as.character(y), como.factor = y)
ggplot(df, aes(x = como.caracter)) + geom_bar() +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))

```



Si utilizamos el factor ordenado obtenemos:

```
ggplot(df, aes(x = como.factor)) + geom_bar() +  
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```



Nota

En R muchas cosas son más fáciles si se utiliza la estructura de datos apropiada y se establecen correctamente todos los metadatos necesarios al objeto para que los defaults de R faciliten el trabajo.

Al utilizar factores cuando es pertinente se gana, al menos, lo siguiente:

- Se almacenan los datos usando menos memoria. Esto es importante pues R trabaja en ram.
- El rendimiento es mejor que usando caracteres.
- Por *default* R utiliza los métodos apropiados para el tipo de variable. Por ejemplo, si se introduce un factor como variable dependiente en un modelo de regresión, automáticamente aplica un logit y no un ols.
- Se pueden especificar elementos, como el orden de los niveles, que son útiles para el análisis y presentación de resultados.

A pesar de sus ventajas, si son incorrectamente utilizados las estructuras de datos pueden resultar en pérdidas de información o comportamientos indeseados.

Data frames

Los dataframes son una de las estructuras de datos más importantes para guardar datos en R (Hadley Wickham 2014a, sección “Data frames”). En python, existe una estructura similar en la librería **pandas** creado para facilitar el análisis de datos en este lenguaje sin la necesidad de cambiar a un lenguaje de dominio específico como R (McKinney 2010, sección

“What problem does pandas solve?”).

Este objeto es tan importante porque muchos de los modelos estadísticos que se utilizan necesitan una estructura de datos tabular.

Los dataframes tienen atributos adicionales a los que tienen los vectores:

- `rownames()`
- `colnames()`
- `names()`
- `head()` te enseña las primeras 6 líneas.
- `tail()` te enseña las últimas 6 líneas.
- `nrow()` te da el número de filas
- `ncol()` te da el número de columnas
- `str()` te dice el tipo de cada columna y te muestra ejemplos

Podemos ver a los dataframes como un tipo de lista con algunas restricciones (R Core Team 2016c, sección “Data frames”):

- Los componentes deben ser vectores, factores, matrices numéricas, listas u otros dataframes.
- Las matrices, listas y otros data frames proveen de tantas columnas, elementos o variables como las originales, respectivamente.
- Los vectores numéricos, lógicos y factores se incluyen en el dataframe sin transformaciones adicionales. Los vectores tipo carácter se coercionan a factores.
- Todos los elementos (las columnas) deben tener la misma longitud o tamaño.

Los dataframes se pueden crear utilizando comandos como `read.table()` (que tiene como caso particular `read.csv()`). Se verá con detalle el uso de estas funciones en la sección 5.

Para convertir un dataframe a una matriz se utiliza `data.matrix()`. La coerción es forzada y no necesariamente da lo que uno espera.

Se pueden crear data.frames con la función `data.frame()`.

```
df <- data.frame(
  x = rnorm(10),
  y = runif(10),
  n = LETTERS[1:10],
  stringsAsFactors = F # F = FALSE, T = TRUE
)

head(df)

##           x         y n
## 1  0.89814648 0.52584158 A
## 2 -0.37532709 0.68789829 B
## 3 -2.17112789 0.60079780 C
## 4 -0.63011959 0.86922537 D
```

```

## 5 0.03778982 0.44119997 E
## 6 0.35565256 0.02638035 F

dim(df)

## [1] 10 3

str(df)

## 'data.frame': 10 obs. of 3 variables:
## $ x: num 0.8981 -0.3753 -2.1711 -0.6301 0.0378 ...
## $ y: num 0.526 0.688 0.601 0.869 0.441 ...
## $ n: chr "A" "B" "C" "D" ...

```

¿Por qué usar la opción `stringsAsFactors = F`?

Por *default* las columnas tipo carácter en un `dataframe` son convertidas a factor. Esto es útil cuando se tienen los datos limpios y se va a proceder a modelar sin realizar mayores transformaciones o limpiezas a los datos. Sin embargo, cuando se realizarán manipulaciones a los mismos, es recomendable cambiar la opción por *default* para leer columnas como carácter, de esta forma se evitan los problemas mencionados en la sección de factores.

Es posible concatenar columnas o filas:

```

df <- cbind(df, data.frame(z = rexp(10)))
df <- rbind(df, c(rnorm(1), runif(1), "K", rexp(1)))
dim(df)

```

```
## [1] 11 4
```

Repasamos las principales maneras de extraer los elementos de un `dataframe` utilizando el objeto `df`:

```

df <- data.frame(
  x = rnorm(4),
  y = runif(4),
  n = LETTERS[1:4],
  stringsAsFactors = F # F = FALSE, T = TRUE
)

# df[i]
df[1] # La primera columna

##          x
## 1 0.38112864
## 2 0.51810269
## 3 -0.84698783
## 4 0.03558633

```

```

class(df[1]) # Regresa un dataframe

## [1] "data.frame"

# df[[i]]
df[[1]] # La primera columna

## [1] 0.38112864 0.51810269 -0.84698783 0.03558633

class(df[[1]]) # Regresa un vector

## [1] "numeric"

# df[i, j]
df[1,2] # elemento en la primera fila, segunda columna

## [1] 0.5346584

# df[[i, j]]
df[[1, 2]] # mismo resultado

## [1] 0.5346584

# df$columna
df$x # La columna llamada x

## [1] 0.38112864 0.51810269 -0.84698783 0.03558633

# df$"columna"
df$"x"

## [1] 0.38112864 0.51810269 -0.84698783 0.03558633

df[["n"]][1] # Podemos navegar igual que en una lista

```

[1] "A"

[] vs. [[]]

Debido a que los *dataframes* son un caso particular de las listas, [[es también fundamental para accesar correctamente los objetos.

[[quita los nombres o atributos, permite extraer únicamente un elemento a la vez y devuelve el objeto dentro del i-ésimo elemento.

[devuelve un data frame con la columna(s) que sea nombradas o los índices que sean utilizados.

Cuando se declara un `dataframe` automáticamente se verifican que los nombres sean sintácticamente válidos con la función `make.names`

```
data.frame("2000" = c(100:104)
          , "una-variable" = c(200:204)
          , ".2000" = c(300:304)
)
```

```
##   X2000 una.variable X.2000
## 1    100           200    300
## 2    101           201    301
## 3    102           202    302
## 4    103           203    303
## 5    104           204    304
```

Nombres sintácticamente válidos

Los `dataframes` pueden tener únicamente nombres sintácticamente válidos (Hornik 2016, sección *What are valid names?*):

- Está compuesto por letras, números, puntos o guiones bajos.
- No deben empezar con números. Tampoco pueden empezar con un punto seguido de un número.
- No se permiten palabras reservadas (if, else, repeat, next, TRUE, FALSE, entre otras).

Estructuras de datos fuera de R-básico

Fuera del R Core Team (2016a), se han desarrollado otras estructuras de datos particularmente útiles para hacer análisis de datos. Hay paquetes compatibles con este tipo de estructuras que facilitan el trabajo. A continuación presentaremos `data.tables` y `tibbles` que sustituyen el `data.frame`. Ambos se coercionan a *dataframe* de manera automática cuando se llama a una función que contiene únicamente métodos para *dataframe*.

Data tables

`data.table` provee una versión de alto rendimiento para los *dataframes* del R Core Team (2016a). Está implementado en el paquete del mismo nombre (Dowle y col. 2015).

Un `data.table` se crea en forma análoga a un `data.frame`. En la sección anterior, creamos un objeto llamado `df`

```
df <- data.frame(
  x = rnorm(4),
  y = runif(4),
  n = LETTERS[1:4],
```

```
  stringsAsFactors = F
)
```

Usamos la función `data.table()`:

```
library(data.table)
dt <- data.table(
  x = rnorm(4),
  y = runif(4),
  n = LETTERS[1:4]
)
str(dt)

## Classes 'data.table' and 'data.frame': 4 obs. of 3 variables:
## $ x: num 1.777 1.024 0.952 -0.492
## $ y: num 0.0295 0.9999 0.4618 0.6732
## $ n: chr "A" "B" "C" "D"
## - attr(*, ".internal.selfref")=<externalptr>

class(dt)

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

Por *default* los vectores tipo carácter son leidos *as-is*, es decir, sin coercionar a factor. El objeto con clase `data.table` retiene todos los atributos del `data.frame`. Podemos usar las funciones: `head`, `names`,

Para crear `data.tables` se puede coercionar cualquier `data.frame` con

```
as.data.table(diamonds)
```

| | carat | cut | color | clarity | depth | table | price | x | y | z |
|-----------|-------|-----------|-------|---------|-------|-------|-------|------|------|------|
| ## 1: | 0.23 | Ideal | E | SI2 | 61.5 | 55 | 326 | 3.95 | 3.98 | 2.43 |
| ## 2: | 0.21 | Premium | E | SI1 | 59.8 | 61 | 326 | 3.89 | 3.84 | 2.31 |
| ## 3: | 0.23 | Good | E | VS1 | 56.9 | 65 | 327 | 4.05 | 4.07 | 2.31 |
| ## 4: | 0.29 | Premium | I | VS2 | 62.4 | 58 | 334 | 4.20 | 4.23 | 2.63 |
| ## 5: | 0.31 | Good | J | SI2 | 63.3 | 58 | 335 | 4.34 | 4.35 | 2.75 |
| ## --- | | | | | | | | | | |
| ## 53936: | 0.72 | Ideal | D | SI1 | 60.8 | 57 | 2757 | 5.75 | 5.76 | 3.50 |
| ## 53937: | 0.72 | Good | D | SI1 | 63.1 | 55 | 2757 | 5.69 | 5.75 | 3.61 |
| ## 53938: | 0.70 | Very Good | D | SI1 | 62.8 | 60 | 2757 | 5.66 | 5.68 | 3.56 |
| ## 53939: | 0.86 | Premium | H | SI2 | 61.0 | 58 | 2757 | 6.15 | 6.12 | 3.74 |
| ## 53940: | 0.75 | Ideal | D | SI2 | 62.2 | 55 | 2757 | 5.83 | 5.87 | 3.64 |

Nota como los `datatables` tienen por *default* una impresión diferente que un `dataframe` pues imprimen los primeros y los últimos 6 renglones.

Los `data.tables` incorporan nuevas maneras de extraer objetos, agruparlos y juntarlos con otras tablas. Para más detalles, se pueden revisar las viñetas del paquete:

| Item | Title |
|-------------------------------|--|
| datatable-faq | Frequently asked questions (source, pdf) |
| datatable-intro | Quick introduction (source, pdf) |
| datatable-intro-vignette | Vignette Title (source, html) |
| datatable-keys-fast-subset | Vignette Title (source, html) |
| datatable-reference-semantics | Vignette Title (source, html) |
| datatable-reshape | Vignette Title (source, html) |

Para ver una viñeta en específico se puede usar `vignette("datatable-faq", package = "data.table")`.

Viñetas

Todos los paquetes tienen viñetas en donde se documentan en forma más detallada sus funciones, clases y métodos.

Tibbles

Los **tibbles** son escencialmente **dataframes** pero con algunas modificaciones a los *defaults* para facilitar el trabajo (Hadley Wickham y Garrett Grolemund 2016, ver sección “tibbles”).

Un **tibble** se crea en forma análoga a un **data.frame**. En la sección `??`, creamos un objeto llamado `df`

```
df <- data.frame(
  x = rnorm(4),
  y = runif(4),
  n = LETTERS[1:4],
)
```

Usamos la función `tibble()` del paquete Hadley Wickham, Francois y Müller (2016):

```
library(tibble)
tb <- tibble(
  x = rnorm(4),
  y = runif(4),
  n = LETTERS[1:4]
)
str(tb)

## Classes 'tbl_df', 'tbl' and 'data.frame':    4 obs. of  3 variables:
##   $ x: num  0.208 1.155 0.731 -0.214
##   $ y: num  0.146 0.895 0.418 0.809
##   $ n: chr  "A" "B" "C" "D"
```

```
class(tb)

## [1] "tbl_df"     "tbl"        "data.frame"

Para crear tibbles se puede coercionar cualquier data.frame con

as_tibble(diamonds)

## # A tibble: 53,940 × 10
##   carat      cut color clarity depth table price     x     y     z
##   <dbl>     <ord> <ord>    <ord> <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1 0.23     Ideal     E     SI2   61.5    55   326  3.95  3.98  2.43
## 2 0.21     Premium   E     SI1   59.8    61   326  3.89  3.84  2.31
## 3 0.23     Good      E     VS1   56.9    65   327  4.05  4.07  2.31
## 4 0.29     Premium   I     VS2   62.4    58   334  4.20  4.23  2.63
## 5 0.31     Good      J     SI2   63.3    58   335  4.34  4.35  2.75
## 6 0.24 Very Good   J     VVS2   62.8    57   336  3.94  3.96  2.48
## 7 0.24 Very Good   I     VVS1   62.3    57   336  3.95  3.98  2.47
## 8 0.26 Very Good   H     SI1   61.9    55   337  4.07  4.11  2.53
## 9 0.22     Fair      E     VS2   65.1    61   337  3.87  3.78  2.49
## 10 0.23 Very Good  H     VS1   59.4    61   338  4.00  4.05  2.39
## # ... with 53,930 more rows
```

Los tibbles, como los datatables tienen una impresión diferente que un dataframe. En este caso, se imprime la dimensión y los primeros 10 renglones.

Al crear un tibble nunca hay conversión de tipos (no se convierte de carácter a factor), no se cambian los nombres de las variables y no se crean nombres para las filas. Se permite también tener nombres de variables que no son sintácticamente válidos, para utilizar estos nombres, se declaran con acento invertido ‘`’

```
tibble(
  `2000` = c(100:104),
  `una-variable` = c(200:204),
  `año` = c(300:304)
)

## # A tibble: 5 × 3
##   `2000` `una-variable`  año
##   <int>       <int> <int>
## 1 100         200   300
## 2 101         201   301
## 3 102         202   302
## 4 103         203   303
## 5 104         204   304
```

Para utilizar nombres no sintácticamente válidos es necesario conjugarlos con el acento invertido al nombrarlos en otros paquetes como dplyr, ggplot2, entre otros que son compatibles

con el objeto `tibble`.

La otra diferencia importante entre los `tibbles` y los `data.frames` es que pueden incorporar elementos de listas en una columna en forma sencilla (Hadley Wickham 2016a).

```
try(df <- data.frame(x = list(1:2, 3:5)))

df<- data.frame(x = I(list(1:2, 3:5)))
df

##           x
## 1     1, 2
## 2 3, 4, 5

tb <- tibble(x = list(1:2, 3:5))
tb

## # A tibble: 2 × 1
##       x
##   <list>
## 1 <int [2]>
## 2 <int [3]>
```

Objetos importantes

Infinito

`Inf` es como R denomina al infinito. En el mundo de R se permite también positivo o negativo.

```
1/0

## [1] Inf

1/Inf

## [1] 0
```

No es un número

`NaN` es como R denota a algo que no es un número (literal: *not a number*).

```
0/0

## [1] NaN
```

Valores perdidos (missing values)

En la página ?? se habló de otros objetos en R. De particular importancia es `NA` para valores perdidos en general y `NaN` para operaciones matemáticas no definidas. Lógicamente, podemos preguntar a R si un objeto es de este tipo

```
is.na()
is.nan()
```

Los valores `NA` tienen una clase particular. Puede haber valores perdidos enteros `NA_integer_` o caracteres `NA_character_`. `NaN` es un `NA` pero no al revés.

```
x <- c(1, 4, 6, NA, NaN, 45)
is.nan(x)

## [1] FALSE FALSE FALSE FALSE  TRUE FALSE

is.na(x)

## [1] FALSE FALSE FALSE  TRUE  TRUE FALSE
```

Cuando tenemos un dataframe que tiene valores perdidos y lo queremos incorporar, por ejemplo, a un modelo de regresión, lo primero que hará el método es excluir todos los renglones que tengan *algún* valor perdido usando `na.exclude(datos)`. Normalmente, este `no` es el tratamiento deseado para valores perdidos pero es el comportamiento por *default*.

Estructuras de control

Las estructuras de control permiten controlar la ejecución. Pueden ser utilizadas en un script o dentro de funciones. Entre las más comunes se encuentran:

- `if, else`
- `for`
- `while`
- `repeat`
- `break`
- `next`
- `return`

If

```
if ( condicion ) {
  # Cuando se cumple la condicion, ejecuta esto
} else {
```

```
# Para todo lo que no se cumple la condicion, ejecuta esto
}
```

Ejemplo,

```
x <- 1:20
if ( sample(x, 1) <= 10 ) {
  print("x es menor o igual que 10")
} else {
  print("x es mayor que 10")
}
```

```
## [1] "x es menor o igual que 10"
```

O lo que es lo mismo:

```
ifelse(sample(x, 1) <= 10, "x es menor o igual que 10", "x es mayor que 10")
## [1] "x es menor o igual que 10"
```

También es posible asignar variables dentro de una condición.

```
if ( sample(x, 1) <= 10 ){
  y <- 0
} else {
  y <- 1
}

# o

y <- if ( sample(x, 1) <= 10 ){
  0
} else {
  1
}
```

For

Un ciclo **for** itera una variable y va realizando, para cada iteración, la secuencia de comandos que se especifica dentro del mismo.

```
for (i in 1:3 ){
  print(paste0("i vale: ", i))
}

## [1] "i vale: 1"
## [1] "i vale: 2"
```

```
## [1] "i vale: 3"
```

Es posible también iterar directamente sobre vectores o partes de vectores.

```
x <- c("Andrea", "Liz", "Edwin", "Miguel")
```

```
for ( i in seq(x) ) {
  print(x[i])
}
```

```
## [1] "Andrea"
```

```
## [1] "Liz"
```

```
## [1] "Edwin"
```

```
## [1] "Miguel"
```

```
for ( e in x ) {
  print(e)
}
```

```
## [1] "Andrea"
```

```
## [1] "Liz"
```

```
## [1] "Edwin"
```

```
## [1] "Miguel"
```

```
for ( i in seq(x) ){
  print(x[i])
}
```

```
for ( i in 1:length(x) ) print(x[i])
```

Podemos incluir **fors** dentro de **fors**.

```
m <- matrix(1:10, 2)
```

```
for( i in seq(nrow(m)) ) {
  for ( j in seq(ncol(m)) ) {
    print(m[i, j])
  }
}
```

Whiles

Otra manera de iterar sobre comandos es con la estructura **while**. A diferencia del **for**, este permite iterar sobre la secuencia de comandos especificada hasta que se cumpla cierta condición lógica. Esta última tiene que variar a lo largo de las iteraciones o es posible generar ciclos infinitos. Esta estructura da mucha flexibilidad.

```
x <- runif(1)

while ( x < 0.20 | i <= 10 ) {
  print(x)
  x <- runif(1)
  i <- i + 1
}
```

Importante

Asegurate de especificar una manera de salir de un ciclo while.

Repeat - Break

```
x <- 1
repeat {
  # Haz algo
  print(x)
  x = x+1
  # Hasta que se cumpla lo siguiente
  if (x == 6){
    break
  }
}
```

Next

```
for (i in 1:20) {
  if (i %% 2 == 0){
    next
  } else {
    print(i)
  }
}
```

Este ciclo itera sobre los valores del 1 al 20 e imprime los valores impares.

Importante

R no es muy eficiente cuando se combina con estructuras de control tipo for o while. Sin embargo, estas estructuras son muy comunes y es útil conocerlas. Normalmente, se recomienda utilizar estructuras vectorizadas (como ifelse) pues, de

esta manera, R es mucho más eficiente.

Material adicional

- Curso de **swirl R Programming**, módulos 4 a 9.
- Curso **Introduction to R** de Data Camp.
- Curso **TryR** de Code School.

Capítulo 4

Vectorización, la familia apply y otros

En este capítulo se profundiza en la forma en la que es posible operar los distintos objetos examinados en el capítulo anterior. En particular, se revisa con detalle las diferentes maneras con las que se pueden extraer *subconjuntos* de las estructuras de datos.

Para *vectores*, se detallan los seis tipos de extracciones y, posteriormente, se describe cómo esos seis métodos aplican en *matrices* y *dataframes*.

Se describe la asignación de valores a los subconjuntos extraídos, así como los operadores lógicos disponibles en R y sus particularidades.

Se introducen también distintas aplicaciones del material revisado de forma que sea posible realizar operaciones que típicamente se hacen en Excel, como `buscarv` o `buscarh`. Se revisa cómo expandir una base con pesos, cómo seleccionar muestras aleatorias y cómo generar datos a partir de realizaciones de distribuciones específicas implementadas en R.

Por último, se revisa la estrategia separa, aplica y combina (SAC), así como su implementación en R básico en la familia `apply`. Esta estrategia permitirá que fácilmente se realicen operaciones complejas sobre grupos, especificando explícitamente la estructura de datos de entrada y de salida que se está buscando.

Subconjuntos de diferentes estructuras de datos

Esta sección está basada en Hadley Wickham (2014a, Subsetting) disponible en línea.

Aprender a extraer subconjuntos de los datos es importante y permite realizar operaciones complejas con los mismos. De los conceptos importantes que se deben aprender son

- Los operadores para extraer subconjuntos (subsetting operators)
- Los 6 tipos de extracciones de subconjuntos
- Las diferencias a la hora de extraer subconjuntos de las diferentes estructuras de datos (factores, listas, matrices, dataframes)
- El uso de la extracción de subconjuntos junto a asignar variables.

Cuando tenemos que extraer pedazos de los datos (o analizar solamente parte de éstos), necesitamos complementar `str()` con `[]`, es decir, la estructura nos dirá cómo utilizar el operador subconjunto de manera que de hecho extraigamos lo que queremos.

Operadores para extraer subconjuntos

Dependiendo la estructura de datos que tenemos, será la forma en la que extraemos elementos de ella. Hay dos operadores de subconjunto: `[]` y `$`. `[]` se parece a `[` pero regresa un solo valor y te permite sacar pedazos de una lista. `$` es un atajo útil para `[]`.

Vectores atómicos

¿De qué formas puedo extraer elementos de un vector? Hay varias maneras **sin importar** la clase del vector.

- **Enteros positivos** regresan los elementos en las posiciones especificadas en el orden que especificamos.

```
x <- c(5.6, 7.8, 4.5, 3.3)

x[c(3, 1)]

## [1] 4.5 5.6
## Si duplicamos posiciones, nos regresa resultados duplicados
x[c(1, 1, 1)]

## [1] 5.6 5.6 5.6
## Si usamos valores reales, se coerciona a entero
x[c(1.1, 2.4)]
```

```
## [1] 5.6 7.8
x[order(x)]
```

```
## [1] 3.3 4.5 5.6 7.8
x[order(x, decreasing = T)]
```

```
## [1] 7.8 5.6 4.5 3.3
```

- **Enteros negativos** omiten los valores en las posiciones que se especifican.

```
x

## [1] 5.6 7.8 4.5 3.3

x[-c(3, 1)]
```

```
## [1] 7.8 3.3
```

Mezclar no funciona.

```
x[c(-3, 1)]
```

- **Vectores lógicos** selecciona los elementos cuyo valor correspondiente es TRUE. Esta es una de los tipos más útiles.

```
x[c(TRUE, TRUE, FALSE, FALSE)]
```

```
## [1] 5.6 7.8
```

```
x[c(TRUE, FALSE)] # Autocompleta el vector lógico al tamaño de x
```

```
## [1] 5.6 4.5
```

```
x[c(TRUE, TRUE, NA, FALSE)]
```

```
## [1] 5.6 7.8 NA
```

- **Nada** si no especifico nada, me regresa el vector original

```
x[]
```

```
## [1] 5.6 7.8 4.5 3.3
```

- **Cero** el índice cero no aplica en R, te regresa el vector vacío

```
x[0]
```

```
## numeric(0)
```

- Si el vector tiene **nombres** también los puedo usar.

```
names(x) <- c("a", "ab", "b", "c")
x["ab"]
```

```
## ab
```

```
## 7.8
```

```
x["d"]
```

```
## <NA>
```

```
## NA
```

```
x[grep("a", names(x))]
```

```
## a ab
```

```
## 5.6 7.8
```

Las **listas** operan básicamente igual a vectores recordando que si usamos [regresa una lista y tanto [[y \$ extrae componentes de la lista.

Matrices y arreglos

Para estructuras de mayor dimensión se pueden extraer de tres maneras:

- Con vectores múltiples
- Con un solo vector
- Con una matriz

```
m <- matrix(1:12, nrow = 3)
colnames(m) <- LETTERS[1:4]
m[1:2, ]
```

```
##      A B C D
## [1,] 1 4 7 10
## [2,] 2 5 8 11
m[c(T, F, F), c("B", "C")]
```

```
## B C
## 4 7
m[1, 4]
```

```
## D
## 10
```

Como ven, es solamente generalizar lo que se hace en vectores replicándolo al número de dimensiones que se tiene.

```
m[c(T, F, F)]
## [1] 1 4 7 10
class(m[c(T, F, F)])
## [1] "integer"
```

[simplifica al objeto. En matriz, me quita la dimensionalidad, en listas me da lo que esta dentro de esa celda.

Dataframes

```
df <- data.frame(x = 1:3, y = 3:1, z = letters[1:3])
```

```
df[c(1, 2), ]
```

```
##   x y z
## 1 1 3 a
## 2 2 2 b
```

```
df[, c(1, 2)]
```

```
##   x y  
## 1 1 3  
## 2 2 2  
## 3 3 1
```

```
df[, c("z", "x")]
```

```
##   z x  
## 1 a 1  
## 2 b 2  
## 3 c 3
```

```
df[c("z", "x")]
```

```
##   z x  
## 1 a 1  
## 2 b 2  
## 3 c 3
```

```
class(df[, c("z", "x")])
```

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

```
class(df[c("z", "x")])
```

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

```
str(df["x"])
```

```
## 'data.frame':   3 obs. of  1 variable:  
##   $ x: int  1 2 3
```

```
str(df[, "x"])
```

```
##  int [1:3] 1 2 3
```

```
str(df$x)
```

```
##  int [1:3] 1 2 3
```



Ejercicios

1. Utiliza la base mtcars
2. Arregla los errores al extraer subconjuntos en dataframes


```
mtcars[mtcars$cyl = 4, ]
mtcars[-1:4, ]
mtcars[mtcars$cyl <= 5]
mtcars[mtcars$cyl == 4 | 6, ]
```
3. ¿Por qué al correr `x <- 1:5; x/NA` obtengo valores perdidos?
4. Genera una matriz cuadrada tamaño 5 llamada m. ¿Qué te da correr `m[upper.tri(m)]`?
5. ¿Por qué al realizar `mtcars[1:20]` me da un error? ¿Por qué `mtcars[1:2]` no me lo da? ¿Por qué `mtcars[1:20,]` es distinto?
6. Haz una función que extraiga la diagonal de la matriz m que creaste antes. Debe dar el mismo resultado que ejecutar `diag(m)`
7. ¿Qué hace `df[is.na(df)] <- 0`?

Asignar a un subconjunto

Muchas veces lo que necesitamos es encontrar ciertos valores para poder reemplazarlos con algo más. Por ejemplo, muchas veces queremos imputar valores perdidos con cierto valor.

```
# Variables continuas
x <- c(1, 2, 3, NA, NaN, 7)
media <- mean(x, na.rm = T)
media

## [1] 3.25
x[is.na(x)] <- media
x

## [1] 1.00 2.00 3.00 3.25 3.25 7.00

# Variables discretas
x <- c(rep("azul", 3), "verde", NA, "verde", rep("rojo", 4))
x

## [1] "azul"  "azul"  "azul"  "verde" NA       "verde" "rojo"  "rojo"
## [9] "rojo"  "rojo"

moda <- names(table(x))[which(table(x) == max(table(x)))] # Engorroso, no?
x[is.na(x)] <- moda
x

## [1] "azul"  "azul"  "azul"  "verde" "rojo"  "verde" "rojo"  "rojo"
## [9] "rojo"  "rojo"
```

```
# Puedo reemplazar partes de un vector
x <- 1:5
x[c(1, 2)] <- 2:3
x
## [1] 2 3 3 4 5

# Las longitudes de las asignaciones tienen que ser iguales
x[-1] <- 4:1
x
## [1] 2 4 3 2 1

# No se revisan duplicados
x[c(1, 1)] <- 2:3
x
## [1] 3 4 3 2 1

# Puedo sustituir valores considerando toda la logica
x <- c(1:10)
x[x > 5] <- 0
x
## [1] 1 2 3 4 5 0 0 0 0 0
```

Por último, es útil notar la utilidad de asignar utilizando la forma de asignar **nada** mencionada anteriormente.

```
class(mtcars)

## [1] "data.frame"

mtcars[] <- lapply(mtcars, as.integer)
class(mtcars)

## [1] "data.frame"

dim(mtcars)

## [1] 32 11

mtcars <- lapply(mtcars, as.integer)
class(mtcars)

## [1] "list"

dim(mtcars)

## NULL
```

Asignar utilizando el operador de suconjunto a nada nos permite preservar la estructura del objeto original así como su clase.

En el caso de listas, si combinamos un operador de subconjunto mas asignación a nulo, podemos remover objetos de ésta.

```
x <- list(a = 1, b = 2)
x[[2]] <- NULL
str(x)
```

```
## List of 1
## $ a: num 1
x["b"] <- list(NULL)
str(x)
```

```
## List of 2
## $ a: num 1
## $ b: NULL
```

Operadores lógicos

| Operador | Descripción |
|-----------|-------------------------|
| < | menor que |
| <= | menor o igual que |
| > | mayor que |
| == | exactamente igual que |
| != | diferente de |
| !x | no x |
| x y | x O y |
| x & y | x Y y |
| isTRUE(x) | checa si x es verdadero |

Ejemplo: Supongamos que queremos saber qué elementos de x son menores que 5 ó mayores que 8.

```
x <- c(1:10)
x[(x>8) | (x<5)]
```

```
## [1] 1 2 3 4 9 10
```

```
# ¿Cómo funciona?
x
```

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

```
x > 8
```

```
## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE
```

```
x < 5

## [1] TRUE TRUE TRUE TRUE FALSE FALSE FALSE FALSE FALSE FALSE

x > 8 | x < 5

## [1] TRUE TRUE TRUE TRUE FALSE FALSE FALSE TRUE TRUE

# x > 8 || x < 5
x[c(T,T,T,T,F,F,F,F,T,T)]
```

[1] 1 2 3 4 9 10

[|| vs. | y && vs. &
La diferencia entre & y && (o | y ||) es que el primero es vectorizado y el segundo no.]

Ejercicio ¿Qué crees que pasa en las siguientes situaciones?

```
rm(list = ls())
TRUE || a
FALSE && a
TRUE && a
TRUE | a
FALSE & a
```

La forma larga (la versión doble) no parece ser muy útil. El propósito de ésta es que es más apropiado cuando se programa usando estructuras de control, por ejemplo, en **ifs***

```
if( c(T, F) ) print("Hola")
```

[1] "Hola"

Poner el && me garantiza que la condicional será evaluado sobre un único valor falso/verdadero.

```
(-2:2) >= 0

## [1] FALSE FALSE TRUE TRUE TRUE

(-2:2) <= 0

## [1] TRUE TRUE TRUE FALSE FALSE

((-2:2) >= 0) && ((-2:2) <= 0)
```

[1] FALSE

Ejercicio Explora los siguientes comandos

```
impares <- 1:10 %% 2 == 1
mult.3 <- 1:10 %% 3 == 0
```

```
impares & mult.3
impares | mult.3

xor(impares, mult.3)
```

Aplicaciones

Una de las formas más fáciles de frustrarse con R (y con cualquier otro lenguaje) es no saber decirle al lenguaje lo que se desea hacer. Entender cómo manipular las estructuras de datos y la lógica detrás de su comportamiento ahorra mucho sufrimiento y permite adaptarse ante cosas que necesitamos que aún no se encuentran implementadas por alguien más de una manera más sencilla.

Con saber de subconjuntos podemos realizar varias tareas indispensables.

Buscarv o buscarh

Excel es excelente haciendo estas tareas. Lo malo de excel es que no es **reproducible**. Es muy común que resulte imposible llegar de los datos originales al resultado final pues muchos pasos intermedios de limpieza no están documentados de forma alguna. Un *script* de limpieza nos permite no solamente ir del *raw* a la estructura de datos limpia y analizable sino que permite que alguien más verifique las operaciones que se están realizando, se identifiquen errores y que, cuando nos llega un nuevo mes, sea trivial incluir estos datos al resultado final.

