

# Introducción a ggplot2 y ggmap

Carlos J. Gil Bellotta

PID\_00235524



Universitat Oberta  
de Catalunya



## Índice

<b>1. Introducción a ggplot2 .....</b>	<b>5</b>
1.1. Instalación de ggplot2 .....	7
1.2. Elementos de un gráfico en ggplot2 .....	7
1.2.1. Datos .....	7
1.2.2. Estéticas .....	8
1.2.3. Capas .....	9
1.2.4. Facetas .....	11
1.2.5. Más sobre estéticas .....	11
1.2.6. Temas .....	12
1.3. Ejemplos .....	13
1.3.1. Diagramas de cajas (y de violín) .....	13
1.3.2. Comparación de dos densidades .....	14
1.3.3. Series temporales .....	16
1.4. Datos medianos grandes .....	17
1.5. Resumen .....	20
<b>2. Introducción a ggmap .....</b>	<b>21</b>
2.1. Funciones de ggmap .....	22
2.1.1. Funciones para obtener mapas .....	23
2.1.2. Funciones para representar mapas .....	25
2.2. Ejemplos .....	25
2.2.1. Puntos sobre mapas .....	25
2.2.2. Más allá de los puntos: densidades y retículas .....	27
2.3. Resumen .....	29