```
rm(list = ls())
x <- c("m", "f", "u", "f", "f", "m", "m")
busca <- c(m = "Male", f = "Female", u = NA)
busca[x]

##          m      f      u      f      f      m      m
##  "Male"  "Female"    NA "Female" "Female"  "Male"  "Male"
unname(busca[x])

## [1] "Male"   "Female"  NA        "Female" "Female"  "Male"   "Male"
c(m = "humano", f = "humano", u = "desconocido") [x]

##          m      f      u      f
##  "humano" "humano" "desconocido" "humano" "humano"
##          m      m
##  "humano" "humano"
```

Esto nos permite pegar un vector a una base de datos de acuerdo a una condición.

```

calificaciones <- c(10, 9, 5, 5, 6)
aprueba <- data.frame(
  calificacion = 10:1,
  descripcion = c(rep("excelente", 2), "bueno", rep("aceptable", 2),
                 , rep("no satisfactorio", 5)),
  aprobatorio = c(rep(T, 5), rep(F, 5))
)
id <- match(calificaciones, aprueba$calificacion)

aprueba[id, ]

```

| | calificacion | descripcion | aprobatorio |
|--------|--------------|------------------|-------------|
| ## 1 | 10 | excelente | TRUE |
| ## 2 | 9 | excelente | TRUE |
| ## 6 | 5 | no satisfactorio | FALSE |
| ## 6.1 | 5 | no satisfactorio | FALSE |
| ## 5 | 6 | aceptable | TRUE |

Ejercicios

- Realiza la misma operación con las calificaciones pero utilizando los nombres de las filas, es decir, los rownames(aprueba)
- Carga la librería ggplot2 y utiliza la base de datos diamonds
- Utiliza el comando match para quedarte con las variables cut y x
- Genera la variable categórica tal que, si el precio es mayor que 5,000 el valor de price.cat es cara, si es mayor que 2,000 es normal y barata en otro caso.

Muestras aleatorias

Podemos utilizar índices enteros para generar muestras aleatorias de nuestras bases de datos o de nuestros vectores.

```

set.seed(102030)
aprueba[sample(nrow(aprueba)), ]

```

| | calificacion | descripcion | aprobatorio |
|-------|--------------|------------------|-------------|
| ## 10 | 1 | no satisfactorio | FALSE |
| ## 2 | 9 | excelente | TRUE |
| ## 8 | 3 | no satisfactorio | FALSE |
| ## 7 | 4 | no satisfactorio | FALSE |
| ## 9 | 2 | no satisfactorio | FALSE |
| ## 3 | 8 | bueno | TRUE |
| ## 1 | 10 | excelente | TRUE |
| ## 6 | 5 | no satisfactorio | FALSE |
| ## 4 | 7 | aceptable | TRUE |

```
## 5           6      aceptable      TRUE
aprueba[sample(nrow(aprueba), replace = T, size = 5), ]
##   calificacion      descripcion aprobatorio
## 4             7      aceptable      TRUE
## 1             10     excelente      TRUE
## 4.1            7      aceptable      TRUE
## 7             4 no satisfactorio    FALSE
## 2             9     excelente      TRUE
```



Ejercicios

1. Utiliza la base de datos de iris y genera un conjunto de prueba y uno de entrenamiento correspondientes al 20 y 80 % de los datos, respectivamente.
2. Genera un vector x de tamaño 1000 con realizaciones de una normal media 10, varianza 3.
3. Crea 100 muestras bootstrap del vector x.
4. Calcula la media para cada una de tus muestras.
5. Grafica con la función hist() el vector de medias de tus muestras.
6. Genera un vector l de letras, tamaño 10 y ordénalo. (Usa letters y order).
7. Ordena la base cars de acuerdo a distancia, en forma descendiente (muestra la cola -usa tail- de la base ordenada).

Expande bases

Ahora, a veces tenemos tablas de resumen pero quisieramos extraer los datos originales. Combinamos rep con subconjuntos de enteros para expandir.

```
df <- data.frame(
  color = c("azul", "verde", "amarillo"),
  n = c(4, 3, 5)
)
df

##      color n
## 1      azul 4
## 2      verde 3
## 3 amarillo 5

df[rep(1:nrow(df), df$n), ]

##      color n
## 1      azul 4
## 1.1    azul 4
## 1.2    azul 4
## 1.3    azul 4
```

```
## 2      verde 3
## 2.1    verde 3
## 2.2    verde 3
## 3     amarillo 5
## 3.1   amarillo 5
## 3.2   amarillo 5
## 3.3   amarillo 5
## 3.4   amarillo 5
```

Which, intersect y union

Ya estuvimos utilizando otras aplicaciones de estos comandos: ordenamientos, selección de filas o columnas según una condición lógica.

También utilizamos un comando muy útil llamado which.

```
set.seed(45)
x <- sample(letters, 10)
x <= "e"

## [1] FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE TRUE TRUE

which(x <= "e")

## [1] 7 9 10
```

Junto con which, puedes usar intersect y union.

```
pares <- 1:10 %% 2 == 0
m.5 <- 1:10 %% 5 == 0

c(1:10)[union(which(pares), which(m.5))]
c(1:10)[intersect(which(pares), which(m.5))]
c(1:10)[which(xor(pares, m.5))]
```

Generación de datos a partir de distribuciones

R básico provee funciones de distribución de probabilidad (p), funciones de densidad (d), funciones cuantiles (q) y generación de números aleatorios (r) para varias distribuciones como la binomial, chi-cuadrada, gamma, geométrica, etc. Dutang (2016) resumen otros paquetes en donde se encuentran implementadas otras funcionalidades o distribuciones multivariadas y cópulas.

En las funciones de R básico, podemos construir un *dataframe* con realizaciones de distintas distintas distribuciones, por ejemplo, a continuación generamos la variable *x* como realizaciones de una normal, *y* con realizaciones de una exponencial y *z* con realizaciones de una uniforme.

```
df <- data.frame(  
  x = rnorm(100)  
 , y = rexp(100)  
 , z = runif(100)  
)
```

Split-apply-combine

Muchos problemas en el análisis de datos pueden ser resueltos aplicando la estrategia separa, aplica y combina (SAC) en donde divides un problema en pequeños pedazos manejables, operas en forma independiente cada uno de éstos y después combinás los resultados obtenidos (Hadley Wickham 2011).

Esta estrategia se utiliza en diversas etapas del análisis de datos, por ejemplo (Hadley Wickham 2011):

- **Preparación de datos.** Cuando se crean nuevas variables según grupos, cuando se realizan ordenamientos por grupos, cuando se estandariza o normaliza variables.
- **Estadística descriptiva.** Cuando se crean agregados por grupos como sus medias o medianas.
- **Modelado.** Cuando se calculan modelos separados para cada panel en un estudio de este tipo. Estos modelos pueden examinarse por separado o unificarse para construir modelos más sofisticados que los conjuguen.

Esta estrategia se utiliza en muchas herramientas: en las tablas dinámicas de Microsoft Excel, el operador `group by` de SQL, el argumento `by` disponible en algunos procedimientos de SAS (Hadley Wickham 2011).

El paradigma split-apply-combine se resume en la figura 4.1.

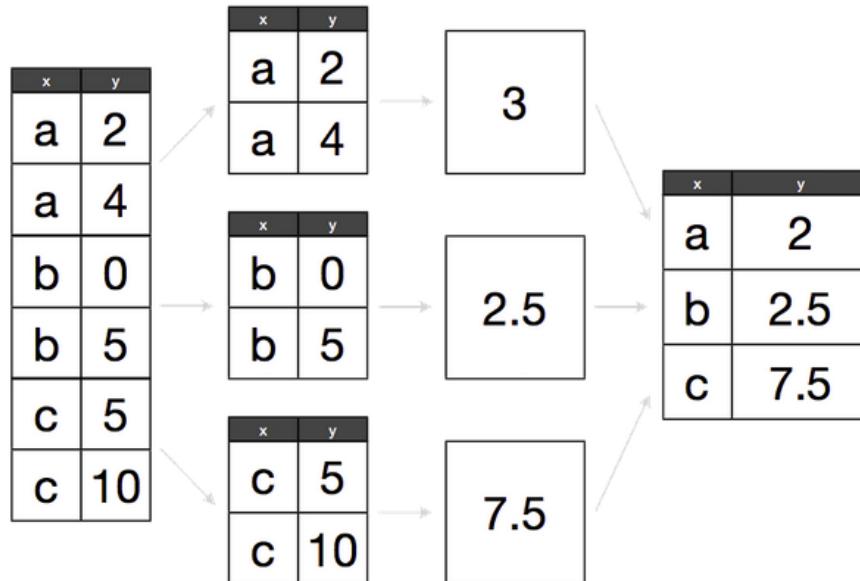


Figura 4.1: Ejemplificación del split-apply-combine Vaidyanathan (2014, Split-Apply-Combine).

Replicamos los vectores x y y de la figura en un *dataframe*:

```

letras <- c("a", "b", "c")
df <- data.frame(
  x = sort(letras[rep(seq(letras), 2)]),
  y = c(2, 4, 0, 5, 5, 10)
)
df

##   x   y
## 1 a   2
## 2 a   4
## 3 b   0
## 4 b   5
## 5 c   5
## 6 c  10

```

Queremos estimar la **media** de los valores en el vector y para cada tipo de letra en el vector x . Esto lo podemos hacer utilizando la estrategia **SAC**, como en la figura.

```

# Dividimos
for (l in unique(df$x)){
  print(paste0("Grupo con letra: ", l))
  print(df[l == df$x, ])
}

## [1] "Grupo con letra: a"
##   x   y
## 1 a   2
## 2 a   4
## [1] "Grupo con letra: b"
##   x   y
## 3 b   0
## 4 b   5
## [1] "Grupo con letra: c"
##   x   y
## 5 c   5
## 6 c  10

# Aplicamos
for (l in unique(df$x)){
  print(paste0("Media para valores de letra: ", l))
  print(mean(df[l == df$x, ]$y))
}

## [1] "Media para valores de letra: a"
## [1] 3
## [1] "Media para valores de letra: b"
## [1] 2.5

```

```

## [1] "Media para valores de letra: c"
## [1] 7.5

# Combinamos
medias <- list()
for (l in unique(df$x) ){
  medias[[l]] <- mean(df[l == df$x, ]$y)
}
as.data.frame(list(letras = names(medias), medias = unname(unlist(medias))))

##   letras medias
## 1      a    3.0
## 2      b    2.5
## 3      c    7.5

```

R tiene muchas funciones que facilitan realizar este tipo de operaciones. En particular, la familia `apply` fue pensada para realizarlas. Cada una de las funciones en esta familia recibe una estructura de datos en particular, aplica de determinada manera la función que se le especifica y combina los resultados de una forma específica.

apply

`apply` aplica una función a cada fila o columna en una matriz.

1. **Separa:** por columna o fila según se especifica en el parámetro `MARGIN` (1 para filas, 2 para columnas).
2. **Aplica:** la función que se especifica en el parámetro `FUN`.
3. **Combina:** regresa un vector con los resultados.

```

m <- matrix(c(1:5, 6:10), nrow = 5, ncol = 2)
# 1 is the row index 2 is the column index
m

```

```

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

```

```
apply(m, 1, sum)
```

```
## [1] 7 9 11 13 15
```

```
apply(m, 2, sum)
```

```
## [1] 15 40
```

 Ejercicio

Haz una función que reciba un vector y devuelva la suma de la posición $v_i + v_{i+1}$. Para el n-esimo elemento, suma el primero. Aplica esa función a las columnas y filas de la matriz m.

```
# Respuesta
suma.rec <- function(v){
  resultado <- c()
  for (e in seq(length(v) - 1)){
    resultado[e] <- v[e] + v[e + 1]
  }
  resultado[length(v)] <- v[length(v)] + v[1]
  return(resultado)
}

m
apply(m, 1, suma.rec)
apply(m, 2, suma.rec)
```

lapply

`lapply` aplica una función a cada elemento en una lista. Como sabemos, un `data.frame` es únicamente un estilo particular de lista tal que todos sus elementos tienen el mismo tamaño. Por ende, también podemos utilizar `lapply` para iterar sobre las columnas de un `data.frame`.

```
lista <- list(a = 1:10, b = 2:20)
lapply(lista, mean)
```

```
## $a
## [1] 5.5
##
## $b
## [1] 11
df <- data.frame(a = 1:10, b = 11:20)
lapply(df, mean)
```

```
## $a
## [1] 5.5
##
## $b
## [1] 15.5
```

 Ejercicio

El `summary` de un `data.frame` genera un resumen para los vectores que la conforman de acuerdo a la clase de la misma. Genera una función que regrese una tabla de frecuencias para factores y caracteres o una lista con media, desviación estándar para vectores numéricos o enteros. Aplícalo a la base diamonds usando `lapply`.

```
# Respuesta
mi.resumen <- function(vector){
  if( class(vector) == "factor" || class(vector) == "character"){
    table(vector)
  } else if ( class(vector) == "numeric" || class(vector) == "integer") {
    list(media = mean(vector), de = sqrt(var(vector)))
  }
}

lapply(names(diamonds), FUN = function(c) mi.resumen(diamonds[, c]))
```

sapply

`sapply` es otra versión de `lapply` que regresa una lista o un vector, dependiendo si se especifica el parámetro `simplify = T` y si la función aplicada regresa un único valor.

```
x <- sapply(lista, mean, simplify = F)
x
```

```
## $a
## [1] 5.5
##
## $b
## [1] 11
x <- sapply(lista, mean, simplify = T)
x
```

```
##      a      b
##  5.5 11.0
```

 Ejercicio

Obtén un vector tipo carácter con los nombres de las columnas de iris.

```
# Respuesta
sapply(iris, class)
```

 Ejercicio

Repite el ejercicio de la suma rara pero usa `sapply`.

Recuerda la instrucción: Haz una función que reciba un vector y devuelva la suma de la posición $v_i + v_{i+1}$. Para el n -esimo elemento, suma el primero. Utiliza `sapply` para realizar esta operación.

```
# Respuesta
x <- 1:10
sapply(seq(x), FUN = function(i){
  if( i == length(x) ){
    x[1] + x[i]
  } else {
    x[i] + x[i + 1]
  }
})
```

mapply

`mapply` es como la versión multivariada de `sapply`. Le aplica una función a todos los elementos correspondientes de un argumento.

```
11 <- list(a = c(1:5), b = c(6:10))
12 <- list(c = c(11:15), d = c(16:20))
11

## $a
## [1] 1 2 3 4 5
##
## $b
## [1] 6 7 8 9 10

12

## $c
## [1] 11 12 13 14 15
##
## $d
## [1] 16 17 18 19 20

mapply(sum, 11$a, 11$b, 12$c, 12$d)

## [1] 34 38 42 46 50
11[["a"]][1] + 11[["b"]][1] + 12[["c"]][1] + 12[["d"]][1]

## [1] 34
```

 Ejercicio

Crea una matriz de 4 x 4 donde el primer renglón sea de unos, el segundo de dos, el tercero de 3 y el cuarto de 4. Usa `mapply` para hacerlo.

```
# Respuesta
m <- matrix(c(rep(1, 4), rep(2, 4), rep(3, 4), rep(4, 4)), nrow = 4, byrow = T)
m

me <- t(mapply(rep, 1:4, 4))
me
```

tapply

`tapply` le aplica una función a subconjuntos de un vector.

```
head(warpbreaks)
```

```
##    breaks  wool tension
## 1      26     A      L
## 2      30     A      L
## 3      54     A      L
## 4      25     A      L
## 5      70     A      L
## 6      52     A      L

with(warpbreaks, tapply(breaks, list(wool, tension), mean))

##            L         M         H
## A 44.55556 24.00000 24.55556
## B 28.22222 28.77778 18.77778

tapply(warpbreaks$breaks,
       list(wool = warpbreaks$wool, tension = warpbreaks$tension),
       mean)

##    tension
## wool        L         M         H
## A 44.55556 24.00000 24.55556
## B 28.22222 28.77778 18.77778
```

 Ejercicio

Utiliza la función `tapply` y la base de datos `diamonds` (que está dentro del paquete `ggplot2`) para obtener las medias de la variable `carat` para los grupos formados por la variable categórica `cut` y la variable categórica `color`.

```
# Respuesta
library(ggplot2)
with(diamonds,
  tapply(carat,
    list(cut, color),
    mean))
```

by

by le aplica una función a subconjuntos de un `data.frame`. Se divide un `data.frame` según los valores de uno o más factores. Se aplica la función FUN a cada subconjunto.

```
head(iris)

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          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          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa

by(data = iris[, 1:2], INDICES = iris[, "Species"], FUN = summary)

## iris[, "Species"]: setosa
##   Sepal.Length   Sepal.Width
##   Min.   :4.300   Min.   :2.300
##   1st Qu.:4.800   1st Qu.:3.200
##   Median :5.000   Median :3.400
##   Mean   :5.006   Mean   :3.428
##   3rd Qu.:5.200   3rd Qu.:3.675
##   Max.   :5.800   Max.   :4.400
## -----
## iris[, "Species"]: versicolor
##   Sepal.Length   Sepal.Width
##   Min.   :4.900   Min.   :2.000
##   1st Qu.:5.600   1st Qu.:2.525
##   Median :5.900   Median :2.800
##   Mean   :5.936   Mean   :2.770
##   3rd Qu.:6.300   3rd Qu.:3.000
##   Max.   :7.000   Max.   :3.400
## -----
## iris[, "Species"]: virginica
##   Sepal.Length   Sepal.Width
##   Min.   :4.900   Min.   :2.200
```

```
## 1st Qu.:6.225 1st Qu.:2.800
## Median :6.500 Median :3.000
## Mean   :6.588 Mean   :2.974
## 3rd Qu.:6.900 3rd Qu.:3.175
## Max.   :7.900  Max.   :3.800
```

Puedo calcular, por ejemplo, la suma de los valores del largo y ancho de los sépalos en la base de datos iris según la especie.

```
res <- by(iris[, c("Sepal.Length", "Sepal.Width")], iris[, "Species"], sum)
```

Posteriormente, se pueden combinar los elementos.

```
as.data.frame(list(
  "species" = names(res),
  "suma" = sapply(seq(length(res)), FUN = function(i) res[[i]]))
```

```
##       species suma
## 1      setosa 421.7
## 2 versicolor 435.3
## 3 virginica 478.1
```



Ejercicio

Vuelve a utilizar la base de diamonds para calcular el promedio de carat según cut y color.

```
# Respuesta
library(ggplot2)
head(diamonds)

res <- by(diamonds[, c("carat")],
          list(cut = as.factor(diamonds$cut),
               , color = as.factor(diamonds$color)))
          , mean, simplify = T)
```

replicate

`replicate` es una función muy útil sobretodo en el contexto de simulación.

```
replicate(5, rnorm(5), simplify = F)
```

```
## [[1]]
## [1] -0.01744899 -0.35920817 -1.46121725 -0.82974515  0.53160175
## 
## [[2]]
## [1] -0.2601847  0.5065636  1.3330763 -1.4945952  0.8039902
```

```

## 
## [[3]]
## [1] -0.2155771 -0.8922644 -0.1466588 -0.3234650 -1.0602366
##
## [[4]]
## [1] -1.4859165  1.4469633  1.5193526  1.4491572  0.3427115
##
## [[5]]
## [1] -0.94652506 -1.25920627 -0.06813108 -0.75359982  0.43192663
replicate(6, rnorm(4), simplify = T)

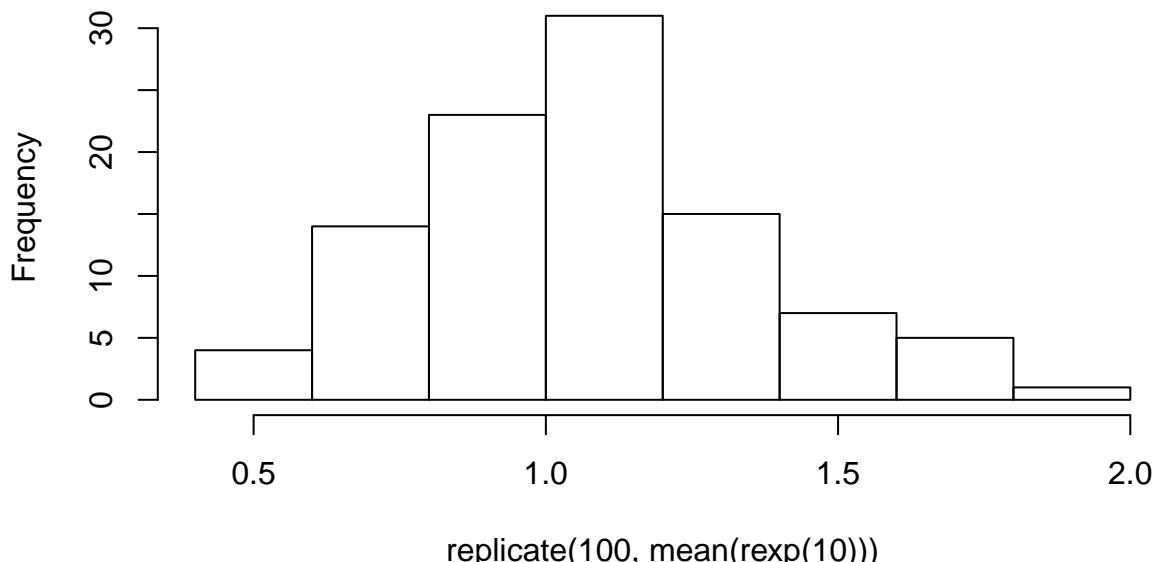
```

```

##          [,1]      [,2]      [,3]      [,4]      [,5]      [,6]
## [1,] 0.12276245 0.4528315 -0.1228450 -1.5292287 -0.42799037 0.6360040
## [2,] 0.00389876 -0.4863693 -2.4085836 -0.7545786  1.92098812 0.3017047
## [3,] -0.80426655  0.1570096  0.1789978  1.6316763  0.03015651 1.4364364
## [4,] 0.12613563 -0.4231777 -0.7788772  0.6512902 -0.21573077 -0.7753874
hist(replicate(100, mean(rexp(10))))

```

Histogram of replicate(100, mean(rexp(10)))



 Ejercicio

Replica el ejercicio de muestras bootstrap utilizando la función `replicate`.

Recordando las instrucciones:

1. Genera un vector `x` de tamaño 1000 con realizaciones de una normal media 10, varianza 3.
2. Crea 100 muestras bootstrap del vector `x`.
3. Calcula la *media* para cada una de tus muestras.
4. Grafica con la función `hist()` el vector de medias de tus muestras.

```
# Respuesta
# 1
x <- rnorm(1000, mean = 10, sd = sqrt(3))
hist( # 4
  replicate(100, # 2
            mean(sample(x, size = 1000, replace = T))), # 3
  main = "Muestras bootstrap",
  xlab = "media",
  ylab = "frecuencia"
)
```

¿Puede ser más fácil?

La familia `apply` viene con R básico. Sin embargo, hay 3 implementaciones excelentes del paradigma split-apply-combine: `plyr`, `dplyr` y `data.table`.

Si la familia `apply` es poderosa, se queda corta comparada con estos tres. `plyr` es la primera versión de SAC de Wickham (Hadley Wickham 2011). Posteriormente, mejoró muchas de las funciones en `dplyr` (Hadley Wickham y Francois s.f.) sobretodo entorno a velocidad y facilidad de uso. `plyr` no termina de ser relevante pues varias de sus funciones aun no están en `dplyr` pero está por ser sustituido.

`data.table` (Dowle y col. 2015), es una implementación con una tradición muy diferente y tiene también funciones muy poderosas aunque con una sintaxis muy distinta a `dplyr`. Es absurdamente eficiente y tiene múltiples aplicaciones.

Muchas de las funciones en `dplyr` también están implementadas en `data.table`. Ambos paquetes son útiles pero priorizan distintos elementos. En el capítulo siguiente se repasarán los verbos en `dplyr`.

Material adicional

- Curso de **swirl R Programming**, módulos 10 a 15.

- Curso **Intermediate R** de Data Camp.
- Curso **Intermediate R - Practice** de Data Camp.

Capítulo 5

Herramientas básicas para un proyecto de datos

Un proyecto de datos tiene una gran cantidad de componentes. Sin embargo, en básicamente todos se necesita iterar sobre el ciclo que se muestra en la figura 5.1.

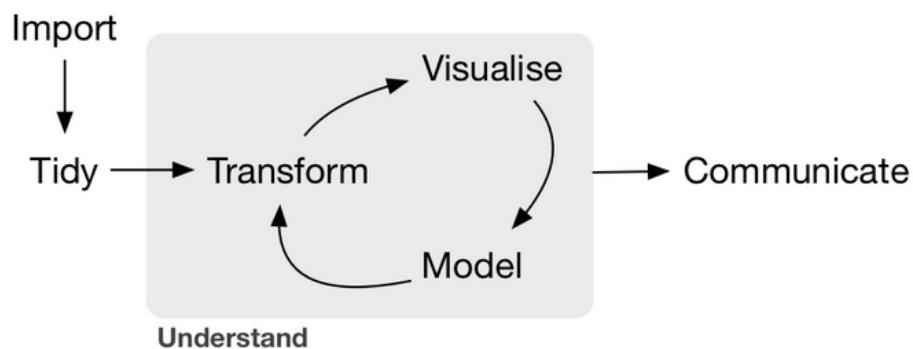


Figura 5.1: Modelo de las herramientas que se necesitan en un proyecto de datos según G. Grolemund y H. Wickham (2016, Introducción).

Primero es necesario **importar** nuestros datos a R. Los datos pueden estar en una gran cantidad de formatos o lugares.

Después, normalmente es necesario **limpiar** nuestros datos, es decir, seguir criterios de datos limpios de tal forma que el cómo guardemos los datos equivalga a la semántica de los datos que tenemos. Es muy importante primero limpiar porque esto provee de consistencia a lo largo del análisis.

Posteriormente, en casi todo proyecto, será necesario **transformar** los datos. A veces esto implica enfocarse en un subconjunto de los datos, generar nuevas variables, calcular estadísticos, arreglar los datos de cierta manera, entre muchos otros.

Solamente después de estas etapas podemos empezar a generar conocimiento a partir de los datos. Para esto tenemos dos herramientas fundamentales: la estadística descriptiva (en el

diagrama reducido a **visualización**) y la generación de **modelos**. La primera es fundamental pues permite derivar preguntas pertinentes a los datos, encontrar patrones, respuestas, plantear hipótesis. Sin embargo, éstas no escalan de la misma manera que los modelos pues éstos, una vez que aceptamos sus supuestos, generan los resultados que esperamos o contestan la pregunta planteada.

Por último, necesitamos **comunicar** los resultados.

En este capítulo nos ocuparemos, por sección, únicamente de 4 de las etapas mencionadas: importación, limpieza, transformación y visualización.

Importación de datos

Esta sección resume algunas de las funciones existentes para **importar** datos de distintos formatos a R. En la figura 5.2 podemos ver la etapa del análisis de datos correspondiente.

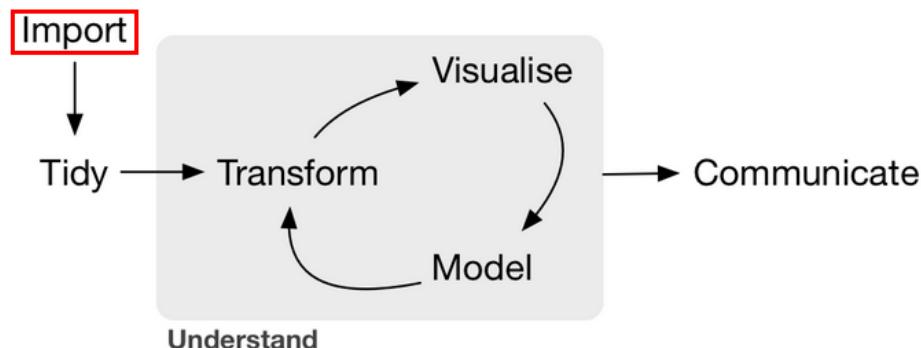


Figura 5.2: Importación en el análisis de datos G. Grolemund y H. Wickham (2016, Introducción).

Para aplicar las herramientas de R a nuestro trabajo, es necesario poder importar nuestros datos a R. R tiene conectores ya implementados para casi cualquier tipo y formato de datos. Entre los más comunes están¹:

| Formato | Lectura | Escritura |
|-----------------|---|---|
| rds | base::readRDS | base::saveRDS |
| separado por * | utils::read.table; readr::read_delim | utils::write.table; readr::write_delim |
| csv | utils::read.csv; readr::read_csv | utils::write.csv; readr::write_csv |
| Microsoft Excel | readxl::read_excel | xlsx::write.xlsx |
| dbf | foreign::read.dbf | foreign::write.dbf |

¹La lista no pretende ser comprehensiva, sin embargo, se presentan algunos de los formatos de datos más comunes. De igual forma, se presentan algunas funciones que sirven para conectar R con datos que están guardados en un manejador de datos externo o en la nube. En caso de presentarse más de un método es porque aunque la recomendación de uso es la función en negritas, la otra opción es más antigua y muy utilizada.

| Formato | Lectura | Escritura |
|--------------------|-----------------------|----------------------|
| IBM SPSS | haven::read_sav | haven::write_sav |
| Stata | haven::read_dta | foreign::write.dta |
| SAS | haven::read_sas | haven::write_sas |
| Google spreadsheet | googlesheets::gs_read | googlesheets::gs_new |
| Google bigquery | bigrquery::query_exec | |
| Heroku Postgres | sql2df | df2sql |
| rdata | base::load | base::save |

Los paquetes utilizados son (corre estos comandos en la consola):

```
library(foreign)
library(haven)
library(readr)
library(readxl)
library(xlsx)
library(googlesheets)
library(bigrquery)
```

Importancia de rutas relativas

Para leer un archivo, recordemos el comando `getwd()` para encontrar la carpeta a la cual R esta dirigido en este momento. Una buena practica es considerar el directorio de trabajo como el lugar en donde esta guardado el archivo o script en el que se trabaja y “moverse” desde ahí hasta el archivo que se quiere leer.

Ya sea en escritura o en lectura, R buscará a partir del directorio de trabajo (el que se despliega con `getwd()`) para buscar a partir de ahí el archivo por leer o para guardar el que se escribirá si se usan rutas relativas.

En caso de usar rutas absolutas (a pesar de que esto **no** es una *buena práctica*) se hará lectura o escritura del archivo en el lugar especificado.



Ejercicios

R tiene conexión con muchos de los formatos en los que se encuentran los datos. Veremos algunos de los mas relevantes.

El código en cada uno de los `chunks` (un chunk es el pedazo del documento en donde hay código de R) está hecho para que puedas correrlo en la consola (excepto cuando dice explícitamente *do not run* (leyenda comúnmente encontrada en los ejemplos de la documentación de las funciones. Con esto entenderás mejor el concepto de rutas relativas.

rds

La extensión `rds` es de las más comúnmente utilizada en R, por ejemplo, para guardar los datos para un paquete. Las funciones pertenecen al `base` (R Core Team 2016a). Permiten guardar un solo objeto de R a un archivo y recuperarlo.

Para **escribirlos**

```
# Creamos un dataframe llamado misdatos
misdatos <- iris
# Los guardamos en comprimido
saveRDS(misdatos, file = "misdatos.rds", ascii = FALSE, version = NULL,
        compress = TRUE, refhook = NULL)
```

Nota como si usas el comando `getwd()` y después vas a la ruta indicada por medio del explorador de archivos, verás en esa carpeta el archivo `misdatos.rds`.

Para **leerlos** usamos la ruta relativa. Dado que los guardamos en el directorio de trabajo actual (Recuerda, se puede cambiar con el comando `setwd`) entonces simplemente los llamamos:

```
misdatos <- readRDS("misdatos.rds")
# Los borramos
file.remove("misdatos.rds")
```

separado por *

Con esto nos referimos a la colección de archivos en texto plano, es decir, `.txt`, `.tsv`, `.psv`, etcétera.

Para **escribirlos** el mas común es `write.table` del paquete `utils` (R Core Team 2016b)

```
# Do not run
write.table(misdatos, file = "~/misdatos.<extension>", append = FALSE
, quote = TRUE, sep = " ", eol = "\n", na = "NA", dec = "."
, row.names = TRUE, col.names = TRUE
, qmethod = c("escape", "double"), fileEncoding = "")
```

En el paquete `readr` se implementa también `write_delim`

```
# Do not run
write_delim(misdatos, path = "~/misdatos.<extension>"
            , delim = "\t", na = "NA", append = FALSE, col_names = !append)
```

Escribamos ahora el *dataframe* `misdatos` en `psv`:

```
write_delim(misdatos, path = "misdatos.psv", delim = "|")
```

Para **leerlos** `read.table` del paquete `utils` (R Core Team 2016b) nos permite especificar casi cualquier particularidad en un archivo de texto plano.

```
# Do not run
misdatos <- read.table("~/misdatos.<extension>", header = FALSE
                      , sep = "", quote = "\\"", dec = "."
                      , numerals = c("allow.loss", "warn.loss", "no.loss")
                      , row.names, col.names, as.is = !stringsAsFactors
                      , na.strings = "NA", colClasses = NA, nrows = -1
                      , skip = 0, check.names = TRUE
                      , fill = !blank.lines.skip, strip.white = FALSE
                      , blank.lines.skip = TRUE, comment.char = "#"
                      , allowEscapes = FALSE, flush = FALSE
                      , stringsAsFactors = default.stringsAsFactors()
                      , fileEncoding = "", encoding = "unknown", text
                      , skipNul = FALSE)
```

La función `read_delim` del paquete `readr` (Hadley Wickham, Hester y Francois 2016) lee los datos más eficientemente a un objeto de clase `tibble`.

```
# Do not run
misdatos <- read_delim(file = "~/misdatos.<extension>", delim
                      , quote = "\\"", escape_backslash = FALSE
                      , escape_double = TRUE, col_names = TRUE
                      , col_types = NULL, locale = default_locale()
                      , na = c("", "NA"), quoted_na = TRUE, comment = ""
                      , trim_ws = FALSE, skip = 0, n_max = Inf
                      , guess_max = min(1000, n_max)
                      , progress = interactive())
```

Leemos el archivo `.psv` que creamos antes:

```
misdatos <- read_delim(file = "misdatos.psv", delim = "|")
# Los borramos
file.remove("misdatos.psv")
```

csv (archivo separado por comas)

Este es un caso particular de archivos de texto en el que se separan por comas. Como es muy utilizado, generalmente se hacen funciones donde ya se especifica el delimitador. Guardaremos el *data frame* `misdatos` en el directorio “arriba” de la ruta que se muestra usando `getwd`. Esto lo podemos hacer anteponiendo al nombre del archivo con `../`.

Para escribirlos

```
# utils
write.csv(misdatos, file = "../misdatos.csv", row.names = F)
# readr
write_csv(misdatos, path = "../misdatos.csv", na = "NA", append = FALSE)
```

Observa en el explorador de archivos en dónde es que se guardó el archivo `misdatos.csv`.

Para **leerlos**, seguimos usando rutas relativas.

```
# utils - como data.frame
misdatos <- read.table("../misdatos.csv", header=TRUE,
  sep=",")  
  
misdatos <- read.csv("../misdatos.csv")  
  
# readr - como tibble
misdatos <- read_csv("../misdatos.csv")  
  
# Lo borro
file.remove("../misdatos.csv")
```

Microsoft Excel

Para **escribirlos** dentro del paquete `xlsx` usamos la función `write.xlsx`

```
misdatos <- iris
write.xlsx(misdatos, "misdatos.xlsx", row.names = F)
```

Para **leerlos** dentro del paquete `readxl` se encuentra la función `read_excel` que es muy útil en este caso.

```
misdatos <- read_excel("misdatos.xlsx", sheet = 1, col_names = TRUE,
col_types = NULL, na = "", skip = 0)  
  
# Lo borro
file.remove("misdatos.xlsx")
```

dbf

Extensión que representa un archivo de una base de datos (*database file*).

Para **escribirlos**:

```
write.dbf(as.data.frame(misdatos), "misdatos.dbf")
```

Nota cómo tuvimos que coercionar el objeto a *data frame*. Como en el ejemplo anterior leímos un *tibble* y el paquete `foreign` es más viejo (y no conoce los *tibbles*) entonces le mandamos un objeto que si conoce.

Veremos más adelante la ventaja de usar *tibbles* aún cuando de vez en cuando se tienen problemas de compatibilidad.

Para **leerlos**:

```
misdatos <- read.dbf("misdatos.dbf")
# Lo borro
file.remove("misdatos.dbf")
```

IBM SPSS

SPSS puede guardar los datos agregando etiquetas y otros metadatos. Para evitar retrabajo, puede leerse directamente a R.

Para **escribirlos**

```
# haven
write_sav(data = misdatos, path = "misdatos.sav")
```

Para **leerlos**

```
# haven - como tibble
misdatos <- read_sav(file = "misdatos.sav", user_na = FALSE)

# Lo borro
file.remove("misdatos.sav")
```

Stata

HOME DIRECTORY

El directorio (carpeta) *home* es muy utilizado. Normalmente, se le denota como \sim y es en donde un sistema operativo guarda los archivos del usuario que se encuentra en sesión. Dependiendo del sistema operativo que utilices, encontrarás este directorio en una ruta específica.

En Microsoft Windows Vista 7, 8 y 10 lo encuentras en `<root>\Users\<username>`.

En Linux lo encuentras en `/home/<username>`.

En Mac OS X lo encuentras en `/Users/<username>`.

Para **escribirlos** en Stata primero tenemos que cambiar los nombres de las variables en el *data frame* pues Stata no admite puntos en los nombres:

```
names(misdatos) <- tolower(gsub("\\.", "_", names(misdatos)))
# foreign
write.dta(data = misdatos, file = "~/misdatos.dta", version = 12)
```

Para leerlos

```
# haven - como tibble
misdatos <- read_dta(file = "~/misdatos.dta", encoding = NULL)

# Lo borramos
file.remove("~/misdatos.dta")
```

SAS

Para usar el paquete `haven` en este caso ejemplificaremos la creación de un directorio de archivos en tu computadora desde R:

```
# Creamos un directorio llamado datos
dir.create("datos_sas")
```

Observa como, en el directorio que se despliega con `getwd` encuentras ahora una carpeta llamada `datos_sas`. Creamos ahí un archivo con la función `write_sas` de `haven`. Nota que, para **escribirlos**, también debemos asegurarnos que los nombres de variables estén compuestos por letras, números o guiones bajos:

```
misdatos <- iris
names(misdatos) <- tolower(gsub("\\.", "_", names(misdatos)))
# haven
write_sas(data = misdatos, path = "datos_sas/misdatos.sas7bdat")
```

Para leerlos, utilizamos `read_sas` del paquete `haven`:

```
# haven - como tibble
misdatos <- read_sas("datos_sas/misdatos.sas7bdat"
                      , catalog_file = NULL, encoding = NULL)
```

Observa desde el explorador de archivos, cómo se creó el archivo dentro del directorio `datos_sas/`. También desde R podemos borrar el directorio:

```
unlink("datos_sas", recursive = T, force = FALSE)
```

La bandera `recursive` le dice al sistema que borre todo lo contenido en esa carpeta.

Google Spreadsheet

Para hacer este ejercicio, debes tener una cuenta de `gmail`.

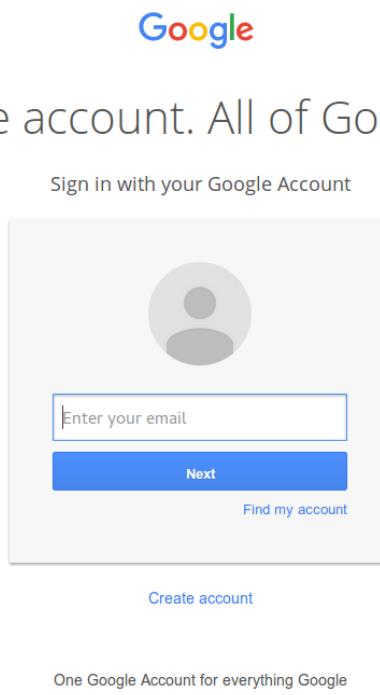
Primero, debe realizarse la autenticación. Esto lo puedes hacer en cualquier sesión interactiva utilizando alguna función del paquete `googlesheets`

```
gs_ls()
```

En la consola de R te aparece:

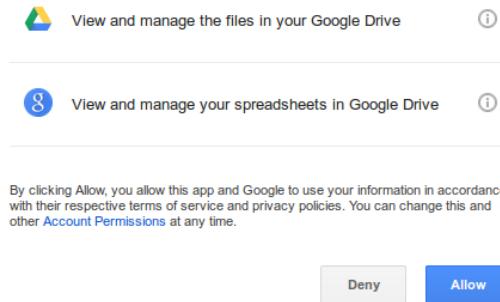
```
> library(googlesheets)
> gs_ls()
Waiting for authentication in browser...
Press Esc/Ctrl + C to abort
```

Se abrirá una ventana del explorador y deberás introducir tus credenciales de tu cuenta de `gmail`



Después de poner tus credenciales, te aparecerá un mensaje pidiendo acceso a tus datos en `drive`:

▼ googlesheets would like to:



Al aceptar darle acceso, recibirás un mensaje parecido a *Authentication complete. Please close this page and return to R.*

Ahora verás en la consola de R un listado de las google spreadsheets en tu cuenta de gmail.

Ahora, vamos a **escribir** una nueva hoja en tu cuenta.

```
gs_new("misdatos", ws_title = "mihoja", input = head(iris)
      , trim = TRUE, verbose = FALSE)
```

Si vas a tu google drive, deberás ver que se creó un nuevo elemento que se ve así:

| | A | B | C | D | E |
|---|--------------|-------------|--------------|-------------|---------|
| 1 | Sepal.Length | Sepal.Width | Petal.Length | Petal.Width | Species |
| 2 | 5.1 | 3.5 | 1.4 | 0.2 | setosa |
| 3 | 4.9 | 3.0 | 1.4 | 0.2 | setosa |
| 4 | 4.7 | 3.2 | 1.3 | 0.2 | setosa |
| 5 | 4.6 | 3.1 | 1.5 | 0.2 | setosa |
| 6 | 5.0 | 3.6 | 1.4 | 0.2 | setosa |
| 7 | 5.4 | 3.9 | 1.7 | 0.4 | setosa |

Agrega 1000 más filas al final.

+ mihoja

De igual forma, puedes ahora **leer** los datos de cualquier google spreadsheet que tengas en tu cuenta.

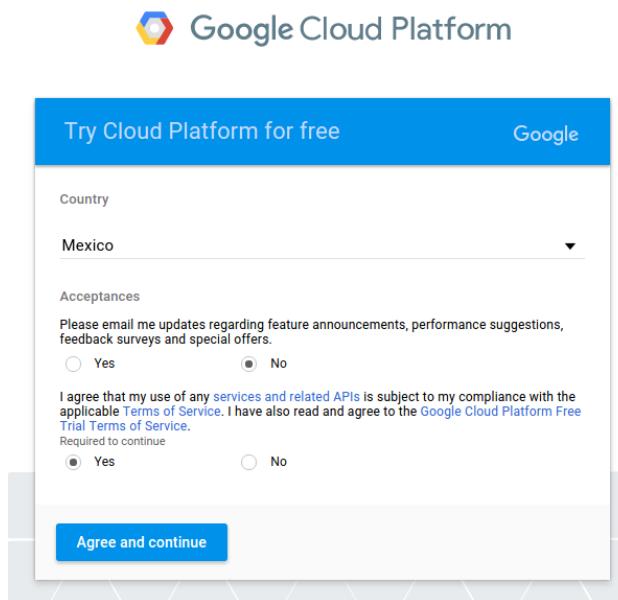
```
misdatos <- gs_read(gs_title("misdatos"), ws = "mihojah")
# La borro
gs_delete(gs_title("misdatos"))
```

Google bigquery

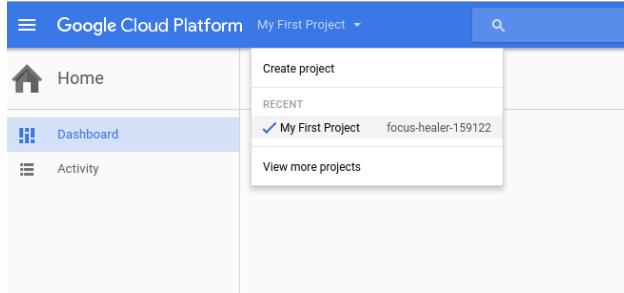
Google bigquery es un *data warehouse* que permite guardar grandes bases de datos. Al contratar el servicio, google se encarga del *hardware* y la infraestructura necesaria para que su procesamiento sea rápido (Platform 2016).

Para guardar tus datos en **bigquery** debes crear un proyecto en la consola de desarrolladores.

Existen varias bases de dato públicas disponibles. Para poder utilizarlas, necesitas tener una cuenta. Puedes empezar una prueba gratis en la página de google cloud platform. Verás una pantalla como esta:



Sigue las instrucciones y eventualmente llegarás a una pantalla como esta



Copia el identificador de tu proyecto para que puedas realizar `queries` (llamadas a las bases de datos).

Leemos la base de datos pública de natalidad en Estados Unidos.

```
project <- "focus-healer-159122" # pon tu projectID aquí

sql <- 'SELECT year, count(*) as babies, avg(mother_age) as mother_age_avg
FROM[publicdata:samples.natality]
WHERE year > 1980 and year < 2006
group by year;'

data <- query_exec(query = sql, project = project)
```

Nota como la tabla cuenta con aproximadamente 140 millones de registros y se obtiene el detalle en segundos.

Heroku Postgres

R no es un manejador de base de datos y, por ende, no es un lenguaje que permite trabajar con una gran cantidad de datos. R guarda los objetos utilizando la memoria virtual de la computadora, i.e. la RAM, misma que depende de varios elementos (incluido el sistema operativo) y que limita los datos que podrán ser procesados.

Cuando necesitamos conocer el tamaño de los datos que están en el ambiente de trabajo, puede utilizarse el paquete `pryr` (Hadley Wickham 2015a, sección “the role of physical memory”).

```
rm(list = ls()) # borramos los objetos del ambiente
# Cargamos datos al ambiente
flights <- read_csv("data/flights.csv")
airports <- read_csv("data/airports.csv")
planes <- read_csv("data/planes.csv")
ls() # mostramos los objetos en el ambiente

library(pryr) # Cargamos el paquete pryr
mem_used() # memoria utilizada
```

```
object_size(flights, units = "Mb") # Obtenemos el tamaño de un objeto
sapply(ls(), function(x) object_size(get(x))) # de todos en el ambiente
```

Las estrategias en memoria se revisaron brevemente en el apartado XXX, en este caso, es pertinente mencionar las estrategias fuera de memoria (*out of memory*).

Es posible explorar un conjunto de datos sin necesidad de cargarlos en R pero utilizando comandos de R y trabajando desde un script de R, permitiendo que herramientas más eficientes (y apropiadas) para el trabajo de grandes volúmenes de datos realicen el procesamiento de los mismos.

Los sistemas gestores de base de datos están optimizados para almacenar y buscar en grandes volúmenes de datos en forma más eficiente que R. Algunos ejemplos populares son Oracle y PostgreSQL (Peng, S Kross y Anderson 2016, sección “working with large datasets”). Hay múltiples paquetes que permiten establecer una conexión con estos sistemas desde una sesión de R.

Los paquetes DBI y Postgresql permiten realizar esta tarea. Debido a que requieren credenciales se muestra una función para leer datos desde PostgreSQL y escribirlos sin necesidad de poner las credenciales dentro del mismo script.

Para que funcionen apropiadamente, es necesario poner en el directorio de trabajo un archivo llamado `parametros.yaml` en donde se escriben las credenciales para Postgres:

```
host : localhost
db : postgres
username : usr
password : password
```

Nota: el salto de línea en la última línea es importante.

Para leer datos, creamos una función a la que podemos enviarle una cadena de comandos en SQL.

```
sql2df <- function(sql.file, df.file = "") {
  require(DBI)
  require(futile.logger)
  require(yaml)
  require(RPostgreSQL)

  if(!file.exists(df.file)) {
    if(file.exists("./parametros.yaml")) {
      x <- yaml::yaml.load_file("./parametros.yaml")
    } else {
      x <- yaml::yaml.load_file("../parametros.yaml")
    }

  # Creamos la conexión a la base de datos
```

```

futile.logger::flog.info("Conectando a la base de datos")

con <- dbConnect(RPostgreSQL::PostgreSQL(), dbname = x$db,
                  host = x$host,
                  port = 5432,
                  user = x$username,
                  password = x$password)

futile.logger::flog.info("Conectado a %s, como %s", x$host, x$username)

# Leemos el query
sql <- paste(readLines(sql.file,encoding="UTF-8")
              , sep=" ", collapse=" ")

tryCatch( {
  futile.logger::flog.info("Ejecutando el query")
  # Creamos el query
  rs <- RPostgreSQL::dbSendQuery(con, sql)
  futile.logger::flog.info("Obteniendo los datos")
  # Obtenemos los datos
  df <- DBI::dbFetch(rs)
  # Liberamos el ResultSet
  futile.logger::flog.info("Limiando el result set")
  RPostgreSQL::dbClearResult(rs)
}, finally=RPostgreSQL::dbDisconnect(con) # Nos desconectamos de la BD
)

if(df.file != ""){
  saveRDS(object=df, file=df.file)
}
} else {
  df <- readRDS(df.file)
}

return(df)
}

```

La función `sql2df`² recibe como parámetro, como cadena, la ruta hacia un archivo de extensión `.sql` con los comandos a ejecutar en el manejador de base de datos. Éste puede verse, por ejemplo, como:

```

select *
from information_schema.tables

```

²Funció adaptada de notas de Adolfo de Únanue.

```
where table_schema = 'information_schema';
```

Guardamos ésta en el archivo `sql/ejemplo.sql` la cláusula de arriba. Después, llamamos a la función.

```
datos <- sql2df("sql/ejemplo.sql", df.file = "ejemplo.rds")
head(datos)
```

Con el parámetro `df.file` es posible especificar una ruta para que se guarde una copia local del resultado de los datos. Esto es útil cuando se está trabajando con los datos, de forma que sea más rápido el trabajo con los mismos.

Para escribir datos, podemos utilizar la función siguiente:

```
df2sql <- function(data.frame, df.schema.name, df.table.name, owner.to = NA) {
  require(DBI)
  require(futile.logger)
  require(yaml)
  require(RPostgreSQL)
  data.frame <- data.frame(data.frame)
  # Normalizamos nombres
  names(data.frame) <- normalizarNombres(names(data.frame))

  if(file.exists("./parametros.yaml")) {
    x <- yaml::yaml.load_file("./parametros.yaml")
  } else {
    x <- yaml::yaml.load_file("../parametros.yaml")
  }

  # Creamos la conexión a la base de datos
  futile.logger::flog.info("Conectando a la base de datos")

  con <- dbConnect(RPostgreSQL::PostgreSQL(), dbname = x$db,
                  host = x$host,
                  port = 5432,
                  user = x$username,
                  password = x$password)
  futile.logger::flog.info("Conectado a %s, como %s"
                           , x$host, x$username)

  tryCatch( {
    flog.info("Ejecutando la escritura de tabla %s en el esquema %s"
              , df.table.name, df.schema.name)
    # Definimos el camino al esquema deseado
    if(df.schema.name != "public"){
      dbSendQuery(conn = con
                  , statement = paste0("SET search_path = "

```

```

        , df.schema.name, ", public"))
}
long.name <- paste0(df.schema.name, ".", df.table.name)
# Escribimos la tabla
dbWriteTable(con,
             df.table.name,
             data.frame,
             overwrite=FALSE,
             append = TRUE)

flog.info("Escribiendo los datos")

if(!is.na(owner.to)){
  flog.info("Otorgando ownership a %s", owner.to)
  dbSendQuery(con, paste0("alter table ", long.name
                          , " owner to ", owner.to, ";"))
}
}, finally=dbDisconnect(con) # Nos desconectamos de la BD
)
flog.info("Escritura finalizada")
}

# Función de ayuda
normalizarNombres <- function(column_names) {
  require(magrittr)
  gsub("\s+", " ", stringr::str_trim(column_names)) %>%
    gsub("^ *|(?=< ) | *$", "", ., perl=T) %>%
    gsub('\\\\ |\\.| ', '_', .) %>%
    gsub("[a-z][A-Z]", "\\\\1_\\\\L\\\\2", ., perl = TRUE) %>%
    gsub('ñ', 'n', .) %>%
    iconv(., to='ASCII//TRANSLIT') %>%
    tolower(.)
}

```

Se especifican en los parámetros el data.frame a escribir, una cadena de caracteres indicando el esquema en el que se escribirá la base, una cadena indicando el nombre de la tabla y es posible especificar qué dueño deberá asignarse para la base:

```
df2sql(iris, "public", "iris", owner.to = "usr")
```

rdata

También es posible guardar objetos específicos del ambiente dentro de un formato especial con extensión `rdata` o `RData`. Esto es muy útil, por ejemplo, para guardar modelos u otros objetos y después poder utilizarlos en producción o en alguna aplicación que requiera un tiempo de respuesta bajo.

Para escribirlos

```
save(...,
  file = "~/misdatos.rdata",
  ascii = FALSE, version = NULL, envir = parent.frame(),
  compress = isTRUE(!ascii), compression_level,
  eval.promises = TRUE, precheck = TRUE)
```

Nota como ... pueden ser uno o más objetos de R.

Para leerlos

```
load("~/misdatos.rdata")
```

Los objetos se cargarán al ambiente con los nombres con los que fueron guardados.

Transformación de datos

Esta sección resume algunas de las funciones existentes para **transformar** datos en R. En particular, se revisan las transformaciones más comunes que se realizan sobre datos. En esta sección, se revisan las acciones implementadas en el paquete `dplyr` (Hadley Wickham y Francois s.f.). En la figura 5.3 podemos ver la etapa del análisis de datos correspondiente.

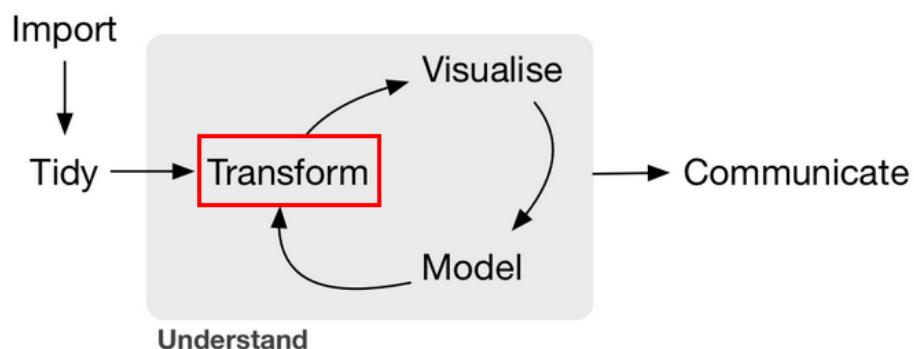


Figura 5.3: Transformación de datos G. Grolemund y H. Wickham (2016, Introducción).

Existen muchas maneras de transformar los datos y una gran cantidad de paquetes que implementan distintas funciones útiles para realizar esta tarea. En particular, resaltamos

`dplyr` y `data.table`. En esta sección se ejemplifican todas las funciones que permiten hacer trabajo con datos que están implementadas en `dplyr`³ y en `tidyR` (Hadley Wickham 2016c).

Tareas comunes en la manipulación de datos

Las funciones implementadas en este paquete están diseñadas para facilitar la transformación de datos. En general, la transformación de datos implica definir qué se hará con ellos, escribir un programa que realice esa tarea y ejecutarlo (Hadley Wickham y Francois s.f., viñeta de introducción).

`dplyr` y `tidyR` simplifican estos pasos al proveer de opciones limitadas consideradas como las tareas más comunes en la transformación de datos. Además, proveen de verbos simples que corresponden a funciones en R y que mapean directamente a estas *tareas más comunes* (ver Tabla 5.2).

³Para iniciar en `data.table` se recomienda ver Srinivasan y Dowle (2016) así como las viñetas del paquete Dowle y col. (2015).

| Acción | Verbos |
|--|---|
| Extracción de subconjuntos de observaciones . | <ul style="list-style-type: none"> ▪ filter: seleccionamos filas de acuerdo a los valores de las variables ▪ distinct: elimina renglones duplicados ▪ sample_frac: selecciona aleatoriamente una fracción de filas ▪ sample_n: selecciona aleatoriamente <i>n</i> filas ▪ slice: selecciona filas por posición ▪ top_n: selecciona y ordena según una variable <i>n</i> entradas |
| Extracción de subconjuntos de variables . | <ul style="list-style-type: none"> ▪ select: seleccionamos un subconjunto de las columnas utilizando los nombres de las variables |
| Creación de resúmenes de datos . | <ul style="list-style-type: none"> ▪ summarise: resume los datos en un valor único ▪ summarise_each: aplica una función de resumen a cada columna ▪ count: cuenta el número de filas con cada valor único de una variable |
| Creación de nuevas variables . | <ul style="list-style-type: none"> ▪ mutate: genera nuevas variables a partir de las variables originales ▪ mutate_each: aplica una función ventana a cada columna ▪ transmute: genera una o más nuevas columnas eliminando las columnas originales |
| Combinación de conjuntos de datos. | <ul style="list-style-type: none"> ▪ left_join: realiza un join conservando todas las observaciones de la primera tabla especificada ▪ right_join: realiza un join conservando todas las observaciones de la segunda tabla especificada ▪ inner_join: realiza un join conservando todas las observaciones que están en ambas tablas ▪ full_join: realiza un join conservando todas las observaciones y valores de ambas tablas ▪ semi_join: conserva todas las observaciones de la primera tabla que están en la segunda tabla ▪ anti_join: conserva todas las observaciones de la primera tabla que no están en la segunda tabla ▪ intersect: conserva observaciones que están tanto en la primera tabla como en la segunda ▪ union: conserva observaciones que están en cualesquier tabla ▪ setdiff: conserva observaciones de la primera tabla que no están en la segunda ▪ bind_rows: une las filas de la segunda tabla a las de la primera ▪ bind_cols: une las columnas de la segunda tabla a las de la primera |
| Agrupar datos. | <ul style="list-style-type: none"> ▪ group_by: agrupa los datos según una o más variables ▪ ungroup: elimina los grupos en un data frame |
| Reorganizar datos (<i>reshape data</i>). | <ul style="list-style-type: none"> ▪ data_frame: combina vectores en un dataframe ▪ arrange: Ordena las filas según una o más variables ▪ rename: renombra columnas de un dataframe |

Cuadro 5.2: Acciones y verbos comunes en la manipulación de datos (**datawrangling**).

Dentro de **tidyR** hay más verbos útiles para reorganizar datos que se verán en la sección 5 junto con los *criterios de datos limpios* que proporcionan un sustento para la conceptualización de la manipulación de datos eficiente en R.

Todos estos verbos funcionan de la misma manera (tienen la misma estructura):

- El primer argumento de la función es un *data.frame*
- Los argumentos subsecuentes indican qué es lo que se debe hacer a ese *data.frame*
- Siempre regresa un *data.frame*

A continuación, se ejemplifica el uso de los distintos verbos de la tabla 5.2. Para esto,

utilizaremos los siguientes conjuntos de datos de muestra, todos disponibles en el paquete `nycflights13` (Hadley Wickham 2016b). Se leerán desde archivo de texto plano para ejemplificar algunos elementos de la limpieza.

Nota como utilizamos la función del paquete `readr read_csv`. Esta es una nueva implementación de `read.csv` pero mucho mas rápida.

```
flights <- read_csv("data/flights.csv")
flights

## # A tibble: 227,496 x 14
##       date   hour minute   dep   arr dep_delay arr_delay
##   <dttm> <int>  <int> <int> <int>    <int>     <int>
## 1 2011-01-01 12:00:00     14      0  1400  1500        0     -10
## 2 2011-01-02 12:00:00     14      1  1401  1501        1     -9
## 3 2011-01-03 12:00:00     13     52  1352  1502       -8     -8
## 4 2011-01-04 12:00:00     14      3  1403  1513        3      3
## 5 2011-01-05 12:00:00     14      5  1405  1507        5     -3
## 6 2011-01-06 12:00:00     13     59  1359  1503       -1     -7
## 7 2011-01-07 12:00:00     13     59  1359  1509       -1     -1
## 8 2011-01-08 12:00:00     13     55  1355  1454       -5    -16
## 9 2011-01-09 12:00:00     14     43  1443  1554       43     44
## 10 2011-01-10 12:00:00    14     43  1443  1553       43     43
## # ... with 227,486 more rows, and 7 more variables: carrier <chr>,
## #   flight <int>, dest <chr>, plane <chr>, cancelled <int>, time <int>,
## #   dist <int>

planes <- read_csv("data/planes.csv")
planes

## # A tibble: 2,853 x 9
##   plane year          mfr      model no.eng no.seats speed
##   <chr> <int> <chr>      <chr>    <int>    <int> <int>
## 1 N576AA 1991 MCDONNELL DOUGLAS DC-9-82(MD-82)     2     172 NA
## 2 N557AA 1993 MARZ BARRY      KITFOX IV      1      2 NA
## 3 N403AA 1974          RAVEN      S55A      NA      1  60
## 4 N492AA 1989 MCDONNELL DOUGLAS DC-9-82(MD-82)     2     172 NA
## 5 N262AA 1985 MCDONNELL DOUGLAS DC-9-82(MD-82)     2     172 NA
## 6 N493AA 1989 MCDONNELL DOUGLAS DC-9-82(MD-82)     2     172 NA
## 7 N477AA 1988 MCDONNELL DOUGLAS DC-9-82(MD-82)     2     172 NA
## 8 N476AA 1988 MCDONNELL DOUGLAS DC-9-82(MD-82)     2     172 NA
## 9 N504AA    NA AUTHIER ANTHONY P      TIERRA II     1      2 NA
## 10 N565AA 1987 MCDONNELL DOUGLAS DC-9-83(MD-83)    2     172 NA
## # ... with 2,843 more rows, and 2 more variables: engine <chr>, type <chr>

airports <- read_csv("data/airports.csv")
airports
```

```
## # A tibble: 3,376 x 7
##   iata      airport      city state country     lat
##   <chr>     <chr>      <chr> <chr> <chr>    <dbl>
## 1 00M      Thigpen     Bay Springs MS USA 31.95376
## 2 00R Livingston Municipal Livingston TX USA 30.68586
## 3 00V      Meadow Lake Colorado Springs CO USA 38.94575
## 4 01G      Perry-Warsaw Perry NY USA 42.74135
## 5 01J      Hilliard Airpark Hilliard FL USA 30.68801
## 6 01M      Tishomingo County Belmont MS USA 34.49167
## 7 02A      Gragg-Wade Clanton AL USA 32.85049
## 8 02C      Capitol Brookfield WI USA 43.08751
## 9 02G      Columbiana County East Liverpool OH USA 40.67331
## 10 03D     Memphis Memorial Memphis MO USA 40.44726
## # ... with 3,366 more rows, and 1 more variables: long <dbl>
```

Extracción de subconjuntos de observaciones

filter

Ya habíamos visto muchas maneras de extraer datos específicos de una base de datos de acuerdo a condiciones lógicas impuestas en los valores de las filas de columnas específicas. `filter` nos permite poner tantas condiciones como queramos de manera muy fácil y entendible por cualquiera que lea nuestro código.

Busquemos, por ejemplo, todos los vuelos hacia SFO u OAK

```
filter(flights, dest == "SFO" | dest == "OAK")
```

```
## # A tibble: 3,508 x 14
##   date      hour minute   dep   arr dep_delay arr_delay
##   <dttm> <int> <int> <int> <int>    <int>    <int>
## 1 2011-01-31 12:00:00     8     51    851   1052      1    -27
## 2 2011-01-31 12:00:00    11     29   1129   1351      4      1
## 3 2011-01-31 12:00:00    14     32   1432   1656      7      5
## 4 2011-01-31 12:00:00    17     48   1748   2001      3    -4
## 5 2011-01-31 12:00:00    21     43   2143   2338     50     24
## 6 2011-01-31 12:00:00     7     29    729   1002     -1      2
## 7 2011-01-31 12:00:00    15     58   1558   1812     -2    -8
## 8 2011-01-30 12:00:00     9     35    935   1203     45     49
## 9 2011-01-30 12:00:00    11     43   1143   1359     18     14
## 10 2011-01-30 12:00:00    14     59   1459   1715     34     24
## # ... with 3,498 more rows, and 7 more variables: carrier <chr>,
## #   flight <int>, dest <chr>, plane <chr>, cancelled <int>, time <int>,
## #   dist <int>
```

Los vuelos con retraso mayor a 5 horas

```
filter(flights, arr_delay > 5)
```

```
## # A tibble: 77,848 x 14
##   date       hour minute dep   arr dep_delay arr_delay
##   <dttm>     <int>  <int> <int> <int>    <int>      <int>
## 1 2011-01-09 12:00:00     14     43  1443  1554        43      44
## 2 2011-01-10 12:00:00     14     43  1443  1553        43      43
## 3 2011-01-11 12:00:00     14     29  1429  1539        29      29
## 4 2011-01-17 12:00:00     15     30  1530  1634       90      84
## 5 2011-01-20 12:00:00     15      7  1507  1622       67      72
## 6 2011-01-31 12:00:00     14     41  1441  1553       41      43
## 7 2011-01-13 12:00:00      7     22   722   841        2       6
## 8 2011-01-16 12:00:00      7     43   743   843       23      8
## 9 2011-01-17 12:00:00      7     24   724   842        4       7
## 10 2011-01-24 12:00:00     7     31   731   904       11      29
## # ... with 77,838 more rows, and 7 more variables: carrier <chr>,
## #   flight <int>, dest <chr>, plane <chr>, cancelled <int>, time <int>,
## #   dist <int>
```

Podemos juntar las preguntas: vuelos con retraso mayor a 5 horas con destino a SFO o OAK

```
filter(flights, dest == "SFO" | dest == "OAK", arr_delay > 5)
```

```
## # A tibble: 1,581 x 14
##   date       hour minute dep   arr dep_delay arr_delay
##   <dttm>     <int>  <int> <int> <int>    <int>      <int>
## 1 2011-01-31 12:00:00     21     43  2143  2338       50      24
## 2 2011-01-30 12:00:00      9     35   935  1203       45      49
## 3 2011-01-30 12:00:00     11     43  1143  1359       18      14
## 4 2011-01-30 12:00:00     14     59  1459  1715       34      24
## 5 2011-01-30 12:00:00     17     49  1749  2011        4       6
## 6 2011-01-30 12:00:00     19     31  1931  2159       41      41
## 7 2011-01-30 12:00:00     21      0  2100  2320       10      9
## 8 2011-01-29 12:00:00      8     52   852  1126        2      12
## 9 2011-01-29 12:00:00     14     42  1442  1655       17      9
## 10 2011-01-29 12:00:00    18      9  1809  2021       14      11
## # ... with 1,571 more rows, and 7 more variables: carrier <chr>,
## #   flight <int>, dest <chr>, plane <chr>, cancelled <int>, time <int>,
## #   dist <int>
```

distinct

Para eliminar duplicados, usamos la función `distinct`

```

flights.dup <- bind_rows(flights, flights)
dim(flights.dup)

## [1] 454992      14

dim(distinct(flights.dup))

## [1] 227496      14

rm(flights.dup)

```

sample_n, sample_frac, slice

En ocasiones necesitamos extraer subconjuntos aleatorios de los datos, para ello podemos especificar el número de renglones que necesitamos (usando `sample_n`), el porcentaje de datos que deseamos (usando `sample_frac`) o las posiciones específicas de los datos que queremos (usando `slice`).

```

set.seed(1099)
# extraemos 10 datos en forma aleatoria
sample_n(flights, size = 10)

## # A tibble: 10 x 14
##       date   hour minute   dep   arr dep_delay arr_delay
##   <dttm> <int>  <int> <int> <int>    <int>     <int>
## 1 2011-05-29 12:00:00     12      3   1203   1304        -2      -16
## 2 2011-05-30 12:00:00     13     43   1343   1432        -2       -8
## 3 2011-07-20 12:00:00     18     21   1821   1956        10      20
## 4 2011-07-10 12:00:00     13      5   1305   1355         5       0
## 5 2011-06-17 12:00:00     11     30   1130   1228         5       3
## 6 2011-12-03 12:00:00     7      59    759   851        -1      -9
## 7 2011-06-12 12:00:00     9      19    919   1015         4       4
## 8 2011-09-24 12:00:00     7      55    755   1124        -5      -5
## 9 2011-04-19 12:00:00    17      48   1748      6        18      NA
## 10 2011-10-10 12:00:00    10      26   1026   1406         1      -1
## # ... with 7 more variables: carrier <chr>, flight <int>, dest <chr>,
## #   plane <chr>, cancelled <int>, time <int>, dist <int>

# Extraemos el 1% de los datos de flights
sample_frac(flights, size = 0.01)

## # A tibble: 2,275 x 14
##       date   hour minute   dep   arr dep_delay arr_delay
##   <dttm> <int>  <int> <int> <int>    <int>     <int>
## 1 2011-03-21 12:00:00     13      21   1321   1635         6       0
## 2 2011-06-17 12:00:00     21      2   2102   2151         2      11

```

```

## 3 2011-07-12 12:00:00    10    17 1017 1448      2      1
## 4 2011-08-08 12:00:00    10    55 1055 1422      0     -3
## 5 2011-09-13 12:00:00    19     1 1901 2314     -4     -5
## 6 2011-03-06 12:00:00    18    32 1832 2044     -3    -11
## 7 2011-05-09 12:00:00     6    53  653  843      18    -15
## 8 2011-10-06 12:00:00    15    33 1533 1628      3     -7
## 9 2011-06-17 12:00:00    13    50 1350 1545      0      2
## 10 2011-11-01 12:00:00   14    30 1430 1601      5      1
## # ... with 2,265 more rows, and 7 more variables: carrier <chr>,
## #   flight <int>, dest <chr>, plane <chr>, cancelled <int>, time <int>,
## #   dist <int>

# extraemos las posiciones 100 a 110
slice(flights, 100:110)

## # A tibble: 11 x 14
##       date   hour minute   dep   arr dep_delay arr_delay
##   <dttm> <int>  <int> <int> <int> <int>      <int>      <int>
## 1 2011-01-12 12:00:00    16    31 1631 1739      1     -6
## 2 2011-01-13 12:00:00    16    30 1630 1733      0    -12
## 3 2011-01-14 12:00:00    16    29 1629 1734     -1    -11
## 4 2011-01-15 12:00:00    16    32 1632 1736      2     -9
## 5 2011-01-16 12:00:00    17     8 1708 1819     38     34
## 6 2011-01-17 12:00:00    16    32 1632 1744      2     -1
## 7 2011-01-18 12:00:00    16    25 1625 1740     -5     -5
## 8 2011-01-19 12:00:00    16    29 1629 1731     -1    -14
## 9 2011-01-20 12:00:00    16    41 1641 1752     11      7
## 10 2011-01-21 12:00:00   16    38 1638 1746      8      1
## 11 2011-01-22 12:00:00   16    23 1623 1742     -7     -3
## # ... with 7 more variables: carrier <chr>, flight <int>, dest <chr>,
## #   plane <chr>, cancelled <int>, time <int>, dist <int>

```

top_n

Podemos obtener los 5 vuelos con mayor retraso de salida:

```

top_n(flights, 5, dep_delay)

## # A tibble: 5 x 14
##       date   hour minute   dep   arr dep_delay arr_delay carrier
##   <dttm> <int>  <int> <int> <int> <int>      <int>      <int>  <chr>
## 1 2011-06-21 12:00:00    23    34 2334 124      869     861  UA
## 2 2011-06-09 12:00:00    20    29 2029 2243     814     793  MQ
## 3 2011-08-01 12:00:00     1    56 156   452     981     957  CO
## 4 2011-11-08 12:00:00     7    21  721   948     931     918  MQ

```

```
## 5 2011-12-12 12:00:00      6      50    650    808      970      978      AA
## # ... with 6 more variables: flight <int>, dest <chr>, plane <chr>,
## #   cancelled <int>, time <int>, dist <int>
```

O con el menor retraso de salida:

```
top_n(flights, 5, desc(dep_delay))
```

```
## # A tibble: 6 x 14
##       date hour minute dep arr dep_delay arr_delay carrier
##   <dttm> <int> <int> <int> <int>     <int>     <int> <chr>
## 1 2011-01-18    15     42   1542   1936      -18      -17   CO
## 2 2011-02-14    19     17   1917   2027      -23      -23   MQ
## 3 2011-04-10    21      1   2101   2206      -19      -12   XE
## 4 2011-08-03    17     41   1741   1810      -19      -40   XE
## 5 2011-10-04    14     38   1438   1813      -18      -31   EV
## 6 2011-12-24    11     12   1112   1314      -33      -25   OO
## # ... with 6 more variables: flight <int>, dest <chr>, plane <chr>,
## #   cancelled <int>, time <int>, dist <int>
```

Extracción de subconjuntos de variables (*select*)

Podemos ahora, mas fácilmente, quedarnos con únicamente ciertas variables. *select* esta implementado de tal manera que funciona *nombrando* las variables que se quieren utilizar.

```
select(flights, flight, dest)
```

```
## # A tibble: 227,496 x 2
##   flight dest
##   <int> <chr>
## 1     428 DFW
## 2     428 DFW
## 3     428 DFW
## 4     428 DFW
## 5     428 DFW
## 6     428 DFW
## 7     428 DFW
## 8     428 DFW
## 9     428 DFW
## 10    428 DFW
## # ... with 227,486 more rows
```

También podemos especificar que queremos todas las variables *menos* algunas.

```
select(flights, -date, -hour, -minute, -dep, -arr, -carrier, -flight)
```

```
## # A tibble: 227,496 x 7
```

```
##   dep_delay arr_delay dest plane cancelled time dist
##   <int>      <int> <chr>  <chr>      <int> <int> <int>
## 1      0      -10 DFW N576AA       0    40  224
## 2      1       -9 DFW N557AA       0    45  224
## 3     -8      -8 DFW N541AA       0    48  224
## 4      3       3 DFW N403AA       0    39  224
## 5      5      -3 DFW N492AA       0    44  224
## 6     -1      -7 DFW N262AA       0    45  224
## 7     -1      -1 DFW N493AA       0    43  224
## 8     -5     -16 DFW N477AA       0    40  224
## 9     43      44 DFW N476AA       0    41  224
## 10    43      43 DFW N504AA       0    45  224
## # ... with 227,486 more rows
```

Podemos pedir las variables que empiezan con algún carácter.

```
select(flights, starts_with("d"))
```

```
## # A tibble: 227,496 x 5
##       date   dep dep_delay dest dist
##   <dttm> <int>      <int> <chr> <int>
## 1 2011-01-01 12:00:00 1400       0 DFW  224
## 2 2011-01-02 12:00:00 1401       1 DFW  224
## 3 2011-01-03 12:00:00 1352      -8 DFW  224
## 4 2011-01-04 12:00:00 1403       3 DFW  224
## 5 2011-01-05 12:00:00 1405       5 DFW  224
## 6 2011-01-06 12:00:00 1359      -1 DFW  224
## 7 2011-01-07 12:00:00 1359      -1 DFW  224
## 8 2011-01-08 12:00:00 1355      -5 DFW  224
## 9 2011-01-09 12:00:00 1443      43 DFW  224
## 10 2011-01-10 12:00:00 1443      43 DFW  224
## # ... with 227,486 more rows
```

O las que contienen algún patrón

```
select(flights, contains("dep"))
```

```
## # A tibble: 227,496 x 2
##       dep dep_delay
##   <int>      <int>
## 1 1400       0
## 2 1401       1
## 3 1352      -8
## 4 1403       3
## 5 1405       5
## 6 1359      -1
## 7 1359      -1
```

```
## 8 1355      -5
## 9 1443      43
## 10 1443     43
## # ... with 227,486 more rows
```



Ejercicios

1. Revisa la ayuda de select con `?select`.
2. Juega con la función `starts_with()`. ¿Qué variables empiezan con “de”?
3. Juega con la función `ends_with()`. ¿Qué variables terminan con “delay”?
4. Utiliza la base de datos `iris` como en el ejemplo de la ayuda.
5. ¿Qué hace la función `contains()`?
6. ¿Cómo es diferente de `matches()`?
7. ¿Cómo obtienes todas las variables *menos* Petal.Width?

Creación de resumenes de datos

Las funciones `summarise`, `summarise_each` y `count` permiten realizar resumenes para ciertas variables existentes o nuevas en los datos.

`summarise`

Ahora, si queremos saber el promedio de velocidad de los vuelos podemos calcularlo fácilmente con `summarise`.

```
flights$velocidad <- flights$dist/flights$time

summarise(flights, vel_prom = mean(velocidad, na.rm = T))
```

```
## # A tibble: 1 x 1
##   vel_prom
##   <dbl>
## 1 7.017055
```

Funciones resumen

Pueden utilizarse junto con `summarise` cualesquiera función en R (por ejemplo: `min`, `max`, `mean`, `median`, `var`, `sd`) que realice agregaciones de vectores. Sin embargo, el paquete `dplyr` implementa varias funciones útiles adicionales como (**datawrangling**):

- `first`: extrae el primer valor de un vector
- `last`: extrae el último valor de un vector
- `n`: cuenta el número de valores en un vector
- `n_distinct` cuenta el número de valores único en un vector

summarise_each

Podemos especificar una función a aplicar a variables específicas en un dataframe. Por ejemplo, extraer la media para las variables: date, dep_delay, arr_delay, time y dist.

```
summarise_each(flights, funs(mean), date, dep_delay, arr_delay, time, dist)
```

```
## # A tibble: 1 x 5
##       date dep_delay arr_delay   time     dist
##   <dttm>     <dbl>     <dbl> <dbl>     <dbl>
## 1 2011-07-02 03:05:12      NA      NA 787.7832
```

Debido a que existen valores perdidos en variables como retraso de salida (*dep_delay*) y retrasos de llegada (*arr_delay*), debemos especificar la opción para no utilizar los NAs en la función.

```
summarise_each(flights, funs(mean(., na.rm = T)), date, dep_delay,
               arr_delay, time, dist)
```

```
## # A tibble: 1 x 5
##       date dep_delay arr_delay   time     dist
##   <dttm>     <dbl>     <dbl> <dbl>     <dbl>
## 1 2011-07-02 03:05:12  9.444951  7.094334 108.1423 787.7832

# opcion 2
mean_na <- function(x){
  mean(x, na.rm = T)
}
summarise_each(flights, funs(mean_na), date, dep_delay, arr_delay, time, dist)

## # A tibble: 1 x 5
##       date dep_delay arr_delay   time     dist
##   <dttm>     <dbl>     <dbl> <dbl>     <dbl>
## 1 2011-07-02 03:05:12  9.444951  7.094334 108.1423 787.7832
```

count

Podemos contar los valores únicos en variables categóricas, por ejemplo, contar el número de vuelos por aerolínea:

```
count(flights, carrier, sort = T)
```

```
## # A tibble: 15 x 2
##   carrier     n
##   <chr>    <int>
## 1 XE        73053
## 2 CO        70032
```

```
## 3      WN 45343
## 4      OO 16061
## 5      MQ 4648
## 6      US 4082
## 7      AA 3244
## 8      DL 2641
## 9      EV 2204
## 10     FL 2139
## 11     UA 2072
## 12     F9 838
## 13     B6 695
## 14     AS 365
## 15     YV 79
```

Creación de nuevas variables

mutate y **transmute**

Muchas veces lo que se desea es generar nuevas variables utilizando funciones sobre las variables de la tabla.

Por ejemplo, queremos saber cual fue el vuelo mas rápido. Para esto queremos calcular la velocidad promedio del vuelo.

```
select(arrange(mutate(flights, velocidad = dist/time), desc(velocidad)),
       flight, dest, velocidad)

## # A tibble: 227,496 x 3
##   flight  dest velocidad
##   <int> <chr>     <dbl>
## 1 1646   AUS    12.72727
## 2 5229   MEM    11.16667
## 3 944    CLT    10.74118
## 4 4634   HOB    10.65957
## 5 500    IND    10.30488
## 6 106    EWR    10.14493
## 7 644    CLE    10.10185
## 8 1074   CLE    10.10185
## 9 1054   EWR    10.07194
## 10 1424   DCA   10.06667
## # ... with 227,486 more rows
```

Esta manera de transformar a los datos (utilizando varios de los verbos) es confusa y difícil de leer. Es mas sencillo utilizar el operador pipe⁴ de R implementado en el paquete **magrittr**,

⁴Para una explicación más detallada de la importancia de este operador en la simplificación de código, ver

es decir, %>% (Bache y Hadley Wickham 2014).

```
flights2 <- mutate(flights, velocidad = dist/time) %>%
  arrange(., desc(velocidad)) %>%
  select(., flight, dest, velocidad)
flights2

## # A tibble: 227,496 x 3
##   flight  dest velocidad
##   <int> <chr>     <dbl>
## 1 1646   AUS    12.72727
## 2 5229   MEM    11.16667
## 3 944    CLT    10.74118
## 4 4634   HOB    10.65957
## 5 500    IND    10.30488
## 6 106    EWR    10.14493
## 7 644    CLE    10.10185
## 8 1074   CLE    10.10185
## 9 1054   EWR    10.07194
## 10 1424   DCA   10.06667
## # ... with 227,486 more rows
```

La lectura es mucho mas sencilla de esta forma pues se lee de manera secuencial las transformaciones que se están realizando a los datos:

1. Agrego la columna de velocidad a la base de datos de flights, luego (operador pipe)
2. Ordeno los vuelos en forma descendiente según su velocidad, luego
3. Seleccióno las variables de vuelo, destino y velocidad.

Que el código sea entendible no es importante únicamente en el contexto del trabajo colaborativo pues muchas veces los lectores de su código serán ustedes en el futuro.

`transmute` es muy similar a `mutate` pero elimina las variables que no fueron creadas por la función.

```
dplyr::transmute(flights, velocidad = dist/time) %>%
  arrange(., desc(velocidad))

## # A tibble: 227,496 x 1
##   velocidad
##       <dbl>
## 1 12.72727
## 2 11.16667
## 3 10.74118
## 4 10.65957
## 5 10.30488
## 6 10.14493
```

la nota del autor del paquete Stefan Milton en Galili (2014) y la viñeta `magrittr` dentro del paquete.

```
## 7 10.10185
## 8 10.10185
## 9 10.07194
## 10 10.06667
## # ... with 227,486 more rows
```

Ejercicios

1. ¿Cuáles son los 10 aviones-aerolíneas mas lentos? Utiliza el operador pipe, mutate, arrange y head.
2. Utiliza la función `str_sub` dentro de `stringr` para extraer únicamente el día del campo `date`.
3. Utiliza la función `ymd` del paquete `lubridate` para declarar date como una fecha (¡otra clase!).
4. Utiliza las funciones `paste0` del `base` y `ymd_hm` de `lubridate` para declarar date como un `datetime`.

Respuestas

```
#1
mutate(flights, velocidad = dist/time) %>%
  arrange(velocidad) %>%
  head(10) %>%
  select(plane, carrier, velocidad)

# Más lindo, usando group_by y top_n: más específicamente el más
# lento por carrier
mutate(flights, velocidad = dist/time) %>%
  group_by(carrier) %>%
  arrange(velocidad) %>%
  top_n(1) %>%
  select(plane, carrier, velocidad)

#2
mutate(flights, dia = stringr::str_sub(date, 9, 10)) %>%
  select(date, dia)
head(flights)

# 3
mutate(flights,
       fecha = stringr::str_sub(date, 1, 10)
     , fecha = lubridate::ymd(fecha)) %>%
  select(date, fecha)

# 4
mutate(flights,
       fecha = lubridate::ymd_hms(date)) %>%
  select(date, fecha)
```

Funciones ventana *window functions*

Dentro de mutate, se pueden utilizar otras funciones que realicen transformaciones sobre vectores que regresen un vector del mismo tamaño, así como funciones propias.

Ahora bien, dplyr implementa otras funciones ventana como (**datawrangling**):

- **lead**: regresa todos los valores del vector movidos por una posición posterior
- **lag**: regresa todos los valores del vector movidos por una posición anterior
- **dense_rank**: rango sin huecos
- **min_rank**: rango especificando el criterio de mínimo para empates
- **percent_rank**: rango reescalado para estar entre 0 y 1
- **row_number**: número de fila
- **ntile**: creación de n percentiles
- **between**: verifica si el valor está entre dos valores
- **cume_dist**: distribución acumulada
- **cumall**: para vectores lógicos, intersección de los valores al renglón i-ésimo
- **cumany**: para vectores lógicos, unión de los valores al renglón i-ésimo
- **cummean**: acumula la media

Para mayor detalle, puede revisarse la viñeta “window-functions” en el paquete dplyr (Hadley Wickham y Francois s.f.) con el comando `vignette("window-functions", package = "dplyr")`

mutate_each

Igual que con `summarise_each`, `mutate_each` permite especificar una transformación vía una función ventana para variables específicas. Por ejemplo, extraemos los deciles correspondientes para las variables tiempo (*time*) y distancia (*dist*).

```
flights.m <- mutate_each(flights, funs(ntile(., 10)), time, dist)
table(flights.m$time)
```

```
##  
##      1      2      3      4      5      6      7      8      9      10  
## 22388 22387 22388 22387 22387 22388 22387 22388 22387 22387
```

Combinación de conjuntos de datos (*joins*)

Muchas veces la información se tiene repartida entre diferentes tablas pero es necesario juntar las variables de las diferentes observaciones en una sola tabla para modelarlas o describirlas. Es muy estándar, en el lenguaje SQL, el tipo de joins que se pueden utilizar. La figura 5.4 muestra un resumen del tipo de joins que pueden realizarse.

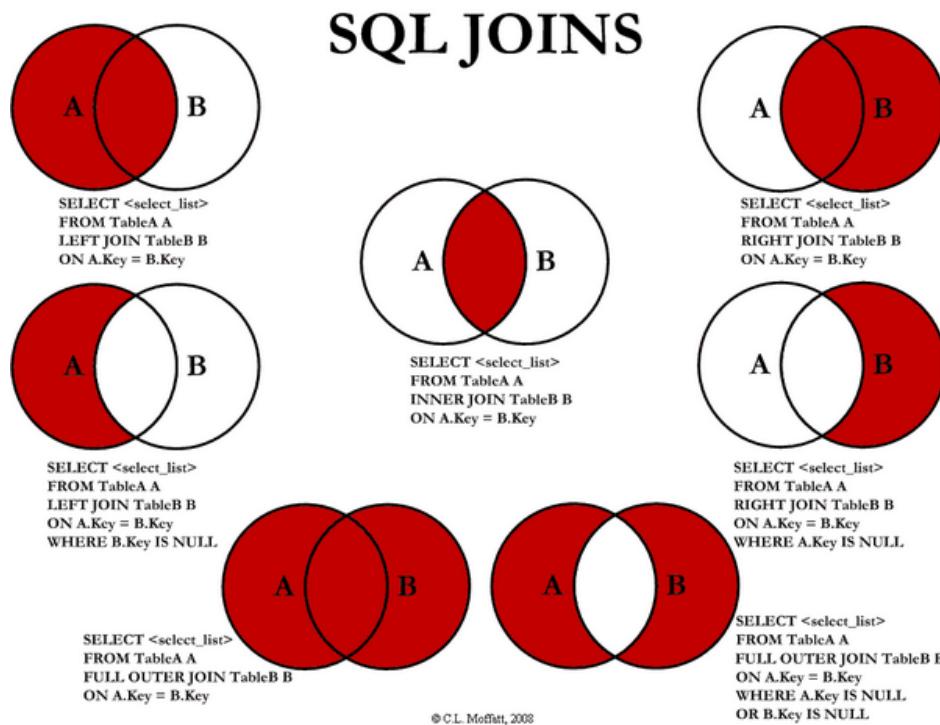


Figura 5.4: Joins en el lenguaje SQL (Moffat 2009).

El paquete `dplyr` implementa estos joins de manera natural, utilizando la lógica de SQL.

- `inner_join`: regresa todas las filas de x en donde hay valores correspondientes para y, junto con todas las columnas.
- `left_join`: regresa todas las filas de x, rellenando con NA para valores que no encontró en y.
- `right_join`: regresa todas las filas de y, rellenando con NA para valores que no encontró en x.
- `full_join`: regresa todas las filas y todas las columnas para x y y. Donde no hay valores en alguno de los dos, rellena con NA.
- `semi_join`: regresa todas las filas de x para las que hay valores en y regresando únicamente las columnas de x.
- `anti_join`: regresa todas las filas de x donde no hay valores en y, manteniendo solo las columnas de x.

Ahora, supongamos que queremos saber la velocidad promedio de los aviones que tenemos en nuestros datos para todos sus vuelos.

```
# base de aviones con velocidad

vel_aviones <- flights %>%
  group_by(plane) %>%
  summarise(vel_prom = mean(dist/time, na.rm = T))
```

```
inner_join(
  planes,
  vel_aviones
) %>%
  select(plane, year, vel_prom) %>%
  arrange(desc(vel_prom))
```

```
## # A tibble: 2,853 x 3
##   plane   year vel_prom
##   <chr> <int>    <dbl>
## 1 N653JB  2007  9.333333
## 2 N709UW  1999  9.316327
## 3 N3744F  2001  9.070175
## 4 N3769L  2002  9.065789
## 5 N623JB  2005  9.037975
## 6 N607JB  2005  8.936351
## 7 N580JB  2003  8.869907
## 8 N658JB  2007  8.869565
## 9 N589JB  2004  8.814815
## 10 N760JB 2008  8.788441
## # ... with 2,843 more rows
```

Ahora, queremos saber los destinos con mayores retrasos.

```
destinos <- flights %>%
  group_by(dest) %>%
  summarise(retraso = mean(arr_delay, na.rm = T))
```

```
inner_join(
  airports,
  destinos,
  by = c("iata" = "dest")
) %>%
  arrange(desc(retraso))
```

```
## # A tibble: 114 x 8
##   iata                      airport          city state
##   <chr>                     <chr>           <chr> <chr>
## 1 ANC Ted Stevens Anchorage International Anchorage AK
## 2 CID Eastern Iowa Cedar Rapids IA
## 3 DSM Des Moines International Des Moines IA
## 4 SFO San Francisco International San Francisco CA
## 5 BPT Southeast Texas Regional Beaumont/Port Arthur TX
## 6 GRR Kent County International Grand Rapids MI
## 7 DAY James M Cox Dayton Intl Dayton OH
## 8 VPS Eglin Air Force Base Valparaiso FL
```

```

## 9   SAV           Savannah International          Savannah    GA
## 10  RIC           Richmond International       Richmond    VA
## # ... with 104 more rows, and 4 more variables: country <chr>, lat <dbl>,
## #   long <dbl>, retraso <dbl>

```

Agrupar datos

group_by

Un verbo muy poderoso es el `group_by`. Es por este tipo de verbos que resultó necesario generar un objeto diferente al tradicional `dataframe`. Básicamente, el `tbl_df` es capaz de guardar agrupamientos específicos sobre los cuáles puede hacer funciones ventana (*window functions*) o resúmenes sobre los grupos que se definen.

Ejemplo

Los datos de vuelos se agrupan naturalmente sobre las aerolíneas.

```

flights.g <- flights %>% group_by(., carrier)
flights.g

```

```

## Source: local data frame [227,496 x 15]
## Groups: carrier [15]
##
## # A tibble: 227,496 x 15
##       date   hour minute   dep   arr dep_delay arr_delay
##       <dttm> <int> <int> <int> <int>     <int>     <int>
## 1 2011-01-01 12:00:00     14      0 1400  1500        0     -10
## 2 2011-01-02 12:00:00     14      1 1401  1501        1     -9
## 3 2011-01-03 12:00:00     13     52 1352  1502       -8     -8
## 4 2011-01-04 12:00:00     14      3 1403  1513        3      3
## 5 2011-01-05 12:00:00     14      5 1405  1507        5     -3
## 6 2011-01-06 12:00:00     13     59 1359  1503       -1     -7
## 7 2011-01-07 12:00:00     13     59 1359  1509       -1     -1
## 8 2011-01-08 12:00:00     13     55 1355  1454       -5    -16
## 9 2011-01-09 12:00:00     14     43 1443  1554       43     44
## 10 2011-01-10 12:00:00    14     43 1443  1553       43     43
## # ... with 227,486 more rows, and 8 more variables: carrier <chr>,
## #   flight <int>, dest <chr>, plane <chr>, cancelled <int>, time <int>,
## #   dist <int>, velocidad <dbl>

```



Ejercicios

1. Busca en la ayuda la función `top_n`.
2. Utilízala, junto con `group_by`, `arrange` y `summarise` para extraer los 10 aviones por aerolínea con el menor promedio de velocidad.
3. ¿Cuáles son los aeropuertos con mayor promedio de retraso en la salida?
4. ¿Cuáles son los aeropuertos con mayor promedio de retraso en las llegadas?

```
# Respuestas
# 1
?top_n
# 2
flights %>%
  mutate(velocidad = dist/time) %>%
  group_by(plane, carrier) %>%
  summarise(vel_prom = mean(velocidad, na.rm = T)) %>%
  ungroup() %>%
  group_by(carrier) %>%
  arrange(vel_prom) %>%
  top_n(1)

# 3
flights %>%
  group_by(dest) %>%
  summarise(arr_delay = mean(arr_delay, na.rm = T)) %>%
  arrange(desc(arr_delay)) %>%
  head(10)
```

Afectación de otros verbos por `group_by`

El establecer grupos puede conjugarse con los demás verbos implementados en el paquete `dplyr` y los afecta de diferente manera (Hadley Wickham y Francois s.f., viñeta de introducción):

- `select` realiza la misma operación pero retiene siempre las variables de agrupación.
- `arrange` realiza el arreglo según las variables especificadas pero ordena primero según los grupos.
- `mutate` y `filter` realizan las operación dentro de los grupos definidos y son más útiles cuando se utilizan en conjunción a las funciones ventana.
- `sample_n` y `sample_frac` extraen el número o fracción de observaciones según el número o fracción de filas en cada grupo.
- `summarise` en conjunción con `group_by` llevan acabo el paradigma split (por las variables definidas en el `group_by`), apply (por las funciones especificadas en el `summarise`) y lo combinan en un dataframe.

ungroup

La función `group_by` agrega la clase `grouped_df`, así como atributos al objeto.

```
g.df <- group_by(flights, plane, carrier)
class(g.df)

## [1] "grouped_df" "tbl_df"      "tbl"        "data.frame"
names(attributes(g.df))

## [1] "names"          "row.names"     "spec"
## [4] "class"          "vars"         "drop"
## [7] "indices"        "group_sizes"   "biggest_group_size"
## [10] "labels"

class(flights)

## [1] "tbl_df"       "tbl"        "data.frame"
names(attributes(flights))

## [1] "names"          "row.names"     "spec"        "class"
```

La función `ungroup` los elimina:

```
class(ungroup(g.df))

## [1] "tbl_df"       "tbl"        "data.frame"
names(attributes(ungroup(g.df)))

## [1] "class"        "names"       "row.names"    "spec"
```

Ejercicios

1. ¿Cuáles son los aeropuertos que SI están en la base de destinos?
2. ¿Cuáles son los aeropuertos que NO están en la base de destinos?
3. Realiza un resumen de los vuelos por aeropuerto para varias variables con la función `summarise`.
4. Utiliza la función `inner_join` para pegar la tablas de resumen creada en el paso anterior a la tabla `airports`
5. Realiza un resumen de los vuelos por avión para varias variables con la función `summarise`.
6. Utiliza la función `left_join` para pegar la tablas de resumen creada en el paso anterior a la tabla `planes`
7. Extrae el porcentaje de vuelos cancelados por aerolínea cada año y el porcentaje de vuelos retrasados por aerolínea cada año.

Respuestas

```

# 1
semi_join(airports, flights, by = c("iata" = "dest"))
# 2
anti_join(airports, flights, by = c("iata" = "dest"))
# 3
res.vuelos <- flights %>%
  group_by(dest) %>%
  summarise(
    flights = n()
    , planes = n_distinct(plane)
    , carriers = n_distinct(carrier)
    , cancelled_flights = sum(cancelled, na.rm = T)
    , dep_delay_mean = mean(dep_delay, na.rm = T)
  )
# 4
airports_flights <- inner_join(airports, res.vuelos
                                , by = c("iata" = "dest"))
# 5
res.aviones <- flights %>%
  group_by(plane) %>%
  summarise(
    flights = n()
    , carriers = n_distinct(carrier)
    , cancelled_flights = sum(cancelled, na.rm = T)
    , dep_delay_mean = mean(dep_delay, na.rm = T)
    , arr_delay_mean = mean(arr_delay, na.rm = T)
    , dist_mean = mean(dist, na.rm = T)
    , vel_mean = mean(dist/time, na.rm = T)
  )
# 6
planes_flights <- left_join(planes, res.aviones)
# 7
flights %>%
  mutate(anio = lubridate::year(date)) %>%
  group_by(carrier, anio) %>%
  summarise(
    vuelos.anuales = n()
    , cancelados = sum(cancelled, na.rm = T)/vuelos.anuales * 100
    , retrasados = sum(dep_delay > 0, na.rm = T)/vuelos.anuales * 100
  ) %>%
  ungroup()

```

Reorganizar datos

data_frame

Al igual que la función `data.frame`, esta función permite generar un data frame a partir de vectores:

```
df <- data_frame(
  x = rnorm(100)
  , y = runif(100)
)
class(df)

## [1] "tbl_df"     "tbl"        "data.frame"

df <- data.frame(
  x = rnorm(100)
  , y = runif(100)
)
class(df)

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

La clase del objeto generado por la función `data_frame` es un `tibble` mientras que la función del base `data.frame` es de clase `data.frame`.

arrange

`order` habíamos visto que es la implementación del base para ordenar vectores o en su defecto, dataframes de acuerdo a valores de vectores en esta. Sin embargo, es engorrosa la manera de llamarlo.

Podemos arreglar los valores de las tablas, fácilmente con `arrange`. Por ejemplo, podemos ver los 5 vuelos con mayor retraso de llegada.

```
head(arrange(flights, desc(arr_delay)), n=5)

## # A tibble: 5 x 15
##       date   hour minute   dep    arr dep_delay arr_delay carrier
##       <dttm> <int> <int> <int> <int>    <int>    <int> <chr>
## 1 2011-12-12 12:00:00     6     50    650    808      970      978    AA
## 2 2011-08-01 12:00:00     1     56    156    452      981      957    CO
## 3 2011-11-08 12:00:00     7     21    721    948      931      918    MQ
## 4 2011-06-21 12:00:00    23     34   2334    124      869      861    UA
## 5 2011-05-20 12:00:00     8     58    858   1027      803      822    MQ
## # ... with 7 more variables: flight <int>, dest <chr>, plane <chr>,
## #   cancelled <int>, time <int>, dist <int>, velocidad <dbl>
```

O los 5 con menor atraso de llegada

```
head(arrange(flights, arr_delay), n=5)
```

```
## # A tibble: 5 x 15
##   date     hour minute   dep   arr dep_delay arr_delay carrier
##   <dttm> <int> <int> <int> <int>    <int>    <int> <chr>
## 1 2011-07-03 12:00:00    19     14 1914  2039      -1     -70    XE
## 2 2011-12-25 12:00:00     7     41  741   926      -4     -57    00
## 3 2011-08-21 12:00:00     9     35  935  1039     -10     -56    00
## 4 2011-08-31 12:00:00     9     34  934  1039     -11     -56    00
## 5 2011-08-26 12:00:00    21      7 2107  2205      -3     -55    00
## # ... with 7 more variables: flight <int>, dest <chr>, plane <chr>,
## #   cancelled <int>, time <int>, dist <int>, velocidad <dbl>
```

Podemos arreglar primero por destino y luego por retraso de llegada.

```
arrange(flights, dest, arr_delay)
```

```
## # A tibble: 227,496 x 15
##   date     hour minute   dep   arr dep_delay arr_delay
##   <dttm> <int> <int> <int> <int>    <int>    <int>
## 1 2011-11-25 12:00:00    12     55 1255  1344      0     -26
## 2 2011-01-12 12:00:00     8     56  856  1000     -14     -25
## 3 2011-12-25 12:00:00    19     50 1950  2045      -5     -25
## 4 2011-03-16 12:00:00    17     39 1739  1841     -8     -24
## 5 2011-03-17 12:00:00    11     17 1117  1214     -3     -24
## 6 2011-12-30 12:00:00    17     27 1727  1828     -3     -24
## 7 2011-01-29 12:00:00    17     32 1732  1837     -3     -23
## 8 2011-05-29 12:00:00    18     11 1811  1902     -4     -23
## 9 2011-02-13 12:00:00    17     34 1734  1843     -1     -22
## 10 2011-04-17 12:00:00   17     18 1718  1818     -7     -22
## # ... with 227,486 more rows, and 8 more variables: carrier <chr>,
## #   flight <int>, dest <chr>, plane <chr>, cancelled <int>, time <int>,
## #   dist <int>, velocidad <dbl>
```

`rename`

Es posible renombrar las variables por nombre utilizando la función `rename`:

```
rename(flights, iata = dest)
```

```
## # A tibble: 227,496 x 15
##   date     hour minute   dep   arr dep_delay arr_delay
##   <dttm> <int> <int> <int> <int>    <int>    <int>
## 1 2011-01-01 12:00:00    14      0 1400  1500      0     -10
## 2 2011-01-02 12:00:00    14      1 1401  1501      1      -9
```

```

##  3 2011-01-03 12:00:00    13    52 1352 1502    -8    -8
##  4 2011-01-04 12:00:00    14     3 1403 1513     3     3
##  5 2011-01-05 12:00:00    14     5 1405 1507     5    -3
##  6 2011-01-06 12:00:00    13    59 1359 1503    -1    -7
##  7 2011-01-07 12:00:00    13    59 1359 1509    -1    -1
##  8 2011-01-08 12:00:00    13    55 1355 1454    -5   -16
##  9 2011-01-09 12:00:00    14    43 1443 1554    43    44
## 10 2011-01-10 12:00:00    14    43 1443 1553    43    43
## # ... with 227,486 more rows, and 8 more variables: carrier <chr>,
## #   flight <int>, iata <chr>, plane <chr>, cancelled <int>, time <int>,
## #   dist <int>, velocidad <dbl>

```

Datos limpios

Esta sección resume algunas de las funciones existentes para **limpiar** datos de distintos formatos a R. En particular, se utiliza la conceptualización de datos limpios presentada en Hadley Wickham y col. (2014) e implementada en el paquete **tidyverse** (Hadley Wickham 2016c). En la figura 5.5 podemos ver la etapa del análisis de datos correspondiente.

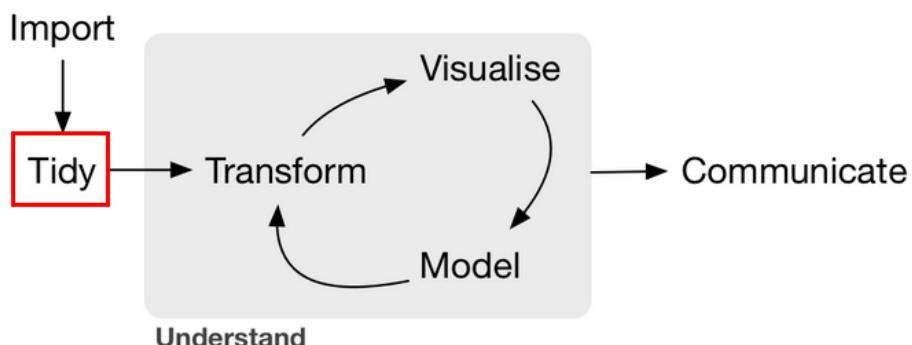


Figura 5.5: Limpieza de datos G. Grolemund y H. Wickham (2016, Introducción).

Mucho del esfuerzo en analítica lidia con la limpieza de datos. Tomar datos de diferentes fuentes y poderlas poner en la forma en la que uno los necesita para realizar analítica toma mucho tiempo y esfuerzo. Existen herramientas que permiten que esta parte sea más fácil y eficiente. Entre éstas se encuentran los criterios de datos limpios.

Los conjuntos de datos limpios (*tidy datasets*) permiten manipularlos fácilmente, modelarlos y visualizarlos. Aunque funcionan independientemente del lenguaje utilizado, son particularmente útiles en R pues éste es un lenguaje pensado para estructuras de datos tabulares; mismas que son más fácilmente explotables si se usan datos limpios (Hadley Wickham y Garrett Grolemund 2016).

Los datos limpios tienen una estructura específica: cada variable es una columna, cada

observación una fila y tipo de unidad observacional forma una tabla⁵.

Datos limpios en el procesamiento de datos

Esta actividad incluye una gran cantidad de elementos: revisión de valores atípicos, extracción de variables de cadenas en datos no estructurados, imputación de valores perdidos. Los datos limpios son tan solo un subconjunto de este proceso y lidian con el cómo estructurar los datos de manera que se facilite el análisis.

Los criterios de datos limpios están diseñados para facilitar la exploración inicial y el análisis de datos, así como simplificar el desarrollo de herramientas para el análisis de datos que trabajen bien con datos limpios.

Los criterios de datos limpios están muy relacionados a los de las bases de datos relacionales y, por ende, al álgebra relacional de Codd (Hadley Wickham y col. 2014). Sin embargo, se expresan y enmarcan en lenguaje que le es familiar a estadísticos.

Están creados para lidiar con conjuntos de datos que se encuentran en el mundo real pues -aunque parecen simples- es difícil encontrar datos limpios de origen. Los criterios de datos limpios proporcionan un marco mental a través del cual la intuición es explícita, es decir, proporcionan una manera estándar de ligar la estructura de un dataset (es decir su layout físico) con su semántica (su significado).

Estructura de datos: un ejemplo

La mayoría de los datos estadísticos están conformados por tablas rectangulares compuestas por filas y columnas. Las columnas casi siempre están etiquetadas (*colnames*) y las filas a veces lo están.

Tomamos el ejemplo de datos de la figura 5.6 en donde se presentan datos de un experimento⁶. La tabla contiene dos columnas y tres filas, ambas etiquetadas.

| | treatmenta | treatmentb |
|--------------|------------|------------|
| John Smith | — | 2 |
| Jane Doe | 16 | 11 |
| Mary Johnson | 3 | 1 |

Figura 5.6: Típica presentación de datos.

Podemos estructurar los datos de diferentes maneras pero la abstracción de filas y columnas solamente nos permite pensar en la representación transpuesta que se muestra en la figura 5.7. El diseño cambia pero los datos son los mismos. Además de la apariencia, deberíamos

⁵Hadley Wickham y Garrett Grolemund (2016, sección Data Tidying) explica este último criterio como “cada valor ocupa su propia celda”.

⁶Ejemplo tomado de (Hadley Wickham y col. 2014, p. 3).

de poder describir la semántica -el significado- de los valores que se muestran en una tabla (Hadley Wickham y col. 2014, p. 3) pero la abstracción de filas y columnas no da para más.

| | John Smith | Jane Doe | Mary Johnson |
|------------|------------|----------|--------------|
| treatmenta | — | 16 | 3 |
| treatmentb | 2 | 11 | 1 |

Figura 5.7: Mismos datos que en 5.6 pero traspuestos.

Semántica

Un conjunto de datos es una colección de **valores** (normalmente cuantitativos/números o cualitativos/caracteres). Los valores se organizan de dos maneras: cada valor pertenece simultáneamente a una variable y a una observación.

- Una *variable* contiene todos los valores de una medida y del mismo atributo subyacente (por ejemplo, temperatura, duración, altura, latitud) a través de unidades.
- Una *observación*, en cambio, contiene todos los valores medidos para la misma unidad (por ejemplo, una persona, un día, un municipio) a través de distintos atributos.

Los mismos datos en las figuras 5.6 y 5.7 los pensamos ahora en estos términos. Tenemos 3 variables:

1. *persona* con tres posibles valores (John, Jane, Mary)
2. *tratamiento* con dos posibles valores (a o b)
3. *resultado* con 6 valores (-, 16, 3, 2, 11, 1)

El diseño del experimento mismo nos habla de la estructura de las observaciones y los posibles valores que pueden tomar. Por ejemplo, en este caso el valor perdido nos dice que, por diseño, se debió de capturar esta variable pero no se hizo (por eso es importante guardarla como tal)⁷.

En la figura 5.8 se muestran los mismos datos que antes pero pensados tal que las variables son columnas y las observaciones (en este caso, cada punto en el diseño experimental) son filas.

Normalmente, es fácil determinar cuáles son las observaciones y cuáles son las variables en los distintos casos, pero es difícil dar una definición en forma precisa. Por ejemplo, si tienes teléfonos de casa y celulares, se pueden considerar como dos variables distintas en muchos contextos pero en prevención de fraude necesitas una variable que guarde el tipo de teléfono y otra en la que se guarde el número pues el uso regular del mismo número de teléfono por parte de la misma persona puede ayudar a detectarlo.

⁷Los valores perdidos estructurales, representan mediciones de valores que no se puede hacer o que no suceden y, por tanto, se pueden eliminar (por ejemplo, hombres embarazados). En este ejemplo, tenemos un valor perdido no estructural.

| name | trt | result |
|--------------|-----|--------|
| John Smith | a | — |
| Jane Doe | a | 16 |
| Mary Johnson | a | 3 |
| John Smith | b | 2 |
| Jane Doe | b | 11 |
| Mary Johnson | b | 1 |

Figura 5.8: Observaciones son filas, variables columnas.

En general, es más fácil describir las *relaciones funcionales entre las variables* que entre las filas pues las puedes operar fácilmente: por ejemplo, el radio entre dos variables, una combinación lineal de varias variables. También es más fácil *hacer comparaciones entre grupos de observaciones* que entre columnas: la suma, el promedio, la varianza, la moda (Hadley Wickham y col. 2014, p. 4).

Las observaciones, por su parte, son más complejas pues normalmente *se enmarcan en un análisis específico* que se desea realizar con los datos y existen varios niveles. Por ejemplo, en un análisis de ingreso podemos tener datos sociodemográficos de los individuos, datos geográficos del lugar en el que viven, datos macroeconómicos del tiempo específico, datos de la familia del individuo, datos del trabajo del individuo, entre otros.

Datos limpios

Éstos mapean de forma estándar el significado y la estructura de los datos. Un conjunto de datos se considera sucio o limpio dependiendo de cómo las filas, columnas y tablas mapean a observaciones, variables y tipos. En **datos limpios**:

1. Cada *variable* es una columna.
2. Cada *observación* es una fila.
3. Cada *tipo de unidad observacional* es una tabla / cada valor tiene su celda.

En la figura 5.9 podemos ver estos tres elementos de los datos limpios y cómo se representan en un *dataframe*.

Esto equivale a la tercera forma normal de Codd (Hadley Wickham y col. 2014, p. 4) enfocado a un solo conjunto de datos y no a datos conectados como en bases relacionales. Los datos sucios son cualquier otro tipo de manera de organizar los datos.

La tabla 5.8 corresponde a datos limpios: cada fila es una observación, es decir, el resultado de un tratamiento a una persona. Cada columna es una variable. Solo tenemos un tipo de unidad observacional, es decir, cada renglón es una unidad del diseño experimental.

Con los datos así ordenados, suele ser más fácil extraer datos que, por ejemplo, en Tabla 5.6.

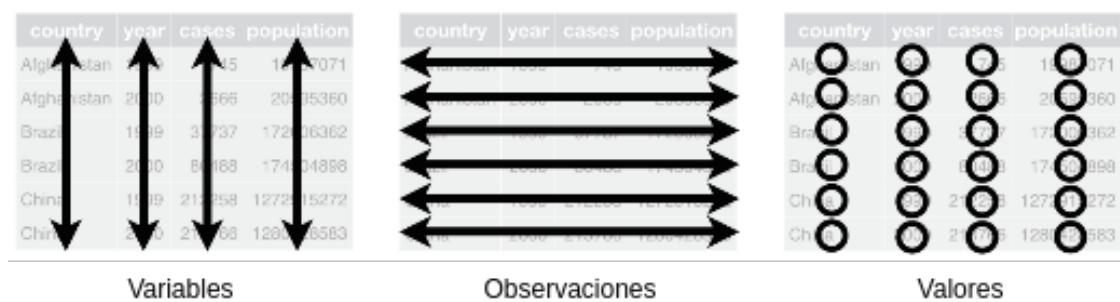


Figura 5.9: Ejemplificación de datos limpios (Hadley Wickham y Garrett Grolemund 2016, sección Data Tidying).



Ejercicios

1. Crea un dataframe con los valores de la tabla 5.6 y otro con los valores de la tabla 5.8.
2. Extrae el resultado para (John Smith, tratamiento a) en la primera configuración y en la segunda.
3. Especifica el número de tratamientos con la forma sucia y la forma limpia.
4. ¿Cuál es la media de los resultados? Usa la forma 1 y la forma 2.
5. Extrae los tratamientos del tipo a en la forma 2.

Los datos limpios permiten hacerle preguntas a los datos de manera simple y sistemática. En particular, es una estructura muy útil para programación vectorizada como la que R tiene (el ejercicio 5) porque la *forma* en la que almacenamos los datos se asegura que valores para diferentes variables de la misma observación siempre están apareados.

Por convención, las variables se acomodan de una forma particular. Las variables *fijas* (en este ejemplo, las propias al diseño experimental) van primero y posteriormente las variables *medidas*. Ordenamos éstas de forma que las que están relacionadas sean contiguas.

De sucio a limpio

Los conjuntos de datos normalmente **no cumplen** con estos criterios. Es raro obtener un conjunto de datos con el cuál podemos trabajar de manera inmediata.

Los 5 problemas más comunes para llevar datos sucios a limpios (Hadley Wickham y col. 2014, p. 5) son:

1. Los nombres de las columnas son valores, no nombres de variables.
2. Múltiples variables se encuentran en la misma columna.
3. Las variables están guardadas tanto en filas como en columnas.
4. Muchos tipos de unidad observacional se encuentran en la misma tabla.
5. Una sola unidad observacional se guardó en varias tablas.

Estos problemas pueden ser resueltos con las funciones implementadas en el paquete `tidyverse`:

`gather`, `spread`, `separate` y `unite`.

Los nombres de las columnas son valores, no nombres de variables

La tabla 5.3 muestra datos sucios con este problema. La base acompaña al paquete `tidyR` (Hadley Wickham 2016c) y es una muestra de los datos del reporte sobre tuberculosis de la organización mundial de la salud. Contiene observaciones anuales por país para casos de tuberculosis según distintos grupos.

La descripción de todas las variables se puede ver tecleando `?who`

Dentro de un reporte, la representación que se tiene de las variables tiene sentido. Por ejemplo, en la tabla 5.3 vemos los casos de tuberculosis para distintos grupos de edad de hombres en México para cierto tipo de diagnóstico.

| country | year | new_sp_m014 | new_sp_m1524 | new_sp_m2534 | new_sp_m3544 | new_sp_m4554 | new_sp_m5564 | new_sp_m65 |
|---------|------|-------------|--------------|--------------|--------------|--------------|--------------|------------|
| Mexico | 2000 | 214 | 1079 | 1387 | 1162 | 1235 | 972 | 1126 |
| Mexico | 2001 | 130 | 1448 | 1639 | 1683 | 1606 | 1229 | 1566 |
| Mexico | 2002 | 154 | 1090 | 1292 | 1301 | 1146 | 986 | 1144 |
| Mexico | 2003 | 187 | 1207 | 1461 | 1417 | 1313 | 1005 | 1352 |
| Mexico | 2004 | 86 | 1053 | 1276 | 1181 | 1201 | 958 | 1209 |
| Mexico | 2005 | 100 | 1095 | 1376 | 1314 | 1238 | 1042 | 1288 |
| Mexico | 2006 | 129 | 986 | 1320 | 1333 | 1275 | 1012 | 1215 |
| Mexico | 2007 | 145 | 981 | 1286 | 1286 | 1266 | 942 | 1226 |
| Mexico | 2008 | 124 | 966 | 1292 | 1314 | 1267 | 1004 | 1213 |
| Mexico | 2009 | 103 | 1030 | 1262 | 1401 | 1360 | 1024 | 1252 |
| Mexico | 2010 | 125 | 1081 | 1375 | 1380 | 1392 | 1119 | 1303 |

Cuadro 5.3: Casos de tuberculosis para México del 2000 al 2010 para hombres con diagnóstico por lesiones de pulmón.

En las columnas tenemos varias variables: método de diagnóstico, género y categorías de edad. Para arreglarlo, necesitamos *juntar* (`gather`) las columnas con valores de variables en una sola columna que contenga esos nombres como valores. En otras palabras, debemos convertir de la columna 5 en adelante en filas.

Con el paquete `tidyR` esto se puede realizar en forma fácil con el comando `gather` de manera que obtenemos un dataframe como el que se muestra en la tabla 5.4.

```
junta <- tidyR::gather(who, key = variables, value = casos
                      , -country, -iso2, -iso3, -year, na.rm = T)
```

Los parámetros que recibe la función `gather` son (al menos):

- El `data.frame` como primer parámetro.
- La llave (parámetro `key`) será el nombre que tomará la variable con los nombres de las columnas a juntar.
- El valor (parámetro `value`) es el nombre de la variable que contendrá los valores correspondientes a cada valor (el diagnóstico i –ésimo, el j –ésimo género y el k –ésimo grupo de edad).
- Al último, especificamos las variables que **NO** se deben de juntar (en este caso: el país, iso2, iso3 y el año).

Hay parámetros opcionales en la función. Para estos datos en particular, por ejemplo, es conveniente remover los grupos para los que no se tiene el dato con el parámetro `na.rm = TRUE`.

| country | iso2 | iso3 | year | variables | casos |
|-------------|------|------|------|-------------|-------|
| Afghanistan | AF | AFG | 1997 | new_sp_m014 | 0 |
| Afghanistan | AF | AFG | 1998 | new_sp_m014 | 30 |
| Afghanistan | AF | AFG | 1999 | new_sp_m014 | 8 |
| Afghanistan | AF | AFG | 2000 | new_sp_m014 | 52 |
| Afghanistan | AF | AFG | 2001 | new_sp_m014 | 129 |
| Afghanistan | AF | AFG | 2002 | new_sp_m014 | 90 |

Cuadro 5.4: Valores de variables en una sola variable.

gather

En la figura se ejemplifica la operacionalización de ‘gather’ para los datos de tratamiento.

The diagram illustrates the 'gather' operation. On the left, a data frame has three columns: 'name', 'treatmenta', and 'treatments'. The 'treatmenta' column contains values 'NA', '16', and '3'. The 'treatments' column contains values '2', '11', and '1'. A red box highlights the 'NA' value. An annotation above the 'NA' cell says 'Se elimina por na.rm = T' (Deleted by na.rm = T). An arrow labeled 'gather' points from this row to the right, where a melted data frame is shown. This melted frame has columns 'name', 'trt', and 'result'. It contains six rows: (Jane Doe, a, 16), (Mary Johnson, a, 3), (John Smith, b, 2), (Jane Doe, b, 11), (Jane Doe, b, 11), and (Mary Johnson, b, 1). The second and third rows represent the same person (Jane Doe) under different treatment codes (a and b), which have been combined into a single row per person.

| name | treatmenta | treatments |
|--------------|------------|------------|
| John Smith | NA | 2 |
| Jane Doe | 16 | 11 |
| Mary Johnson | 3 | 1 |

| name | trt | result |
|--------------|-----|--------|
| Jane Doe | a | 16 |
| Mary Johnson | a | 3 |
| John Smith | b | 2 |
| Jane Doe | b | 11 |
| Mary Johnson | b | 1 |

Esto es resultado de ejecutar:

```
gather(fig1, key = trt, value = result, -name, na.rm = T) %>%
  mutate(trt = gsub("treatment", "", trt))
```

Ejercicio

Este tipo de formato de datos (poner valores de variables en las columnas) es útil también cuando se capturan datos al evitar la repetición de valores.

Por ejemplo, pensemos en un experimento clínico en el que seguimos a sujetos a lo largo de un tratamiento midiendo su IMC. Una forma muy sencilla de guardar los datos del experimento es utilizando un procesador de texto común. El capturista no querrá seguir criterios de datos limpios al llenar la información pues implicaría repetir el nombre de la persona, el día de la captura y el nivel de colesterol. Supongamos un experimento con 16 sujetos a lo largo de un año en donde se mide el colesterol una vez al mes (mes1, mes2, etc.). Los datos capturados se muestran en la tabla 5.5.

Nuevamente, queremos convertir la columna 3 a 14 en filas, es decir, observaciones. Utiliza el comando ‘gather’ para realizar esto y obtener el resultado que se muestra en la tabla 5.6.

| sujetos | grupo | mes1 | mes2 | mes3 | mes4 | mes5 | mes6 | mes7 | mes8 | mes9 | mes10 | mes11 | mes12 |
|---------|-------------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| A | control | 23.46 | 24.86 | 25.25 | 25.18 | 25.79 | 25.90 | 26.02 | 26.24 | 27.85 | 27.55 | 28.07 | 27.39 |
| B | tratamiento | 22.73 | 23.33 | 25.44 | 26.10 | 24.85 | 25.36 | 25.70 | 25.55 | 26.77 | 28.43 | 28.26 | 28.93 |
| C | control | 26.80 | 26.95 | 27.47 | 27.11 | 28.56 | 29.35 | 29.74 | 31.04 | 32.37 | 32.37 | 33.18 | 32.19 |
| D | control | 31.08 | 31.72 | 31.04 | 30.58 | 31.92 | 33.24 | 32.06 | 31.29 | 31.55 | 30.36 | 30.72 | 30.89 |
| E | control | 19.07 | 17.65 | 17.30 | 17.14 | 17.11 | 17.39 | 19.11 | 19.12 | 18.64 | 18.73 | 19.92 | 19.87 |
| F | control | 19.18 | 18.81 | 20.20 | 21.24 | 22.82 | 24.50 | 25.23 | 26.09 | 24.97 | 26.37 | 28.18 | 28.18 |
| G | tratamiento | 33.53 | 32.79 | 32.94 | 33.87 | 34.82 | 34.82 | 34.87 | 33.97 | 34.18 | 34.78 | 33.68 | 35.36 |
| H | tratamiento | 20.20 | 21.11 | 23.15 | 24.67 | 24.33 | 26.24 | 27.57 | 27.88 | 29.70 | 31.76 | 31.98 | 31.53 |
| I | control | 19.84 | 21.73 | 20.98 | 22.12 | 22.24 | 23.73 | 24.22 | 23.08 | 21.84 | 22.43 | 22.60 | 23.06 |
| J | tratamiento | 25.08 | 25.08 | 25.35 | 26.13 | 27.71 | 28.61 | 27.93 | 26.91 | 27.51 | 27.41 | 27.87 | 27.35 |
| K | control | 19.76 | 20.98 | 20.09 | 21.90 | 21.73 | 23.54 | 25.66 | 26.40 | 27.47 | 26.41 | 28.22 | 30.77 |
| L | control | 25.11 | 26.30 | 25.56 | 27.17 | 29.32 | 29.47 | 29.34 | 31.59 | 32.60 | 32.85 | 33.01 | 33.24 |
| M | control | 33.69 | 34.61 | 36.90 | 36.51 | 36.56 | 36.98 | 38.92 | 40.26 | 42.56 | 42.68 | 43.35 | 43.52 |
| N | control | 28.18 | 27.48 | 28.46 | 28.91 | 30.38 | 29.29 | 29.06 | 29.68 | 31.65 | 31.95 | 33.92 | 33.72 |
| O | control | 31.94 | 31.74 | 31.73 | 31.69 | 33.14 | 32.84 | 31.13 | 30.93 | 31.11 | 30.49 | 31.10 | 32.80 |
| P | control | 17.99 | 18.08 | 15.83 | 16.70 | 19.22 | 19.39 | 20.94 | 20.87 | 20.20 | 19.98 | 19.92 | 20.65 |

Cuadro 5.5: Mediciones de IMC en sujetos.

| sujetos | grupo | mes | IMC |
|---------|-------------|-------|-------|
| D | control | mes9 | 31.55 |
| K | control | mes1 | 19.76 |
| G | tratamiento | mes6 | 34.82 |
| E | control | mes10 | 18.73 |
| C | control | mes12 | 32.19 |
| H | tratamiento | mes3 | 23.15 |
| O | control | mes5 | 33.14 |
| M | control | mes10 | 42.68 |
| B | tratamiento | mes3 | 25.44 |
| L | control | mes8 | 31.59 |

Cuadro 5.6: Muestra de datos limpios para experimentos IMC.

```
# Creamos los datos
df <- data.frame(
  sujetos = LETTERS[1:16],
  grupo = sample(c("control", "tratamiento"), size = 16, replace = T, prob = c(0.5, 0.5)),
  # , meses = as.vector(sapply(paste0("mes", 1:12), rep, 16))
)
m <- t(sapply(runif(16, 16, 35), FUN = function(x){cumsum(c(x, rnorm(11, mean = 0.5, sd = 1.5)))}))
colnames(m) <- paste0("mes", 1:12)
df <- cbind(df, m)

# Respuesta: opción 1
tidyverse::gather(df, key = mes, value = IMC, -sujetos, -grupo)
# opción 2
tidyverse::gather(df, key = mes, value = IMC, mes1:mes12)
```

Múltiples variables se encuentran en la misma columna

Otra forma de datos sucios es cuando una columna con nombres de variables tiene realmente varias variables dentro del nombre.

Si regresamos al ejemplo de la sección anterior, podemos notar que todavía no se tienen datos limpios. Primero, notamos una redundancia: todos los valores tienen el sufijo “new_” o “new” pero éste no tiene significado. Eliminamos ese pedazo de texto de los valores con la función `gsub`.

Segundo, debemos extraer los valores de las variables método de diagnóstico, género y categoría de edad de la columna que acabamos de construir (que llamamos “variables”).

En la descripción de las variables (teclea `?who`) se describen a los títulos de las columnas (que ahora están guardados en la variable `variables`) tales que contienen como prefijo “new_”, seguido del diagnóstico que puede ser de dos o tres caracteres, “_f” para mujeres o “_m” para hombres y, por último, el rango de edad.

Expresiones regulares o regex

En esencia, las *regex* (por el inglés regular expressions) utilizan caracteres para definir patrones de más caracteres. Éstos se conocen como metacaracteres y pueden ser herramientas poderosas en la limpieza de datos.

Pueden ser utilizadas para buscar una cadena en específico en forma exacta, buscar una cadena dentro de otra o para reemplazar una parte de una cadena con otra cosa. Peng, S Kross y Anderson (2016, sección “text processing and regular expressions”) realiza una buena revisión de las funciones disponibles en R para utilizar expresiones regulares.

Para eso, utilizamos la función `extract` del paquete `tidyverse`. A esta función, debemos decirle cuál es el nombre de la variable que contiene varios valores (parámetro `col`), los nuevos nombres de columnas (parámetro `into`) y la expresión regular con la que irá capturando los pedazos y asignándolos a la columna correcta (parámetro `regex`).

```
limpios <- junta %>%
  mutate(variables = gsub("new_|new", "", variables)) %>%
  tidyverse::extract(., col = variables
    , into = c("diagnostico", "genero", "edad")
    , regex = "[[:alnum:]]+([a-z])([0-9]+)")
```

Por último, se deben limpiar las categorías de edad. En este caso, se decide volverlos un factor con las categorías ordenadas por los grupos de edad existentes en la base:

```
limpios %<>%
  mutate(
    edad = factor(edad,
      levels = c("014", "1524", "2534", "3544"
                 , "4554", "5564", "65")
      , labels = c("0-14", "15-24", "25-34", "35-44"
                 , "45-54", "55-64", "65>")
      , ordered = T)
  )
```

Nota

Nota como en el último ejemplo, se utiliza el símbolo %<>% que es equivalente a realizar una asignación (<-) y un pipe (%>%) de los datos que están guardados en el dataframe limpios.

De esta forma, obtenemos los datos como se ven en la tabla 5.7 donde tenemos una variable para el método de diagnóstico, una para el género, otra para la edad y una última con el número de casos observados.

| country | iso2 | iso3 | year | diagnostico | genero | edad | casos |
|-------------|------|------|------|-------------|--------|------|-------|
| Afghanistan | AF | AFG | 1997 | sp | m | 0-14 | 0 |
| Afghanistan | AF | AFG | 1998 | sp | m | 0-14 | 30 |
| Afghanistan | AF | AFG | 1999 | sp | m | 0-14 | 8 |
| Afghanistan | AF | AFG | 2000 | sp | m | 0-14 | 52 |
| Afghanistan | AF | AFG | 2001 | sp | m | 0-14 | 129 |
| Afghanistan | AF | AFG | 2002 | sp | m | 0-14 | 90 |

Cuadro 5.7: Cada columna es una variable.

Nota

Esta forma es limpia pues cada columna es una variable, cada fila es una observación y no se mezclan unidades observacionales.

Ejercicio

A continuación se crea el dataframe *pob* que contiene un identificador para el individuo (*id*) y una columna llamada *variables* que contiene el sexo, año de nacimiento y la altura en centímetros todos en una columna y separados por “_”.

Utiliza la función *separate* del paquete *tidyverse* para limpiar estos datos.

```
# Respuestas
# Generamos los datos
pob <- tibble(
  id = 1:1000
  , variables = paste0(
    sample(x = c('f', 'm'), size = 1000, replace = T)
    , "_"
    , floor(runif(1000, 45, 99))
    , "_"
    , floor(runif(1000, 50, 190))
  )
)

# Utilizamos separate para generar las variables:
# sexo, año de nacimiento y altura
pob.tidy <- pob %>%
  separate(col = variables
    , into = c("sexo", "anio_nac", "altura"), sep = "_")
```

```
# Pasamos a enteros las variables anio de nac y altura
pob.tidy %<%
  mutate_each(funs(as.integer), anio_nac, altura)

pob.tidy
```

Las variables están guardadas tanto en filas como en columnas

Uno de los problemas más difíciles es cuando las variables están tanto en filas como en columnas.

Para exemplificar este problema, se muestran los datos de temperatura máxima y mínima en algunas zonas de México (Hadley Wickham 2014b, archivo: data/weather.txt). Los datos que limpiaremos se ven en la tabla 5.8. Como se puede ver, tenemos valores del día del mes de la observación como nombres de variables: d1 (día 1), d2 (día 2), etc. Esto es homólogo al problema 1 visto anteriormente.

También tenemos variables en las filas: la temperatura máxima y la temperatura mínima deberían ser el nombre de las columnas.

| id | year | month | element | d1 | d2 | d3 | d4 | d5 | d6 | d7 | d8 | d9 | d10 | d11 |
|---------|------|-------|---------|----|-------|-------|-------|-------|-------|-------|----|----|-----|-------|
| MX17004 | 2010 | 1 | tmax | | | | | | | | | | | |
| MX17004 | 2010 | 1 | tmin | | | | | | | | | | | |
| MX17004 | 2010 | 2 | tmax | | 27.30 | 24.10 | | | | | | | | 29.70 |
| MX17004 | 2010 | 2 | tmin | | 14.40 | 14.40 | | | | | | | | 13.40 |
| MX17004 | 2010 | 3 | tmax | | | | | 32.10 | | | | | | 34.50 |
| MX17004 | 2010 | 3 | tmin | | | | | 14.20 | | | | | | 16.80 |
| MX17004 | 2010 | 4 | tmax | | | | | | | | | | | |
| MX17004 | 2010 | 4 | tmin | | | | | | | | | | | |
| MX17004 | 2010 | 5 | tmax | | | | | | | | | | | |
| MX17004 | 2010 | 5 | tmin | | | | | | | | | | | |
| MX17004 | 2010 | 6 | tmax | | | | | | | | | | | |
| MX17004 | 2010 | 6 | tmin | | | | | | | | | | | |
| MX17004 | 2010 | 7 | tmax | | | 28.60 | | | | | | | | |
| MX17004 | 2010 | 7 | tmin | | | 17.50 | | | | | | | | |
| MX17004 | 2010 | 8 | tmax | | | | | 29.60 | | | | | | 29.00 |
| MX17004 | 2010 | 8 | tmin | | | | | 15.80 | | | | | | 17.30 |
| MX17004 | 2010 | 10 | tmax | | | | | 27.00 | | 28.10 | | | | |
| MX17004 | 2010 | 10 | tmin | | | | | 14.00 | | 12.90 | | | | |
| MX17004 | 2010 | 11 | tmax | | 31.30 | | 27.20 | | 26.30 | | | | | |
| MX17004 | 2010 | 11 | tmin | | 16.30 | | 12.00 | | 7.90 | | | | | |
| MX17004 | 2010 | 12 | tmax | | 29.90 | | | | | 27.80 | | | | |
| MX17004 | 2010 | 12 | tmin | | 13.80 | | | | | 10.50 | | | | |

Cuadro 5.8: Mediciones de temperatura max y min.

Para limpiar, lo primero que debemos hacer es juntar los días (que son valores de la variable día) en una sola columna. Después utilizamos la nueva variable para crear la fecha. Así, obtenemos la tabla 5.10, a partir de los datos en el dataframe *raw*.

```
# Tidy
# Primero, juntamos la variable dias
clean1 <- tidyverse::gather(raw, key = variable, value = value, d1:d31
                           , na.rm = T)

# Despues, generamos la variable dia y fecha
clean1$day <- as.integer(str_replace(clean1$variable, "d", ""))
```

```
clean1$date <- as.Date(ISOdate(clean1$year, clean1$month, clean1$day))

# Seleccionamos las variables limpias y ordenamos
clean1 <- dplyr::select_(clean1, "id", "date", "element", "value") %>%
  dplyr::arrange(date, element)
```

stringr

Otro paquete muy útil para realizar tareas de limpieza con cadenas. La documentación detalla todas sus funciones. En este caso, utilizamos la función `str_replace` que nos permite reemplazar una cadena de caracteres por otra.

| id | date | element | value |
|---------|----------|---------|-------|
| MX17004 | 14639.00 | tmax | 27.80 |
| MX17004 | 14639.00 | tmin | 14.50 |
| MX17004 | 14642.00 | tmax | 27.30 |
| MX17004 | 14642.00 | tmin | 14.40 |
| MX17004 | 14643.00 | tmax | 24.10 |

Cuadro 5.9: Paso 1. Juntar las columnas, limpiar días, crear fecha.

El segundo paso es transformar la variable `element` en dos columnas pues, en realidad, almacena dos variables: temperatura máxima y mínima.

```
# Las temperaturas van a columnas
clean2 <- tidyr::spread(clean1, key = element, value = value)
```

| id | date | element | value |
|---------|----------|---------|-------|
| MX17004 | 14639.00 | tmax | 27.80 |
| MX17004 | 14639.00 | tmin | 14.50 |
| MX17004 | 14642.00 | tmax | 27.30 |
| MX17004 | 14642.00 | tmin | 14.40 |
| MX17004 | 14643.00 | tmax | 24.10 |

Cuadro 5.10: Paso 2. Enviar a columnas las mediciones de temperaturas.

En este caso, se utilizó la función `spread` del paquete `tidyverse`. Esta función realiza una especie de inverso a la operación que hace `gather`. En lugar de juntar nombres de variables, utiliza los valores de una variable como nombres de columnas (parámetro `key`) y rellena apropiadamente las celdas con los valores de otra variable (parámetro `value`). Los demás parámetros son opcionales y, por ejemplo, en lugar de tener un parámetro para especificar qué hacer con los `NA` (en `gather`: `na.rm`), en este caso pide el parámetro `fill` cuyo default es `NA` pero, en algunos casos, es más apropiado insertar otro valor para llenar valores de celdas en donde no había un valor correspondiente.

Muchos tipos de unidad observacional se encuentran en la misma tabla

En ocasiones las bases de datos involucran diferentes tipos de unidad observacional. Para tener datos limpios, cada unidad observacional debe estar almacenada en su propia tabla.

Para este ejemplo, utilizamos la base de datos `billboard` (Hadley Wickham 2014b, archivo: `data/billboard.csv`)

```
billboard <- readr::read_csv("tidyverse/datasets/billboard.csv")
billboard_long <- gather(billboard, week, rank, x1st.week:x76th.week
                         , na.rm = TRUE)
billboard_tidy <- billboard_long %>%
  mutate(
    week = extract_numeric(week),
    date = as.Date(date.entered) + 7 * (week - 1)) %>%
  select(-date.entered)
head(billboard_tidy)

## # A tibble: 6 x 9
##   year      artist.inverted          track     time
##   <int>      <chr>                <chr>     <time>
## 1 2000      Destiny's Child       Independent Women Part I 03:38:00
## 2 2000      Santana              Maria, Maria 04:18:00
## 3 2000      Savage Garden        I Knew I Loved You 04:07:00
## 4 2000      Madonna              Music 03:45:00
## 5 2000 Aguilera, Christina Come On Over Baby (All I Want Is You) 03:38:00
## 6 2000      Janet                Doesn't Really Matter 04:17:00
## # ... with 5 more variables: genre <chr>, date.peaked <date>, week <dbl>,
## #   rank <chr>, date <date>
```



Ejercicio

¿Cuáles son las unidades observacionales en esta tabla?

Respuesta

```
# Tenemos por un lado una unidad observacional: las características de la
# canción.

# Por el otro tenemos otra unidad observacional: las posiciones que tuvieron
# las canciones en cada semana.
```

Debemos separar las unidades observacionales, esto significa separar la base de datos en dos: la tabla `canciones` que almacena artista, nombre de la canción y duración; la tabla `posiciones` que almacena el ranking de la canción en cada semana.

```
canciones <- billboard_tidy %>%
  select(artist.inverted, track, year, time) %>%
```

```

unique() %>%
arrange(artist.inverted) %>%
mutate(song_id = row_number(artist.inverted))

head(canciones, 5)

## # A tibble: 5 x 5
##   artist.inverted
##   <chr>
## 1 2 Pac
## 2 2Ge+her
## 3 3 Doors Down
## 4 3 Doors Down
## 5 504 Boyz
## # ... with 4 more variables: track <chr>, year <int>, time <time>,
## #   song_id <int>

posiciones <- billboard_tidy %>%
  left_join(canciones, c("artist.inverted", "track", "year", "time")) %>%
  select(song_id, date, week, rank) %>%
  arrange(song_id, date) %>%
 tbl_df
posiciones

## # A tibble: 5,307 x 4
##   song_id      date    week  rank
##   <int>     <date> <dbl> <chr>
## 1 1 2000-02-26     1    87
## 2 1 2000-03-04     2    82
## 3 1 2000-03-11     3    72
## 4 1 2000-03-18     4    77
## 5 1 2000-03-25     5    87
## 6 1 2000-04-01     6    94
## 7 1 2000-04-08     7    99
## 8 2 2000-09-02     1    91
## 9 2 2000-09-09     2    87
## 10 2 2000-09-16    3    92
## # ... with 5,297 more rows

```

Una sola unidad observacional se guardó en varias tablas

Este ejemplo y datos se toman de Ortiz (2015).

Es común que los valores sobre una misma unidad observacional estén separados en varios archivos. Muchas veces, cada archivo es una variable, e.g. el mes o el nombre del paciente,

etc. Para limpiar estos datos debemos:

1. Leemos los archivos en una lista de tablas.
2. Para cada tabla agregamos una columna que registra el nombre del archivo original.
3. Combinamos las tablas en un solo dataframe.

La carpeta `tidyR_datasets/specdata` contiene 332 archivos csv que almacenan información de monitoreo de contaminación en 332 ubicaciones de EUA. Cada archivo contiene información de una unidad de monitoreo y el número de identificación del monitor es el nombre del archivo.

Primero creamos un vector con los nombres de los archivos en un directorio con extensión .csv.

```
paths <- dir("tidyR_datasets/specdata", pattern = "\\.csv$"
            , full.names = TRUE)
names(paths) <- basename(paths)
specdata_US <-tbl_df(ldply(paths, read.csv, stringsAsFactors = FALSE))
specdata_US %>% head
```

```
## # A tibble: 6 x 5
##       .id      Date sulfate nitrate   ID
##     <chr>    <chr>   <dbl>   <dbl> <int>
## 1 001.csv 2003-01-01     NA     NA     1
## 2 001.csv 2003-01-02     NA     NA     1
## 3 001.csv 2003-01-03     NA     NA     1
## 4 001.csv 2003-01-04     NA     NA     1
## 5 001.csv 2003-01-05     NA     NA     1
## 6 001.csv 2003-01-06     NA     NA     1
```

Las variables quedaron un poco sucias... las limpiamos y seleccionamos solo las de interés.

```
specdata <- specdata_US %>%
  mutate(
    monitor = extract_numeric(.id),
    date = as.Date(Date)) %>%
  select(id = ID, monitor, date, sulfate, nitrate)
specdata %>% head
```

```
## # A tibble: 6 x 5
##       id monitor      date sulfate nitrate
##     <int>   <dbl>    <date>   <dbl>   <dbl>
## 1     1       1 2003-01-01     NA     NA
## 2     1       1 2003-01-02     NA     NA
## 3     1       1 2003-01-03     NA     NA
## 4     1       1 2003-01-04     NA     NA
## 5     1       1 2003-01-05     NA     NA
## 6     1       1 2003-01-06     NA     NA
```

Ejercicio

En la carpeta `tidyR_datasetsinforme` se encuentran dos archivos que contienen los datos de la relación del pago de pensiones del IMSS e ISSSTE respecto a su gasto programable devengado por entidad federativa.

Los datos tienen todos los problemas explorados en esta sección, salvo el que solo hay un tipo de unidad observacional.

Los datos están divididos en los archivos `M2_218.xlsx` y `M2_219.xlsx` y se puede observar un ejemplo de éstos en la figura 5.10. El ejercicio consiste en leer los datos y limpiarlos de forma que sean datos limpios^a.

^aLos datos fueron tomados de *Cuarto informe de gobierno, 2015-2016. Anexo estadístico.* (2016, p. 218-219) y el catálogo de estados se tomó de (MGN 2005-2016)

Relación del pago de pensiones del IMSS e ISSSTE respecto a su gasto programable devengado, por entidad federativa^{1/}
(Porcentajes)

| Entidad federativa | IMSS | | | | | | | | | ISSSTE | | | | | |
|---------------------|------|------|------|------|------|------|------|------|------|--------|------|------|------|------|------|
| | 2000 | 2001 | 2002 | 2003 | 2004 | 2005 | 2006 | 2007 | 2008 | 2000 | 2001 | 2002 | 2003 | 2004 | 2005 |
| Total nacional | 41.2 | 41.1 | 40.6 | 37.3 | 41.4 | 39.5 | 40.9 | 43.9 | 45.8 | 45.7 | 44.6 | 51.9 | 54.1 | 54.0 | 56.5 |
| Aguascalientes | 41.5 | 23.9 | 38.4 | 35.5 | 40.2 | 36.4 | 39.3 | 40.8 | 42.7 | 59.6 | 60.9 | 64.0 | 67.0 | 66.6 | 68.6 |
| Baja California | 38.9 | 28.5 | 45.5 | 35.0 | 39.2 | 37.4 | 37.8 | 40.6 | 42.4 | 59.4 | 61.3 | 63.9 | 67.3 | 66.7 | 69.4 |
| Baja California Sur | 28.2 | 15.9 | 31.1 | 21.1 | 24.2 | 23.0 | 23.3 | 26.2 | 28.9 | 42.6 | 51.8 | 54.9 | 57.6 | 57.3 | 59.7 |
| Campeche | 37.5 | 18.3 | 33.7 | 29.5 | 32.2 | 31.5 | 32.5 | 34.2 | 36.2 | 40.3 | 50.8 | 51.0 | 55.7 | 56.1 | 59.5 |
| Coahuila | 46.5 | 27.3 | 45.1 | 40.9 | 43.7 | 41.9 | 43.8 | 46.7 | 48.9 | 57.6 | 62.9 | 66.3 | 68.7 | 67.2 | 69.7 |
| Colima | 37.9 | 19.3 | 35.5 | 31.3 | 35.4 | 33.6 | 34.4 | 37.4 | 40.2 | 47.4 | 57.0 | 59.6 | 60.4 | 57.0 | 56.6 |
| Chiapas | 34.9 | 11.7 | 25.8 | 23.8 | 25.3 | 31.5 | 31.7 | 35.1 | 37.5 | 48.8 | 57.6 | 61.4 | 63.5 | 64.5 | 64.9 |
| Chihuahua | 39.2 | 27.3 | 44.9 | 34.5 | 38.4 | 38.2 | 37.5 | 39.3 | 42.3 | 50.0 | 64.3 | 67.2 | 69.3 | 68.8 | 70.7 |
| Ciudad de México | 35.8 | 24.7 | 39.4 | 35.5 | 43.0 | 37.0 | 39.3 | 42.8 | 43.4 | 39.1 | 32.2 | 41.3 | 42.4 | 42.5 | 44.4 |
| Durango | 42.7 | 95.9 | 35.3 | 35.0 | 38.3 | 38.3 | 39.7 | 41.3 | 43.6 | 50.4 | 49.6 | 54.5 | 56.6 | 57.0 | 61.2 |
| Guanajuato | 45.3 | 27.9 | 46.0 | 37.9 | 42.0 | 39.6 | 41.8 | 43.7 | 46.3 | 43.3 | 49.6 | 53.3 | 57.0 | 56.0 | 55.4 |
| Guerrero | 35.4 | 17.8 | 31.2 | 29.2 | 31.3 | 29.6 | 31.7 | 33.4 | 34.4 | 44.8 | 47.0 | 51.4 | 53.8 | 51.7 | 58.1 |
| Hidalgo | 43.5 | 18.7 | 35.9 | 34.2 | 38.0 | 40.5 | 41.7 | 45.0 | 47.9 | 49.4 | 56.5 | 61.2 | 63.8 | 64.0 | 65.0 |

Figura 5.10: Datos de pensiones del IMSS e ISSSTE 2000 a 2016 (*Cuarto informe de gobierno, 2015-2016. Anexo estadístico.* 2016, p. 218-219).

```
# Respuesta
rm(list = ls())
# Funciones auxiliares

# Como son archivos de excel, muchas líneas están vacías pero R no
# lo entiende. Esta función quita esas líneas.
quita.nas <- function(df, prop.na = 0.8) {
  df <- df[, !is.na(names(df)) & names(df) != ""]
  r <- sapply(seq(nrow(df)), FUN = function(x){
    sum(is.na(df[x,]))
  })
  r <- r/ncol(df)
  df[r < prop.na, ]
}

# Como pegaremos estados a partir de sus nombres (y no una clave)
# Esta función limpia los nombres de los estados
limpia <- function(nn) {
  gsub("\s+", " ", stringr::str_trim(nn)) %>%
  
```

```

gsub("^ *|(?<= ) | *$", "", ., perl=T) %>%
  iconv(., to='ASCII//TRANSLIT') %>%
  tolower(.)
}

## Funcion para pegar la clave del estado a partir del nombre
pega.estados <- function(df, nombres = "entidad") {
  df$pega <- limpia(df[[nombres]])

  d <- rbind(readRDS("tidyR_datasets/informe/estados_p.rds"),
             data.frame(
               estado = c("00", "00", "30", "16", "15", "05")
               , pega = c("nacional", "total nacional", "veracruz"
                         , "michoacan", "estado de mexico", "coahuila")
               , stringsAsFactors = F
             )
  )
}

dd <- dplyr::left_join(df, d)
dplyr::select(dd, -pega)
}

# Limpieza de datos
df <- read_excel("tidyR_datasets/informe/M2_218.xlsx", skip = 4
                 , col_names = T)
names(df) <- c('entidad', paste0('imss_', 2000:2008)
               , paste0('issste_', 2000:2008))
df <- df %>%
  quita.nas(.) %>%
  pega.estados(.) %>%
  tidyR::gather(., key = variable, value = valor, -entidad
                , -estado, na.rm = T) %>%
  tidyR::separate(variable, c("indicador", "anio"), extra = "drop") %>%
  dplyr::mutate(indicador = paste0("pago_pensiones_", indicador))

all <- df
df <- read_excel("tidyR_datasets/informe/M2_219.xlsx", skip = 4
                 , col_names = T)
names(df) <- c('entidad', paste0('imss_', 2009:2016)
               , paste0('issste_', 2009:2016))
df <- df %>%
  quita.nas(.) %>%
  pega.estados(.) %>%
  tidyR::gather(., key = variable, value = valor, -entidad

```

```

      , -estado, na.rm = T) %>%
tidyverse::separate(variable, c("indicador", "anio"))
      , extra = "drop") %>%
dplyr::mutate(indicador = paste0("pago_pensiones_", indicador))
all <- rbind(all, df)

all

## # A tibble: 1,122 x 5
##       entidad estado     indicador   anio   valor
##       <chr>    <chr>     <chr>    <chr>   <dbl>
## 1 Total nacional 00 pago_pensiones_imss 2000 41.2
## 2 Aguascalientes 01 pago_pensiones_imss 2000 41.5
## 3 Baja California 02 pago_pensiones_imss 2000 38.9
## 4 Baja California Sur 03 pago_pensiones_imss 2000 28.2
## 5 Campeche        04 pago_pensiones_imss 2000 37.5
## 6 Coahuila         05 pago_pensiones_imss 2000 46.5
## 7 Colima           06 pago_pensiones_imss 2000 37.9
## 8 Chiapas          07 pago_pensiones_imss 2000 34.9
## 9 Chihuahua        08 pago_pensiones_imss 2000 39.2
## 10 Ciudad de México 09 pago_pensiones_imss 2000 35.8
## # ... with 1,112 more rows

```

Visualización

La graficación es una manera eficiente de resumir, y mostrar información. Es fundamental entender el contexto de negocio y de la generación de los datos para tener más información con respecto a lo que se está graficando.

Aunque no hay mucha teoría alrededor de gráficos estadísticos, muchos se describen en distintos libros de texto de estadística (Unwin 2015, Introducción).

Unwin (2015, citando a John Tukey) resume el propósito de la visualización de datos en cuatro frases:

1. Las gráficas son para análisis cualitativos o descriptivos y quizás semi cuantitativos, nunca para análisis profundo cuantitativo (las tablas son mejores para esta tarea).
2. Las gráficas son para realizar comparaciones (en el tiempo, entre grupos), no para describir cantidades particulares.
3. Las gráficas son para impactar (visualmente, para sorprender, para transmitir información), pero casi nunca sirven para reflejar patrones escondidos en los datos.
4. Las gráficas deberían de reportar un análisis de datos trabajado, fino y cuidadoso. Jamás deben de reemplazar el análisis. Las gráficas están para fortalecer el análisis, no para fundamentarlo y los gráficos finales en un análisis deben reflejar el análisis

realizado.

Esta sección resume algunas de las funciones implementadas para **visualizar** datos implementados en el paquete `ggplot2` de R. Está fuera de la alcance de este apartado las consideraciones estadísticas que deben de realizarse en un análisis de datos pues el enfoque es mostrar cómo realizar esta tarea en R. En la figura 5.11 podemos ver la etapa del análisis de datos correspondiente.

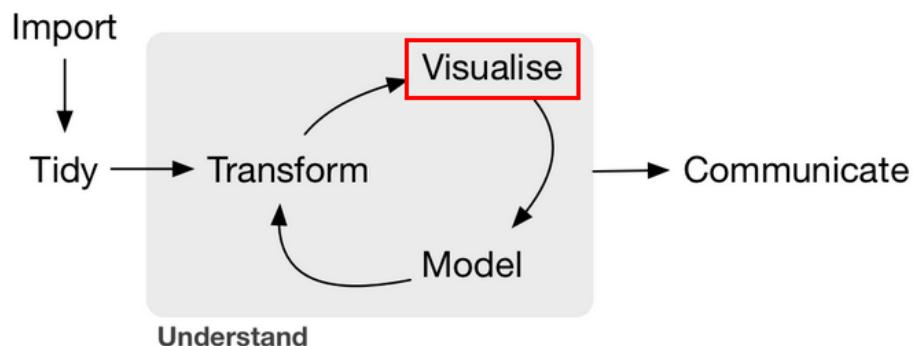


Figura 5.11: Visualización en el análisis de datos G. Grolemund y H. Wickham (2016, Introducción).

En R hay muchas maneras de realizar una tarea, esto es particularmente cierto en lo que se refiere a visualización de datos (Yau 2016). A fin de cuentas, lo más importante de la visualización de datos es cuán útil es esta herramienta para el análisis de datos y la forma en la que se le indica a la máquina como realizar la tarea es un elemento importante solo en la medida en la que facilite el trabajo del estadístico.

Aunque posiblemente cualquiera de los gráficos que se pueden hacer con `ggplot2` pueden hacerse también con los gráficos implementados en el `base graphics` de R (Yau 2016), aquí se cubre visualización en `ggplot2` principalmente debido a que:

- El patrón de programación está definido formalmente mientras que en el `base` no lo está. Esto significa que cuando entiendes cómo hacer una gráfica en `ggplot2`, puedes hacer una gran cantidad de gráficos (Yau 2016).
- `ggplot2` tiene una gran cantidad de *defaults* predefinidos que facilitan realizar gráficos ilustrativos y estéticos muy rápidamente (Mejia 2013).
- `ggplot2` es compatible con *piping*⁸ (Leek 2016).
- Muchas veces, resulta necesario realizar un mismo gráfico para varios subconjuntos de datos. Esta tarea se puede hacer en forma directa en `ggplot2` y no en el `base` a través de facetas.

Primero, se cubrirá el concepto detrás de `ggplot2`, la gramática de las gráficas. Posteriormente,

⁸Se introdujo el operador *pipe* de R en la sección de transformación en este capítulo y se mencionó como ayuda en la lectura y escritura de código. Un *pipeline* en programación consiste en un arreglo de elementos de procesamiento en donde la salida de cada elemento es la entrada del siguiente elemento. Este concepto fue concebido por Douglas McIlroy (Mahoney s.f.).

se describen los componentes de una gráfica en `ggplot2`, las capas y sus componentes proporcionando ejemplos en cada caso.

La gramática de las gráficas

Es una herramienta que nos permite (Hadley Wickham 2010):

- Describir los componentes de una gráfica en forma concisa
- Ir más allá de los nombres de la gráfica (e.g. scatterplot, boxplot, etc.)
- Entender la estructura detrás de los gráficos estadísticos

¿Qué es?

La gramática le da reglas al lenguaje y es un sistema formal para generar enunciados (Wilkinson 2006). Wilkinson (2006) proporciona una gramática para gráficos que permite describir y construir una gran cantidad de gráficos estadísticos. (Hadley Wickham 2010) implementa esta gramática en el paquete `ggplot2` de R (Hadley Wickham 2009).

La gramática define un gráfico estadístico como un mapeo de datos a atributos estéticos (un color, una forma) en objetos geométricos (barras, líneas, puntos). Además, una gráfica puede implicar transformaciones estadísticas de los datos y se dibuja sobre un sistema de coordenadas específico. Por último, es posible generar el mismo gráfico en diferentes subconjuntos de los datos (facetas). La combinación de estos factores independientes conforman un gráfico (Hadley Wickham 2009, p. 5).

La implementación en capas, permite que los usuarios no se limiten únicamente a los gráficos específicos que se implementan paquete a paquete sino que puedan realizar tantos gráficos como este lenguaje de gráficos permita.

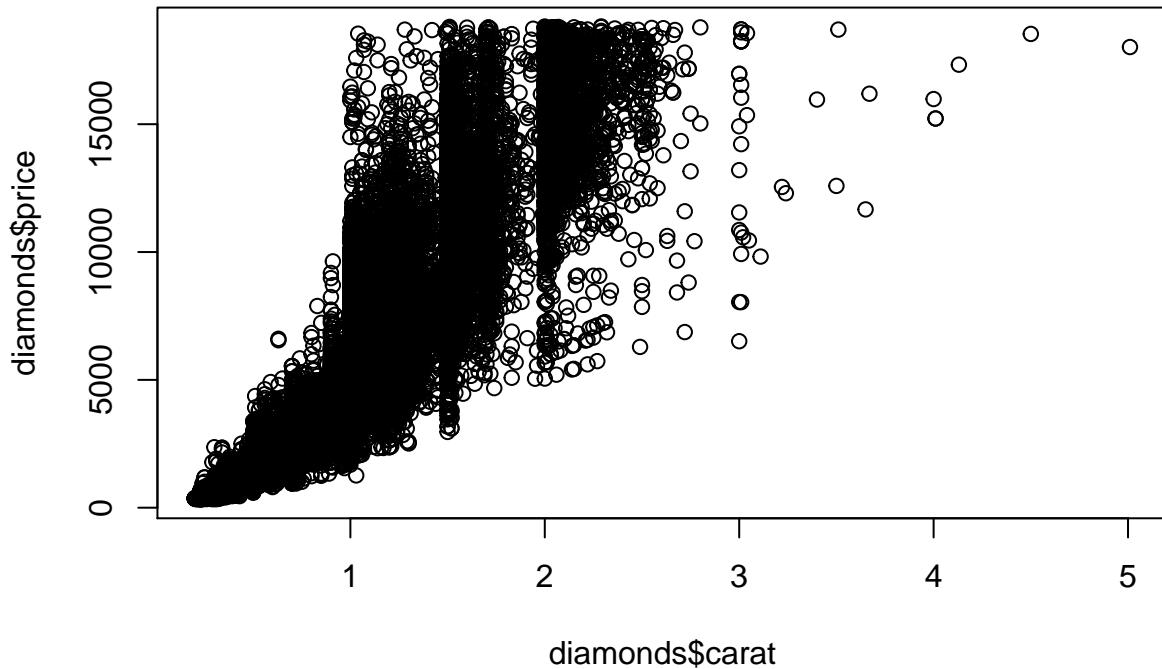
Base plotting vs. ggplot

Observemos una gráfica utilizando la base `diamonds` en donde graficamos en el eje *x* el carataje de diamantes y en el eje *y* su precio:

```
data(diamonds, package = "ggplot2")
str(diamonds)
```

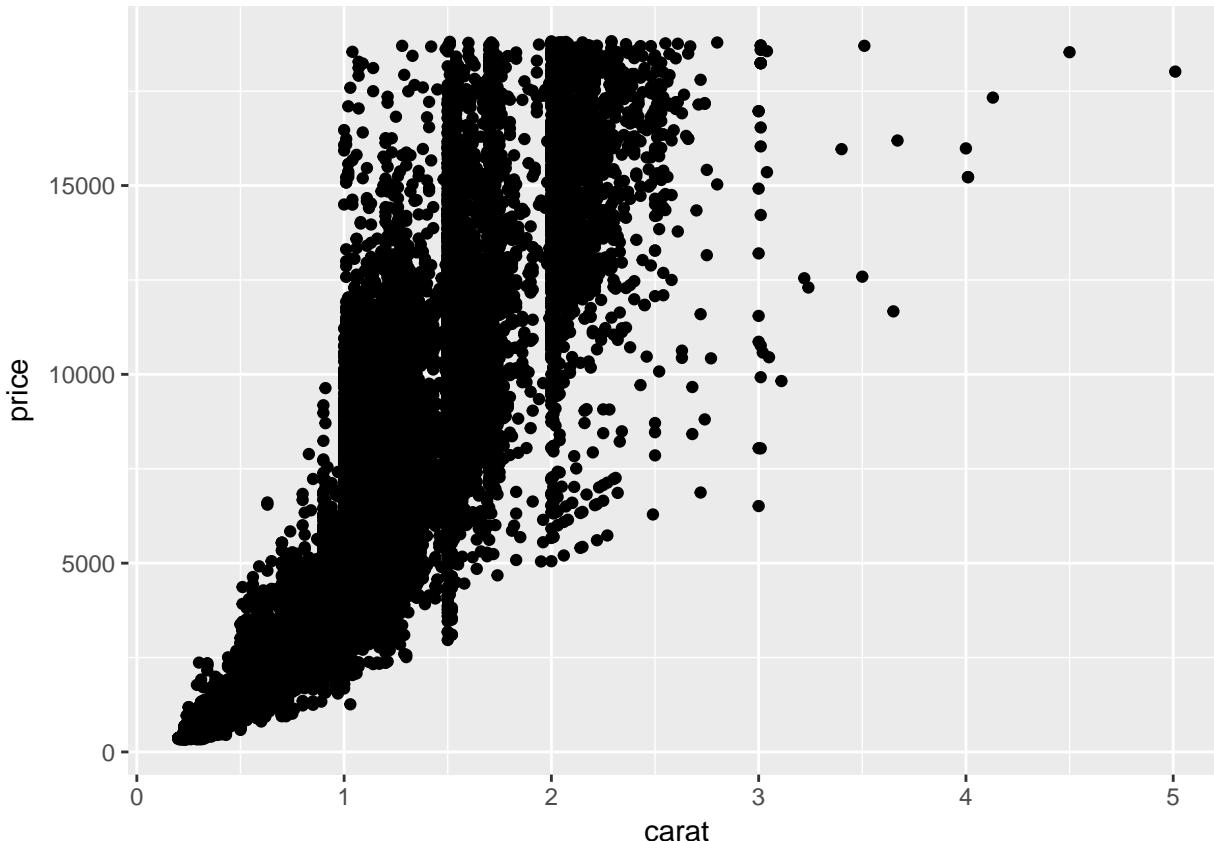
```
## Classes 'tbl_df', 'tbl' and 'data.frame':    53940 obs. of  10 variables:
##   $ carat  : num  0.23 0.21 0.23 0.29 0.31 0.24 0.24 0.26 0.22 0.23 ...
##   $ cut     : Ord.factor w/ 5 levels "Fair"<"Good"<...: 5 4 2 4 2 3 3 3 1 3 ...
##   $ color   : Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<...: 2 2 2 6 7 7 6 5 2 5 ...
##   $ clarity: Ord.factor w/ 8 levels "I1"<"SI2"<"SI1"<...: 2 3 5 4 2 6 7 3 4 5 ...
##   $ depth   : num  61.5 59.8 56.9 62.4 63.3 62.8 62.3 61.9 65.1 59.4 ...
##   $ table   : num  55 61 65 58 58 57 57 55 61 61 ...
```

```
## $ price  : int 326 326 327 334 335 336 336 337 337 338 ...
## $ x      : num 3.95 3.89 4.05 4.2 4.34 3.94 3.95 4.07 3.87 4 ...
## $ y      : num 3.98 3.84 4.07 4.23 4.35 3.96 3.98 4.11 3.78 4.05 ...
## $ z      : num 2.43 2.31 2.31 2.63 2.75 2.48 2.47 2.53 2.49 2.39 ...
plot(diamonds$carat, diamonds$price)
```



Ese mismo gráfico, podemos realizarlo con la función `qplot` del paqueteggplot:

```
qplot(data = diamonds, x = carat, y = price, geom = "point")
```



En donde especificamos que la variable `carat` en la base `diamonds` mapea al eje *x*, la variable `price` en la misma base al eje *y* y queremos que la geometría sea de puntos.

Observa que la estética de la gráfica generada con la función `plot` es un poco distinta a la generada con `qplot` (función que utiliza `ggplot`). Verifica la clase que tiene cada objeto, en el primer caso no tiene *y* en el segundo tenemos un objeto de clase `ggplot` que tiene atributos y que podemos guardar en un objeto.

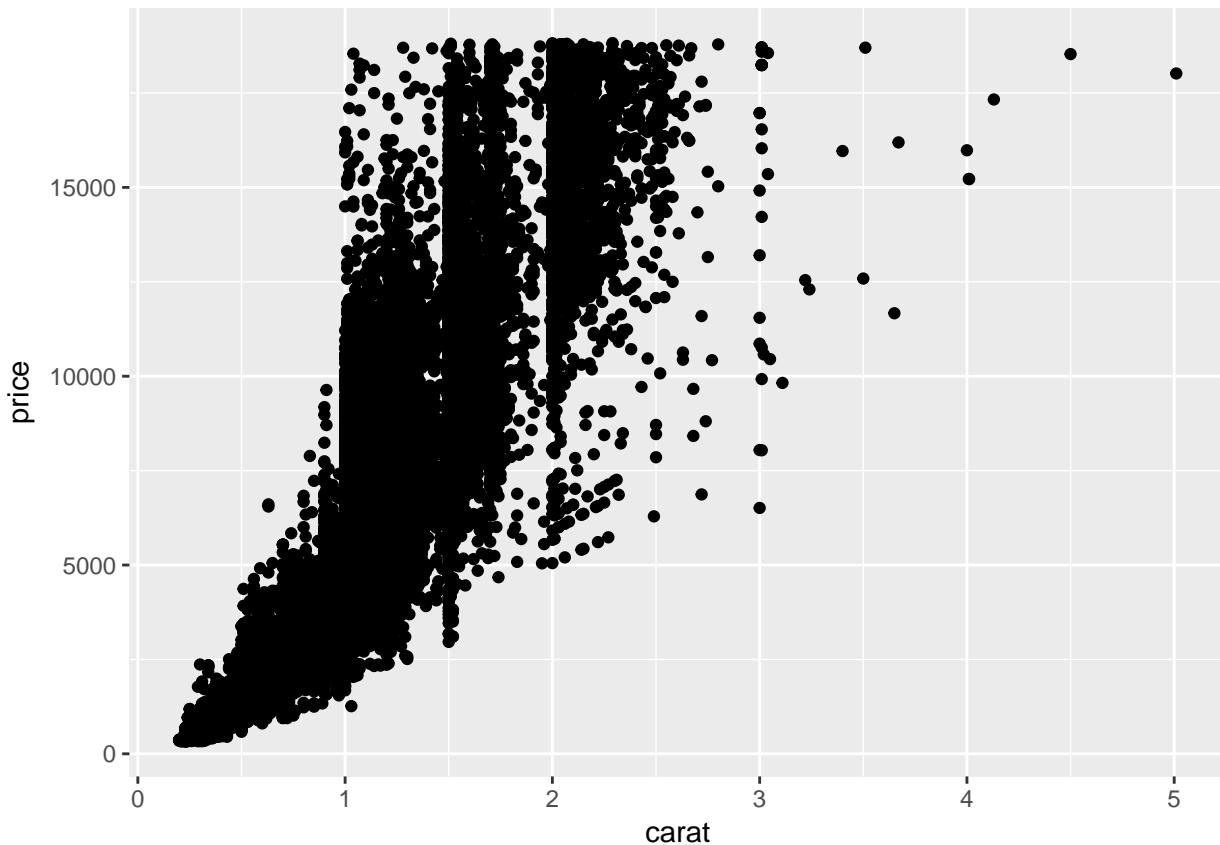
ggplot

Los componentes de la gramática de gráficas específica para `ggplot` son:

1. Una o más capas donde cada cuál tiene:
 - a. Datos
 - b. Mapeo estético de los datos (*aesthetic mappings*)
 - c. Un objeto geométrico
 - d. Una transformación estadística
 - e. Ajustes en las posiciones de los objetos
2. Una escala para cada estética
3. Un sistema de coordenadas
4. Una especificación de facetas

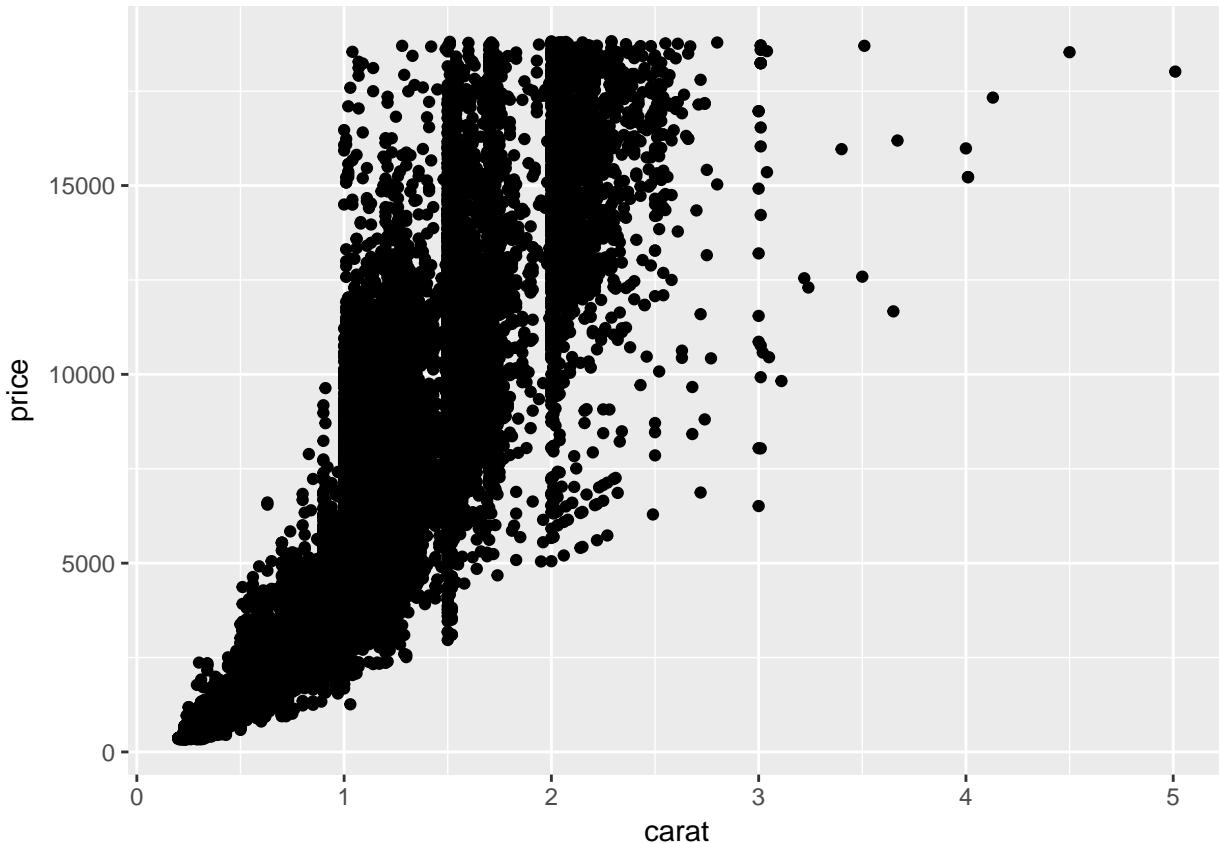
Al graficar en `ggplot` se tiene control sobre todos estos elementos:

```
ggplot() +
  layer( # una capa
    data = diamonds, # datos
    mapping = aes(x = carat, y = price), # mapeo estético
    geom = "point", # geometría
    stat = "identity", # transformación estadística
    position = "identity" # ajuste en posición de objetos
  ) +
  scale_y_continuous() + # Escala para estética continua en y
  scale_x_continuous() + # Escala para estética continua en x
  coord_cartesian() # Sistema de coordenadas
```



ggplot implementa también una serie de **defaults** (Hadley Wickham 2009, p. 3) que facilitan la escritura de nuevas gráficas pues no es necesario especificar cada uno de los detalles al agregar una capa. Por tanto, es posible escribir el mismo gráfico haciendo uso de esos defaults:

```
ggplot(diamonds, aes(carat, price)) + geom_point()
```



Características importantes

Los componentes de una gráfica son ortogonales:

- Cambiar uno no debe romper los otros
- Una configuración distinta de componentes es válida
- Puedes construir mayor complejidad agregando capas

Las capas

`ggplot` produce un objeto que se puede convertir en una gráfica. Es decir, R sabe cómo convertirlo en una gráfica.

Este objeto está formado por capas, mismas que tienen sus entradas (*inputs*) particulares y que comparten argumentos del gráfico `base` generado por la función `ggplot()`. Con el operador `+` se van agregando las distintas capas al mismo objeto.

Así como en otros casos, el objeto en R puede ser guardado en una variable, se le puede imprimir, se le puede guardar como imagen de diferentes formatos, o se puede guardar en una lista o en un `Rdata`.

Componentes de una capa

Datos y mapa estético

Permite mapear las columnas del `data.frame` de entrada a los aspectos de la gráfica.

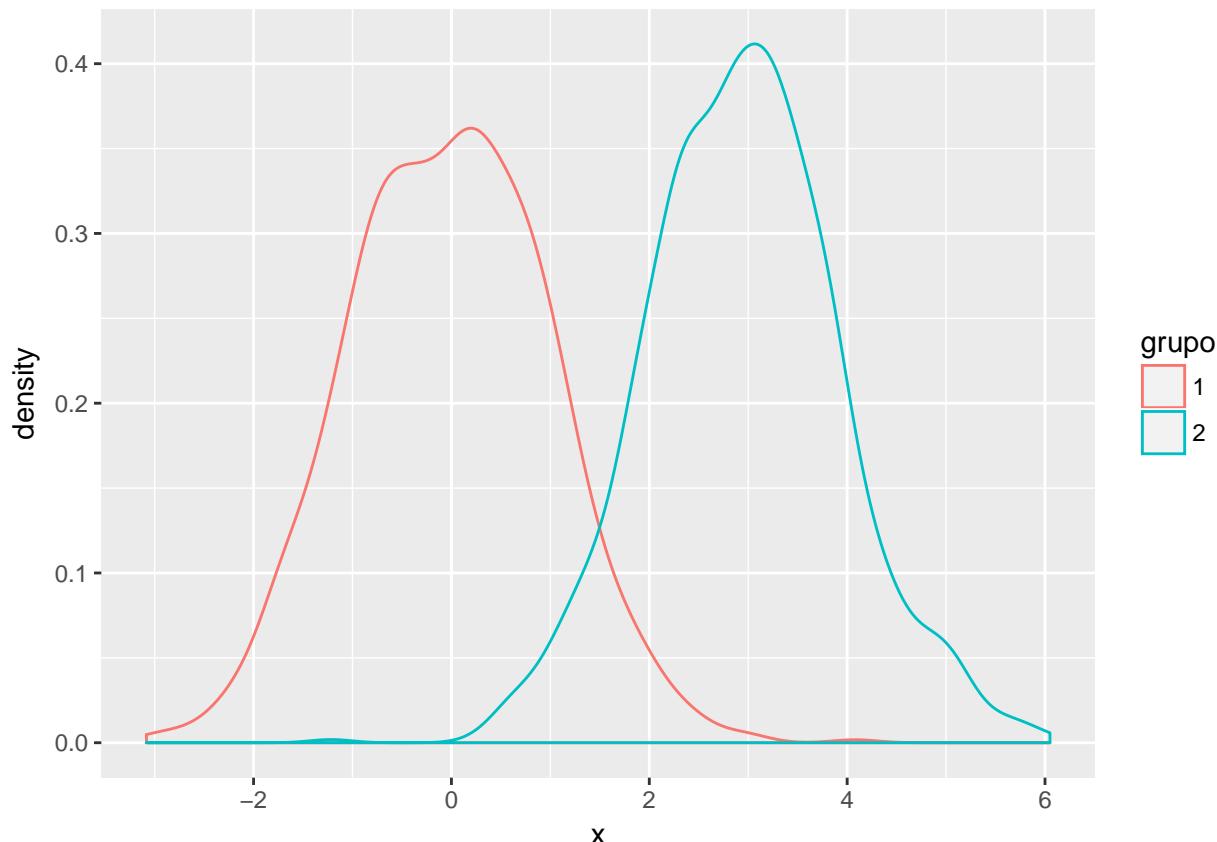
Es decir,

- las coordenadas x, y
- los grupos (definidos por otra variable)
- el tamaño
- el color
- el relleno

Para ejemplificar, generamos una variable x que proviene de dos distribuciones normales: mil realizaciones $N(0, 1)$ y mil $N(3, 1)$. Asignamos un grupo a las primeras mil y otro a la segunda.

```
mix2norm <- data.frame(x = c(rnorm(1000), rnorm(1000, 3)),
                        grupo = as.factor(rep(c(1,2),each=1000)))

ggplot(mix2norm, aes(x=x, color = grupo)) + geom_density()
```



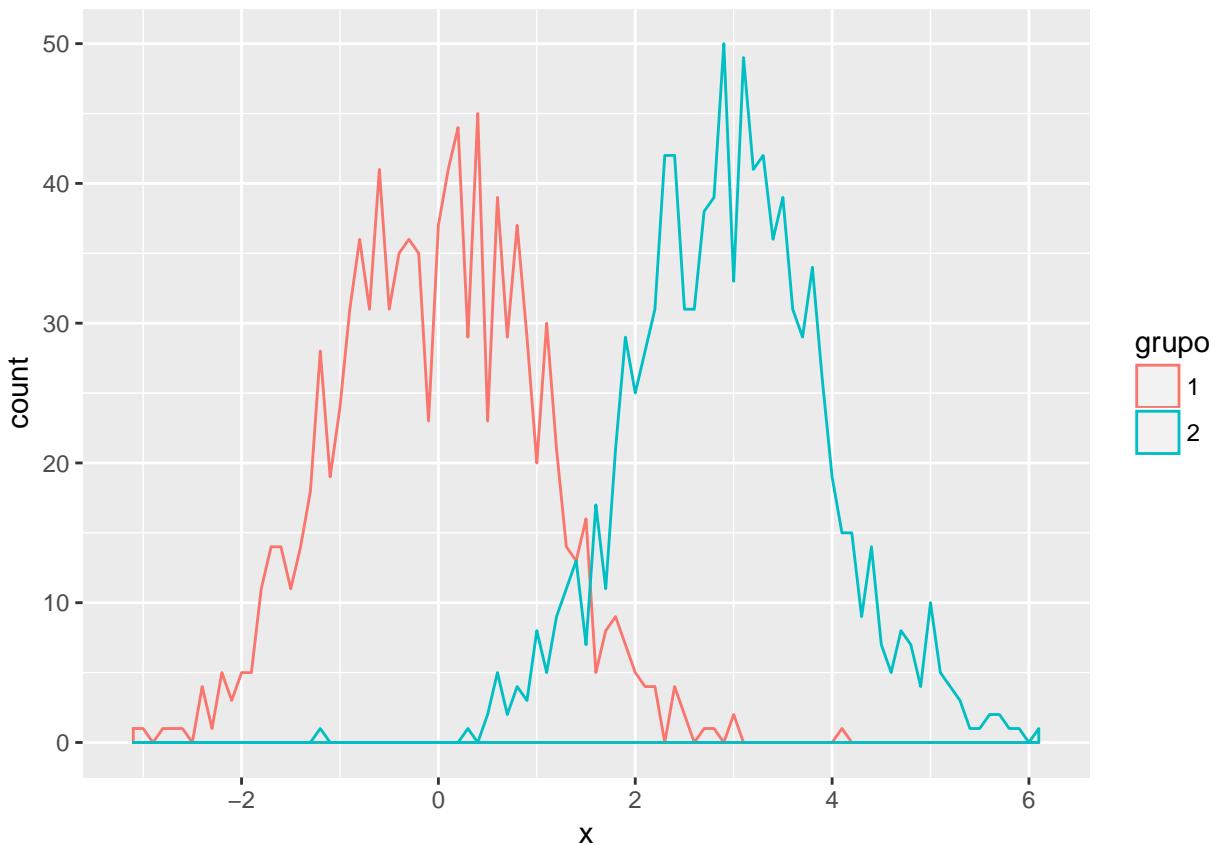
Transformaciones estadísticas

Ésta puede ser, por ejemplo, un resumen de la entrada (*input*) recibido; se especifica vía el comando **stat**. Ejemplos:

- binning
- smoothing
- boxplot
- identity

Utilizamos, por ejemplo, la transformación *bin*, misma que se especifica en el objeto geométrico:

```
ggplot(mix2norm, aes(x=x, color = grupo)) +
  geom_density(stat = "bin", binwidth = 0.1)
```



La transformación utilizada tiene asociada parámetros como lo es el tamaño en el que deben realizarse los colapsos de la variable categórica (*binwidth*).

El objeto geométrico

Esto permite especificar el tipo de gráfico a crear. Se especifica con la **geom**. Se define de acuerdo a su dimensión, es decir,

- 0-dim: puntos, texto

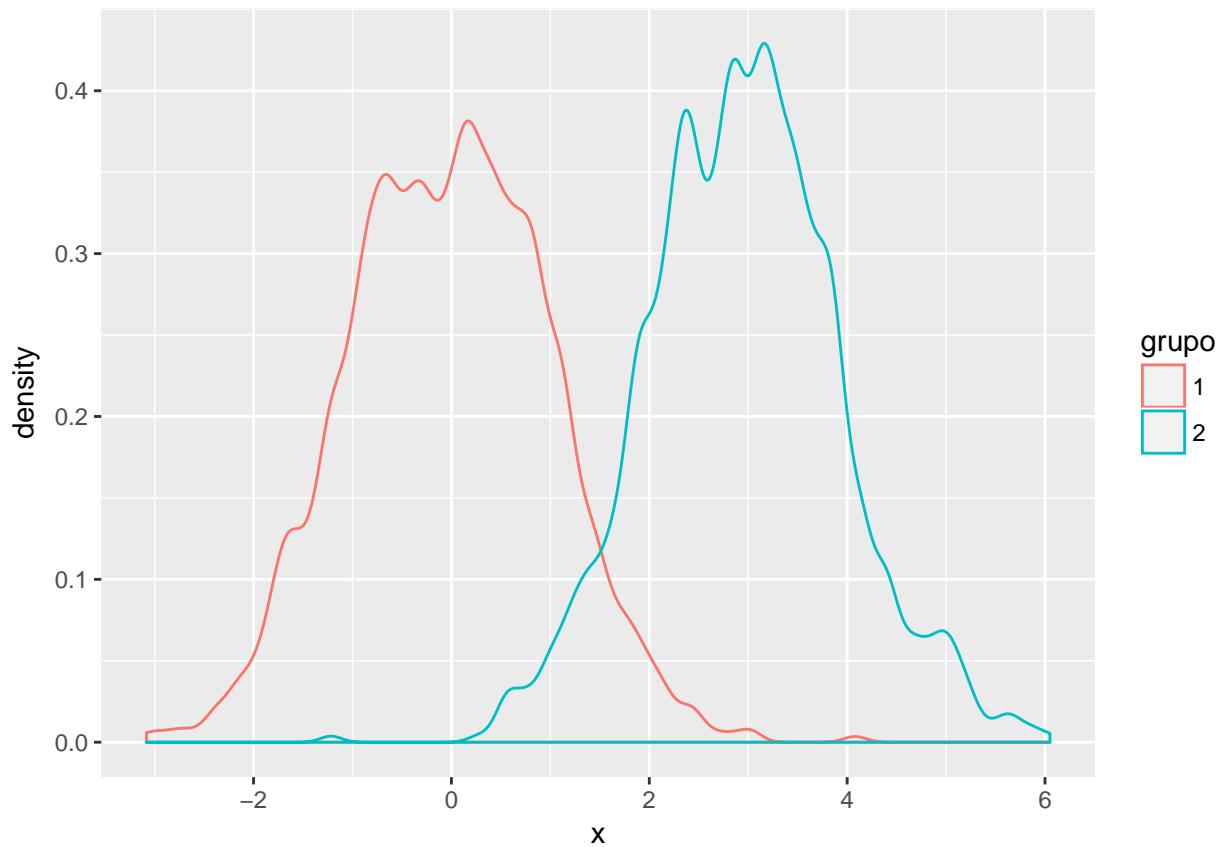
- 1-dim: líneas
- 2-dim: polígonos, intervalos

Otras geometrías incluyen:

- `geom_hist`
- `geom_bar`
- `geom_contour`
- `geom_line`
- `geom_density`

Además, se puede cambiar la transformación estadística manteniendo la geometría fijada. Al ejemplo anterior, le agregamos una transformación estadística dentro del objeto geométrico con el parámetro `adjust`.

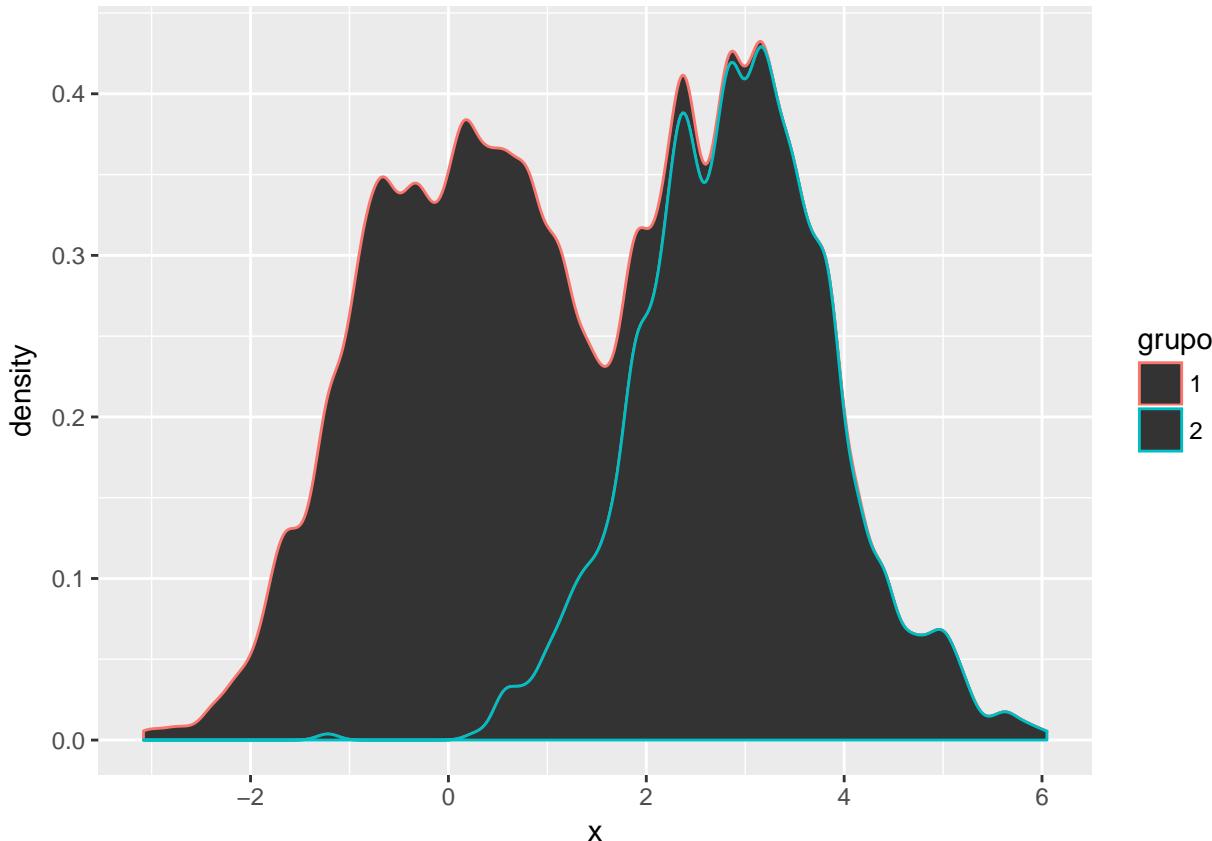
```
ggplot(mix2norm, aes(x = x, color = grupo)) + geom_density(adjust = 1/2)
```



En este caso, estamos pidiendo la mitad del tamaño del bin que se calcula en forma algorítmica por el paquete.

Viceversa, puede cambiarse la geometría pero mantener la transformación estadística.

```
ggplot(mix2norm, aes(x = x, color = grupo)) + stat_density(adjust = 1/2)
```



Revisa el comando `position` y `geometry`. Revisa sus defaults y copia algunos de los ejemplos en la documentación.

`geom_density` por ejemplo utiliza `ribbon` o una cosa que a veces encontrarán en español como violín.



Ejercicios

1. Genera una gráfica con la función `ggplot` en donde los datos sea la base `diamonds` y la estética sea `x = price`. Especifica como geometría una densidad.
2. Cambia el color y el relleno de la geometría a gris (`grey50`)
3. Cambia la geometría a `ribbon`, cambia los parámetros necesarios para que funcione.
4. Agrega una faceta para que se haga un gráfico para cada uno de los subconjuntos definidos por la variable `cut`.
5. Agrega a la gráfica el comando `coord_flip` para que el precio este en el eje `y`.

```
# Respuestas
# 1
g <- ggplot(diamonds, aes(x = price)) + stat_density()
g
# 2
g +
  stat_density(fill = "grey50", colour = "grey50")
# 3
```

```

g <- g +
  stat_density(aes(ymax = ..density.., ymin = -..density..),
               fill = "grey50", colour = "grey50",
               geom = "ribbon", position = "identity")
g
# 4
g <- g +
  facet_grid(. ~ cut)
g
# 5
g + coord_flip()

```

Posición

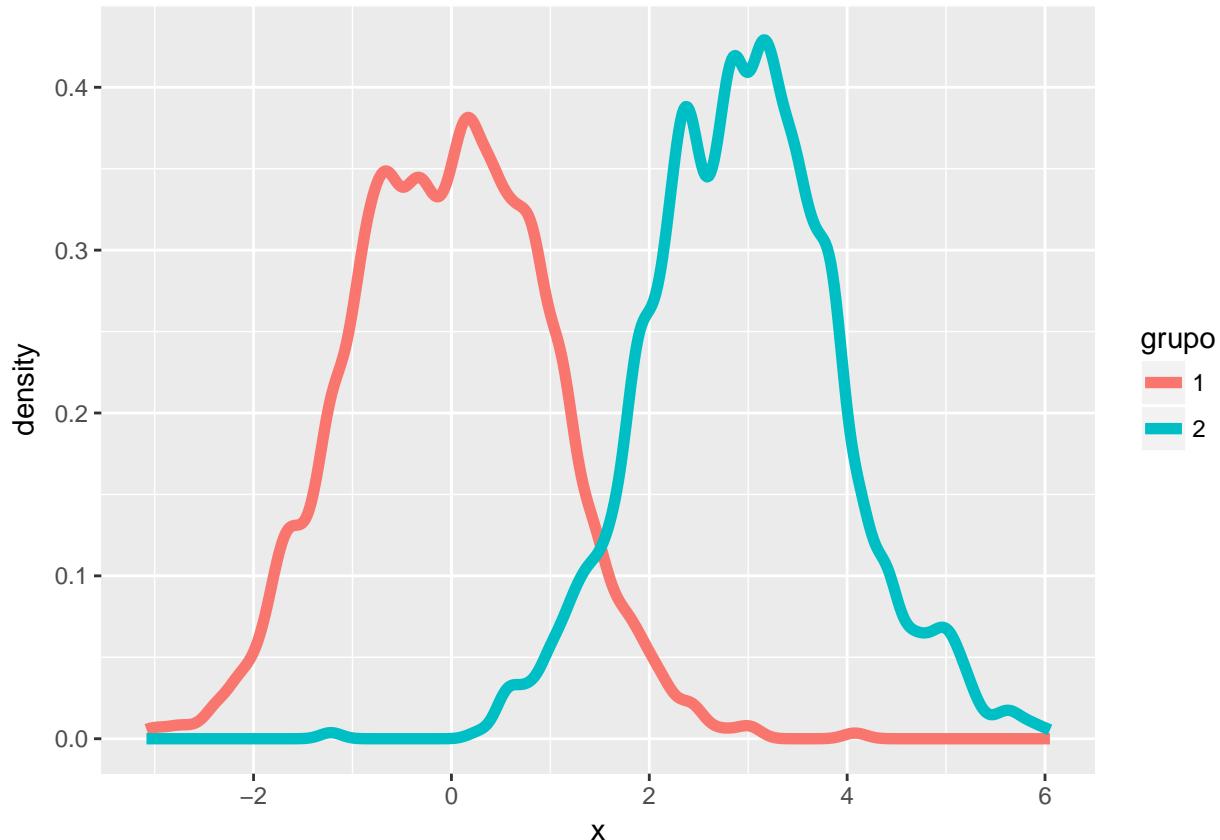
Es posible especificar la posición de cada una de las capas en relación a otras. Ejemplos:

- dodge
- identity
- jitter

```

ggplot(mix2norm, aes(x=x, color = grupo)) +
  stat_density(adjust=1/2, size=2, position = "identity", geom = "line")

```

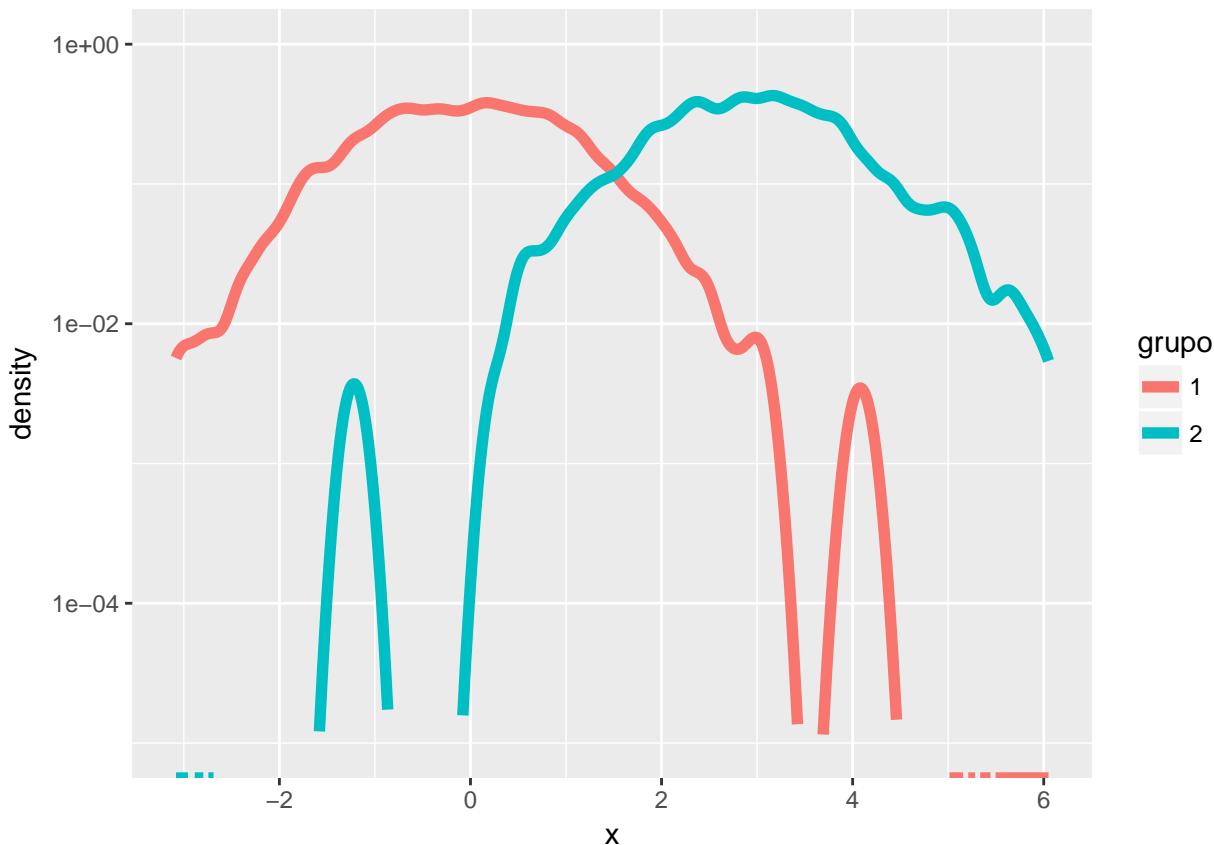


Escalas

Determina cuál valor de entrada mapea a qué estética específica. Se escribe usando `scale`. Hay de todo:

- `continuous`
- `logarithmic`
- `values to shapes`
- `what limits`
- `what labels`
- `what marks`

```
ggplot(mix2norm, aes(x=x, color = grupo)) +
  stat_density(adjust=1/2, size=2, position ="identity", geom ="line") +
  scale_y_log10(limits = c(1e-5,1))
```



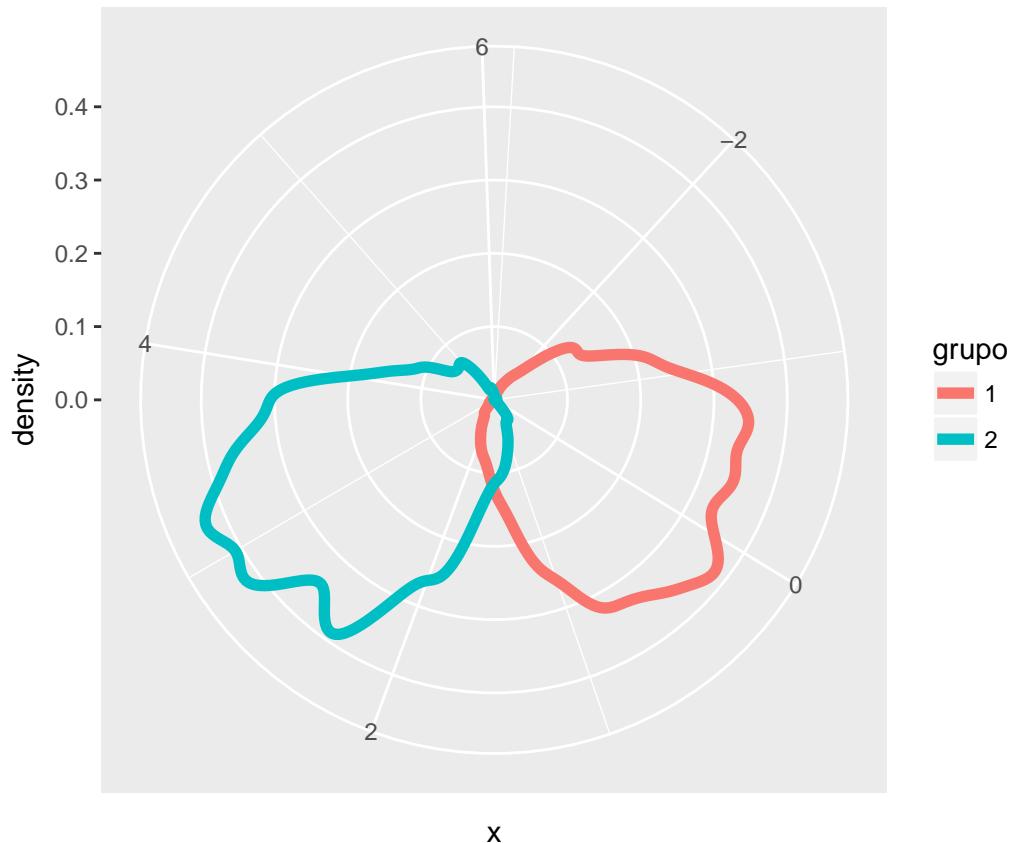
Coordenadas

Te permite especificar las posiciones de las cosas y cómo mapean a las posiciones en la pantalla. Antes todo era entorno a cómo le dices las cosas a R pero también importa cómo las ves. Coordenadas distintas pueden afectar a los objetos geométricos. Ejemplos:

- `cartesian`

- polar
- map-projection

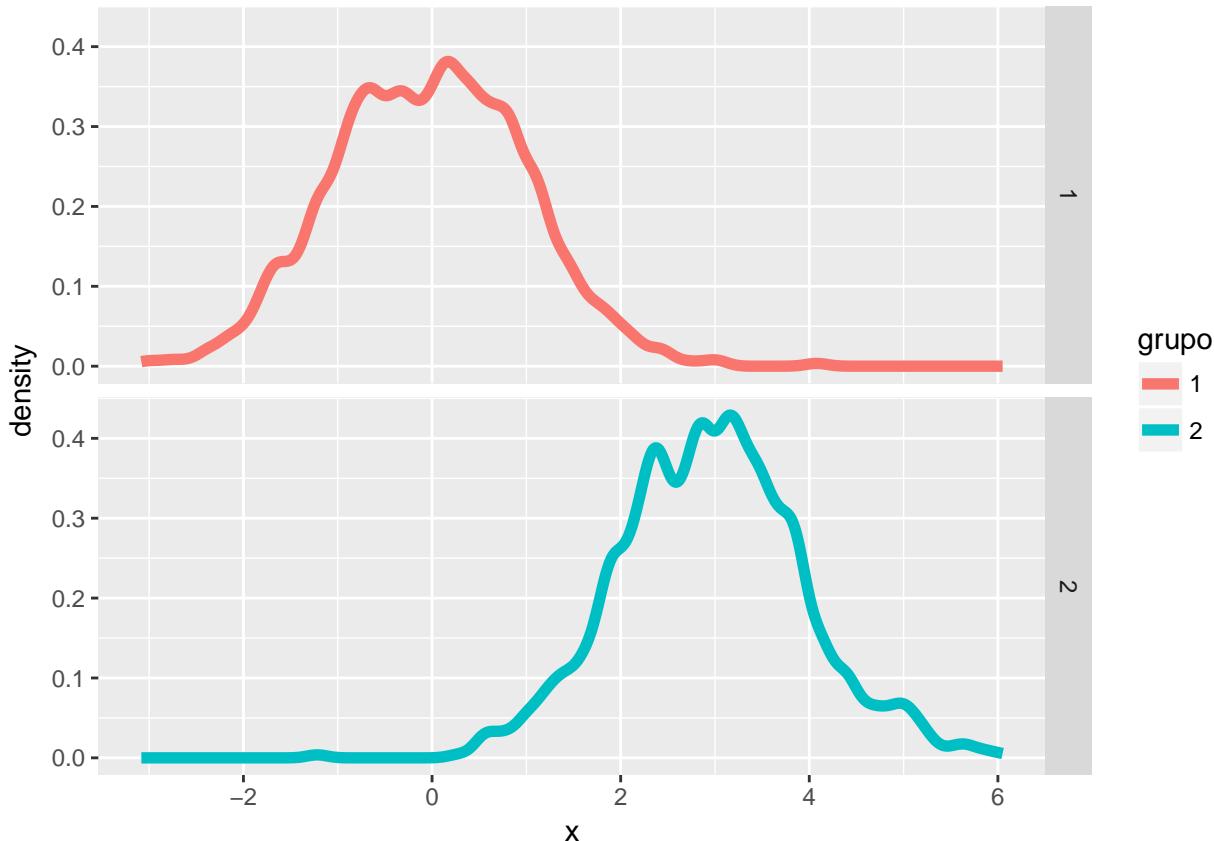
```
ggplot(mix2norm, aes(x = x, color = grupo)) +  
  stat_density(adjust = 1/2, size = 2, position = "identity", geom = "line") +  
  coord_polar()
```



Facetas

Permite arreglar diferentes gráficas en un grid o panel.

```
ggplot(mix2norm, aes(x = x, color = grupo)) +  
  stat_density(adjust = 1/2, size = 2, position = "identity", geom = "line") +  
  facet_grid(grupo ~ .)
```



Ve el help de `facet_wrap`

Material adicional

- Importación
 - Curso **Importing Data in R (Part 1)** de Data Camp.
 - Curso **Importing Data in R (Part 2)** de Data Camp.
- `dplyr` y `tidyverse`
 - Curso de `swirl` **Getting and cleaning data**.
 - Curso **Cleaning Data in R** de Data Camp.
 - Curso **Data Manipulation in R with dplyr** de Data Camp.
 - Curso **Joining data in R with dplyr** de Data Camp.
- Gráficos del `base`
 - Curso de `swirl` **Overview of Statistics**.
- `ggplot2`
 - Curso de `swirl` **Exploratory data analysis**.
 - Curso **Data Visualization in R** de Data Camp.
 - Curso **Data Visualization with ggplot2 (Part 1)** de Data Camp.
 - Curso **Data Visualization with ggplot2 (Part 2)** de Data Camp.
 - Curso **Data Visualization with ggplot2 (Part 3)** de Data Camp.

Capítulo 6

Conclusión

Este manual surge de la necesidad detectada de contar con material en español que permita al usuario aprender a operar un proyecto de datos, más allá del aprendizaje del lenguaje de programación. A través de cada capítulo, se espera que el lector se familiarice con los elementos básicos del ambiente de R, sus más importantes estructuras de datos, así como distintas maneras de operarlas. Sin embargo, el enfoque del manual es que el lector cuente con las herramientas necesarias para poder desarrollar un proyecto de datos.

El manual está estructurado para que en los primeros tres capítulos el lector conozca las principales herramientas que facilitan el trabajo del analista a través del dominio básico de la sintaxis de R, para, posteriormente, en el capítulo 4, centrarse en el ciclo de análisis de datos.

Para éste ciclo, se revisaron los métodos de importación de datos al ambiente de R, la manipulación y limpieza de datos, revisando a fondo los verbos implementados en `dplyr` y el concepto de datos limpios, así como su implementación en `tidyverse`. Por último, para concluir el ciclo de análisis se mostró una introducción a la gramática de gráficas y su implementación en `ggplot2`. Se utilizó la conjugación de los verbos de `dplyr`, pues estos permiten realizar casi todas las operaciones que un analista debe realizar para preparar sus datos para el modelado, facilitando su visualización o presentación en formato tabular, así como permitir la realización de estadística descriptiva.

Por último, con la finalidad de que el lector pueda practicar lo aprendido se incluyeron múltiples ejercicios y referencias a otro tipo de herramientas que salen del alcance de este manual pero que complementan al material presentado, como lo son, las expresiones regulares, el trabajo con fechas, la manipulación de cadenas de caracteres, entre otros.

A través de diversas iteraciones con distintos grupos, el material ha probado ser útil como guía en varios cursos introductorios a R, cumpliendo con el objetivo o alcance del mismo.

Con la finalidad de fortalecer el presente manual, así como seguir construyendo material en español que permita el desarrollo de proyectos de datos cada vez más sólidos, es posible ampliar el material de manera que se cubran distintas herramientas para el modelado, particularmente las implementadas en `purrr` que continúan con el *framework* de datos limpios para esta etapa en el ciclo de análisis; así como ahondar en el apartado de comunicación, que solo se cubrió

en el capítulo 2 presentando **rmarkdown** para compilar documentos reproducibles, así como traducir los cursos de **swirl** a los que se hace referencia.

En este sentido, se invita a que el lector proporcione comentarios y sugerencias para mejorar o ampliar el material a través de *issues* o *pull requests* en el repositorio del proyecto: <https://github.com/animalito/aprendeR>.

Apéndice A

Markdown

Estamos acostumbrados a editores del tipo *what you see is what you get*. Markdown permite escribir contenidos en **texto plano** con una sintaxis para darle formato. Sobre esta sintaxis la herramienta de software en Perl convierte el texto plano a HTML (Gruber 2012).

Tiene un alfabeto y símbolos que, una vez procesados, se ven de cierta forma. A diferencia de un editor como Word en el que la versión de cada computadora cambia la manera en la que se ven los documentos, con Markdown esto no sucede.

Este lenguaje se está volviendo cada vez más común y, en particular, páginas como GitHub y reddit lo utilizan para sus comentarios.

La curva de aprendizaje es mínima. En lo que se va memorizando el alfabeto de markdown, lo más útil es tener una lista de fácil acceso con los caracteres más comunes y lo que hacen. A continuación, se proporciona un listado de la sintaxis más común ¹.

Encabezados

```
# Nivel 1  
## Nivel 2  
### Nivel 3  
#### Nivel 4  
  
... y así
```

¹Basado en Doxygen (2012) y en Pritchard (2012).

Nivel 1

Nivel 2

Nivel 3

Nivel 4

Lineas horizontales

Con tres o mas de los siguientes

Guiones

Asteriscos

Guiones bajos

Guiones

Asteriscos

Guiones bajos

Énfasis

italica o _italica_
negritas o __negritas__
**_combinado_*
**Uno en negritas _el otro combinado_*
~~tachar~~

- *italica* o *italica*
- **negritas** o **negritas**

- *combinado*
- Uno en negritas *el otro combinado*
- tachar

Bloques

> Ejemplo: Un bloque de varias
> líneas.

Ejemplo: Un bloque de varias líneas.

Listas

1. Primero
2. Segundo
 - Primer elemento del segundo
 - Segundo elemento del segundo
7. No tengo que cambiar el nombre, se pondrá el correcto
 - i. Una sublista con incisos
 - ii. Más incisos
4. Más cosas
 - a. Otra cosa
 - b. Una más
5. Otra manera de hacer listas no ordenadas
 - * Usando asteriscos
 - O usando menos
 - O usando el signo de mas
1. Primero
2. Segundo
 - Primer elemento del segundo
 - Segundo elemento del segundo
3. No tengo que cambiar el nombre, se pondrá el correcto
 - I. Una sublista con incisos
 - II. Más incisos
4. Más cosas
 - a. Otra cosa
 - b. Una más
5. Otra manera de hacer listas no ordenadas
 - Usando asteriscos
 - O usando menos
 - O usando el signo de mas

Links

[Esto es lo que se ve](<https://www.google.com>)

[Esto es lo que se ve y el mensaje "Google" aparece en el hover]

(<https://www.google.com> "Google")

[Puedo hacer referencia a un archivo local](readme.md)

[Puedes poner referencias] [1]

Detecta urls completos www.google.com o google.com

Y luego pones a donde te lleva la referencia

[1]: https://en.wikipedia.org/wiki/42_%28number%29#The_Hitchhiker.27s_Guide_to_the_Galaxy

Esto es lo que se ve

Esto es lo que se ve y el mensaje "Google" aparece en el hover

Puedo hacer referencia a un archivo local

Puedes poner referencias con numeros

Luego quieres hacer referencias a otra parte del documento por lo tanto, puedes ligar texto

Detecta urls completos http://www.google.com

Y luego pones a dónde te lleva la referencia

Nota que los espacios entre lineas son importantes.

Imágenes

Puedes ponerlo en una misma linea:

! [alt text] (dw.png "Es lo mejor")

Reference-style:

! [alt text] [logo]

[logo]: dw.png "Sin duda alguna"



Puedes ponerlo en una misma linea:



Reference-style:

Tablas

Una tabla simple

| Encabezado 1 | Encabezado dos |
|--------------|----------------|
| ----- | ----- |
| Contenido | Contenido |
| Contenido | Contenido |

| Encabezado 1 | Encabezado dos |
|--------------|----------------|
| Contenido | Contenido |
| Contenido | Contenido |

Una tabla alineada. La alineacion la puedes hacer por columnas

| Derecha | Centro | Iquierda |
|---------|--------|----------|
| -----: | -----: | ----- |
| 10 | 10 | 10 |
| 1000 | 1000 | 1000 |

| Derecha | Centro | Izquierda |
|---------|--------|-----------|
| 10 | 10 | 10 |
| 1000 | 1000 | 1000 |

HTML

Si sabes html, puedes utilizarlo directo.

```
<dl>
  <dt>Lista de definiciones</dt>
  <dd> Una def.</dd>

  <dt>Markdown en HTML</dt>
  <dd>No *siempre* funciona **bien**. Puro HTML <em>para que funcione</em>. </dd>
</dl>
```

Puedes controlar mejor las imágenes

```

```

Lista de definiciones

Una def.

Markdown en HTML

No *siempre* funciona muy **bien**. Usa puro HTML para que funcione siempre.

Puedes controlar mejor las imágenes

Código

En una linea puedo poner `código`

En una linea puedo poner código.

Para poner bloques de código se utilizan tres acentos invertidos y se especifica el lenguaje.

```
import requests
```

```
print "¡Hola mundo!"
```

```
select * from tabla;
```

```
library(dplyr)
```

Sin lenguaje, no lo resalta

Párrafos

Si

escribo así

me lo junta todo.

Debo separar para iniciar otro parrafo.

Si escribo así me lo junta todo.

Debo separar para iniciar otro párrafo.

Apéndice B

Packrat

Uno de los problemas en el trabajo colaborativo es poder ejecutar código realizado en otra computadora. La analítica reproducible y fácil de insertar en un ambiente de producción es fundamental para minimizar el retrabajo y que lo que se realice se (re)utilice.

Existen múltiples maneras de trabajar de manera que se resuelva el problema de las versiones de software y sus dependencias. Una comprehensiva, por ejemplo, es usando `docker`. Cuando un proyecto incluye únicamente código de R, `packrat` es suficiente para empaquetarlo y que el código sea reproducible en cualquier computadora y sistema operativo (Ushey y col. 2016).

Packrat es un sistema de administración de dependencias para R que busca eliminar los problemas que suele haber para utilizar código realizado en diferentes momentos o máquinas con diferentes versiones de las librerías, entre los típicos son (Inc. 2014):

- La falta de control sobre los paquetes que se necesitaban instalar para correr un script específico.
- Instalar paquetes en el ambiente global y dejarlos para siempre instalados en las computadoras porque no se sabe si algo se romperá al quitarlos.
- Romper código de otros proyectos por actualizar un paquete en otro.

Packrat permite que los proyectos en R sean (Inc. 2014):

- **Aislados:** cada proyecto tiene su paquetería privada.
- **Portables:** puedes transferir rápidamente los proyectos de una computadora a otra -y a través de distintas plataformas- pues facilita la instalación de toda la paquetería sobre la que descansa el proyecto.
- **Reproducibles:** guarda las versiones exactas sobre las que el proyecto fue trabajado y éstos son los que son instalados en cualquier ambiente.

El directorio del proyecto

Packrat se asocia a un directorio específico. Al iniciar una sesión de R dentro de un directorio asociado a un packrat, R va a utilizar únicamente los paquetes dentro de esa librería privada. Al instalar, remover o actualizar un paquete dentro de ese directorio, esos cambios se harán en la librería privada.

Se guarda en el proyecto toda la información que packrat necesita para poder recrear el conjunto de librerías en cualquier otra máquina.

Instalación

El paquete está en el CRAN y se instala desde R con el comando.

```
install.packages("packrat")
```

Inicializarlo

Al iniciar un proyecto, el que sea, que use R, lo recomendable es asociarle packrat. Esto se hace con el comando `packrat::init`.

```
packrat::init("~/prueba-packrat")
```

Con esto, al trabajar en el directorio `"~/prueba-packrat"` ya estás en un proyecto de packrat con su librería privada.

Un proyecto de packrat se distingue porque -igual que git- tiene archivos y directorios adicionales que se crean con la función `init()`:

- `packrat/packrat.lock`: lista las versiones de los paquetes que fueron utilizadas. Este archivo no debe editarse a mano.
- `packrat/packrat.opts`: guarda las opciones de configuración para el proyecto. Este se puede modificar con las opciones `get_opts` y `set_opts`. La lista completa de opciones se puede ver al escribir en la consola de R `?"packrat-options"`.
- `packrat/lib/`: paquetes para el proyecto.
- `packrat/src/`: paquetes para todas las dependencias.
- `.Rprofile`: Le dice a R que la lista específica de librerías que debe utilizar cuando está en ese directorio (o cualquiera de sus subdirectorios) es la privada del proyecto que gestiona packrat.

Agregar, remover y actualizar paquetes

1. Inicializa R dentro de un proyecto packrat.
2. Se instala como siempre, usando `install.packages()`

```
install.packages("dplyr")
```

3. Se toma un `snapshot` para decirle a packrat que guarde los cambios

```
packrat::snapshot()
```

Aquí `packrat` agrega lo que necesita a los folders mencionados antes para poder recrear las versiones y dependencias. También modifica el archivo `packrat.lock`.

4. En cualquier momento, puedes revisar el estatus

```
packrat::status()
```

Debe darte el mensaje **Up to date**.

¡Y listo!

Otras cosas importantes

- Packrat puede incorporar paquetes que no están en CRAN.
- Puede restaurar un snapshot (de manera similar a la que un juego puedes regresar al último checkpoint).



Ejercicio: Restaurando un snapshot en el ejemplo de juguete.

1. Ve a la carpeta `/prueba-packrat`
2. Muevete a la carpeta `packrat/`
3. Borra la librería `rm -R lib/`
4. Regresa a la carpeta del proyecto `cd ..`
5. Inicializa R... y todo se restaura.

Ligas utiles

- Comandos comunes y listas de opciones
- Packrat en RStudio lo hace AUN mas fácil.
- Limitaciones de packrat

Bibliografía

- [Lik32] Rensis Likert. «A technique for the measurement of attitudes.» En: *Archives of psychology* (1932).
- [IG96] Ross Ihaka y Robert Gentleman. «R: a language for data analysis and graphics». En: *Journal of computational and graphical statistics* 5.3 (1996), págs. 299-314.
- [HT99] Andrew Hunt y David Thomas. *The pragmatic programmer*. Addison Wesley, 1999.
- [Ros+04] Anthony J Rossini y col. «Emacs speaks statistics: A multiplatform, multipackage development environment for statistical analysis». En: *Journal of Computational and Graphical Statistics* 13.1 (2004), págs. 247-261.
- [MGN16] MGN. *Marco geoestadístico nacional*. Último acceso: 2016-12-09. INEGI, 2005-2016.
- [Wil06] Leland Wilkinson. *The grammar of graphics*. Springer Science & Business Media, 2006.
- [Mof09] CL Moffat. *Visual Representation of SQL Joins*. Blog. 2009. URL: <https://www.codeproject.com/Articles/33052/Visual-Representation-of-SQL-Joins>.
- [Wic09] Hadley Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2009. ISBN: 978-0-387-98140-6. URL: <http://ggplot2.org>.
- [Adl10] Joseph Adler. *R in a nutshell: A desktop quick reference*. " O'Reilly Media, Inc.", 2010.
- [Edd10] Dirk Eddelbuettel. «Benchmarking single-and multi-core BLAS implementations and GPUs for use with R». En: *Mathematica* (2010).
- [McK10] Wes McKinney. *pandas: Python Data Analysis Library*. 2010. URL: <http://pandas.pydata.org/>.
- [Wic10] Hadley Wickham. «A layered grammar of graphics». En: *Journal of Computational and Graphical Statistics* 19.1 (2010), págs. 3-28.
- [Wic11] Hadley Wickham. «The split-apply-combine strategy for data analysis». En: *Journal of Statistical Software* 40.1 (2011), págs. 1-29.
- [Dox12] Doxygen. *Markdown support*. <https://www.stack.nl/~dimitri/doxygen/manual/markdown.html>. Manual. 2012.
- [Gru12] John Gruber. *Optimized R and Python: standard BLAS vs. ATLAS vs. OpenBLAS vs. MKL*. <https://daringfireball.net/projects/markdown/>. Manual. 2012.
- [Pri12] Adam Pritchard. *Markdown Cheatsheet*. <https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet>. Reference. 2012.

- [Gan13] Christopher Gandrud. *Reproducible research with R and R studio*. CRC Press, 2013.
- [Mej13] Mandy Mejia. *10 reasons to switch to ggplot*. <https://mandymejia.wordpress.com/2013/11/13/10-reasons-to-switch-to-ggplot-7/>. Blog. 2013.
- [BW14] Stefan Milton Bache y Hadley Wickham. *magrittr: A Forward-Pipe Operator for R*. R package version 1.5. 2014. URL: <https://CRAN.R-project.org/package=magrittr>.
- [Gal14] Tal Galili. *Simpler R coding with pipes: the present and future of the magrittr package*. <https://www.r-bloggers.com/simpler-r-coding-with-pipes-the-present-and-future-of-the-magrittr-package/>. Blog. 5 de ago. de 2014.
- [Inc14] RStudio Inc. *Packrat: Reproducible package management for R*. <https://rstudio.github.io/packrat/>. Blog. 2014.
- [Kla14] Brett Klamer. *Faster BLAS in R*. <http://brettklamer.com/diversions/statistical/faster-blas-in-r/>. Blog. 16 de nov. de 2014.
- [Ngu14] Vinh Nguyen. *Optimized R and Python: standard BLAS vs. ATLAS vs. Open-BLAS vs. MKL*. <http://blog.nguyenvq.com/blog/2014/11/10/optimized-r-and-python-standard-blas-vs-atlas-vs-openblas-vs-mkl/>. Blog. 10 de nov. de 2014.
- [Vai14] Ramnath Vaidyanathan. *Introduction to R*. Github books, en pycon 2014, 2014. URL: <https://ramnathv.github.io/pycon2014-r/>.
- [Wic14a] Hadley Wickham. *Advanced R*. CRC Press, 2014.
- [Wic+14] Hadley Wickham y col. «Tidy data». En: *Journal of Statistical Software* 59.10 (2014), págs. 1-23.
- [Wic14b] Hadley Wickham. *tidy-data*. <https://github.com/hadley/tidy-data>. 2014.
- [Dow+15] M Dowle y col. *data.table: Extension of Data.frame*. R package version 1.9.6. 2015. URL: <https://CRAN.R-project.org/package=data.table>.
- [Esc15] MA Escalante. *Instalación de R compilado en Ubuntu*. https://github.com/Skalas/massive-adventure-ubuntu/blob/master/i_R.sh. Repositorio. 2015.
- [Kab15] Robert Kabacoff. *R in action: data analysis and graphics with R*. Manning Publications Co., 2015.
- [Ort15] T Ortiz. *Manipulación y agrupación de datos*. https://dl.dropboxusercontent.com/u/1351973/tutoriales/intro_r_2.html. Blog. 2015.
- [RK15] Tyler W. Rinker y Dason Kurkiewicz. *pacman: Package Management for R*. version 0.4.1. University at Buffalo/SUNY. Buffalo, New York, 2015. URL: <http://github.com/trinker/pacman>.
- [Unw15] Antony Unwin. *Graphical data analysis with R*. Vol. 27. CRC Press, 2015.
- [Wic15a] Hadley Wickham. *pryr: Tools for Computing on the Language*. R package version 0.1.2. 2015. URL: <https://CRAN.R-project.org/package=pryr>.
- [Wic15b] Hadley Wickham. *R packages*. "O'Reilly Media, Inc.", 2015.
- [All16] JJ Allaire. *flexdashboard: R Markdown Format for Flexible Dashboards*. R package version 0.3. 2016. URL: <https://CRAN.R-project.org/package=flexdashboard>.
- [CRA16] CRAN. *The Comprehensive R Archive Network*. R Foundation for Statistical Computing. Vienna, Austria, 2016. URL: <https://cran.r-project.org/>.
- [Csa16] Gabor Csardi. *cranlogs: Download Logs from the 'RStudio' 'CRAN' Mirror*. R package version 2.1.1. 2016. URL: <https://github.com/metacran/cranlogs>.

- [16] *Cuarto informe de gobierno, 2015-2016. Anexo estadístico.* Inf. téc. 4. Presidencia de la República, agosto de 2016. URL: https://framework-gb.cdn.gob.mx/cuartoinforme/4IG_Anexo_Estadistico_TGM_26_08_16_COMPLETO.pdf.
- [Dut16] Christophe Dutang. *CRAN Task View: Probability Distributions.* <https://cran.r-project.org/web/views/Distributions.html>. Manual. 2016.
- [Edd16] Dirk Eddelbuettel. *gcbd: 'GPU'/CPU Benchmarking in Debian-Based Systems.* R package version 0.2.6. 2016. URL: <https://CRAN.R-project.org/package=gcbd>.
- [GW16] G. Grolemund y H. Wickham. *R for Data Science.* O'Reilly Media, Incorporated, 2016. ISBN: 9781491910399. URL: <http://r4ds.had.co.nz/>.
- [Hor16] Kurt Hornik. *R FAQ. Frequently Asked Questions on R.* Versión 2016-06-06. R Foundation for Statistical Computing. Vienna, Austria, 2016. URL: <https://cran.r-project.org/doc/FAQ/R-FAQ.html>.
- [Inc16a] RStudio Inc. *HTML Documents.* http://rmarkdown.rstudio.com/html_document_format.html. Blog. 2016.
- [Inc16b] RStudio Inc. *Introduction to R markdown.* <http://rmarkdown.rstudio.com/lesson-1.html>. Blog. 2016.
- [Inc16c] RStudio Inc. *PDF Documents.* http://rmarkdown.rstudio.com/pdf_document_format.html. Blog. 2016.
- [Inc16d] RStudio Inc. *R Markdown Reference.* <https://www.rstudio.com/wp-content/uploads/2015/03/rmarkdown-reference.pdf>. Blog. 2016.
- [Inc16e] RStudio Inc. *Word Documents.* http://rmarkdown.rstudio.com/word_document_format.html. Blog. 2016.
- [Ing16] Ingenio. *R: the ultimate virus.* <http://www.ingenio-magazine.com/r-the-ultimate-virus/>. Blog. 2016.
- [Kro+16] Sean Kross y col. *swirl: Learn R, in R.* R package version 2.4.2. 2016. URL: <https://CRAN.R-project.org/package=swirl>.
- [Lab16] Sharp Sights Labs. *Why R is the best data science language to learn today.* <http://sharpsightlabs.com/blog/r-recommend-data-science/>. Blog. 2016.
- [Lee16] Jeff Leek. *Why I don't use ggplot2.* <https://simplystatistics.org/2016/02/11/why-i-dont-use-ggplot2/>. Blog. 2016.
- [PKA16] RD Peng, S Kross y B Anderson. *Mastering software development in R.* Libro en desarrollo, disponible en línea, 2016. URL: <http://rdpeng.github.io/RProgDA/>.
- [Pla16] Google Cloud Platform. *What is BigQuery.* <https://cloud.google.com/bigquery/what-is-bigquery?>. Documentation. 23 de ene. de 2016.
- [R C16a] R Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing. Vienna, Austria, 2016. URL: <https://www.R-project.org/>.
- [R C16b] R Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing. Vienna, Austria, 2016. URL: <https://www.R-project.org/>.
- [R C16c] R Core Team. *R Language Definition, manual for R version 3.3.2.* R Foundation for Statistical Computing. Vienna, Austria, 2016. URL: <https://cran.r-project.org/doc/manuals/r-release/R-lang.html>.
- [RSt16] RStudio Team. *RStudio: Integrated Development Environment for R.* RStudio, Inc. Boston, MA, 2016. URL: <http://www.rstudio.com/>.

- [SD16] A Srinivasan y M Dowle. *data.table tutorials*. <https://www.datacamp.com/courses/data-table-data-manipulation-r-tutorial>. Tutorial. 2016.
- [Sta16] Stackoverflow. *Developer Survey Results*. <https://insights.stackoverflow.com/survey/2016>. Survey. 2016.
- [Ush+16] Kevin Ushey y col. *packrat: A Dependency Management System for Projects and their R Package Dependencies*. R package version 0.4.8-1. 2016. URL: <https://CRAN.R-project.org/package=packrat>.
- [Wic16a] Hadley Wickham. «Data Science with the Tidyverse». 4 de nov. de 2016. URL: <http://eventos.itam.mx/es/data-science-tidyverse>.
- [Wic16b] Hadley Wickham. *nycflights13: Flights that Departed NYC in 2013*. R package version 0.2.1. 2016. URL: <https://CRAN.R-project.org/package=nycflights13>.
- [Wic16c] Hadley Wickham. *tidyverse: Easily Tidy Data with 'spread()' and 'gather()' Functions*. R package version 0.6.1. 2016. URL: <https://CRAN.R-project.org/package=tidyr>.
- [WFM16] Hadley Wickham, Romain Francois y Kirill Müller. *tibble: Simple Data Frames*. R package version 1.2. 2016. URL: <https://CRAN.R-project.org/package=tibble>.
- [WG16] Hadley Wickham y Garrett Grolemund. *R for Data Science*. 2016.
- [WHF16] Hadley Wickham, Jim Hester y Romain Francois. *readr: Read Tabular Data*. R package version 1.0.0. 2016. URL: <https://CRAN.R-project.org/package=readr>.
- [Yau16] Nathan Yau. *Comparing ggplot2 and R Base Graphics*. <https://flowingdata.com/2016/03/22/comparing-ggplot2-and-r-base-graphics/>. Blog. 2016.
- [Mah] Michael S Mahoney. *The UNIX oral history project*. <https://www.princeton.edu/~hos/Mahoney/expotape.htm>. Archive.
- [WF] Hadley Wickham y Romain Francois. *dplyr: A Grammar of Data Manipulation*. R package version 0.5.0.9000. URL: <https://github.com/hadley/dplyr>.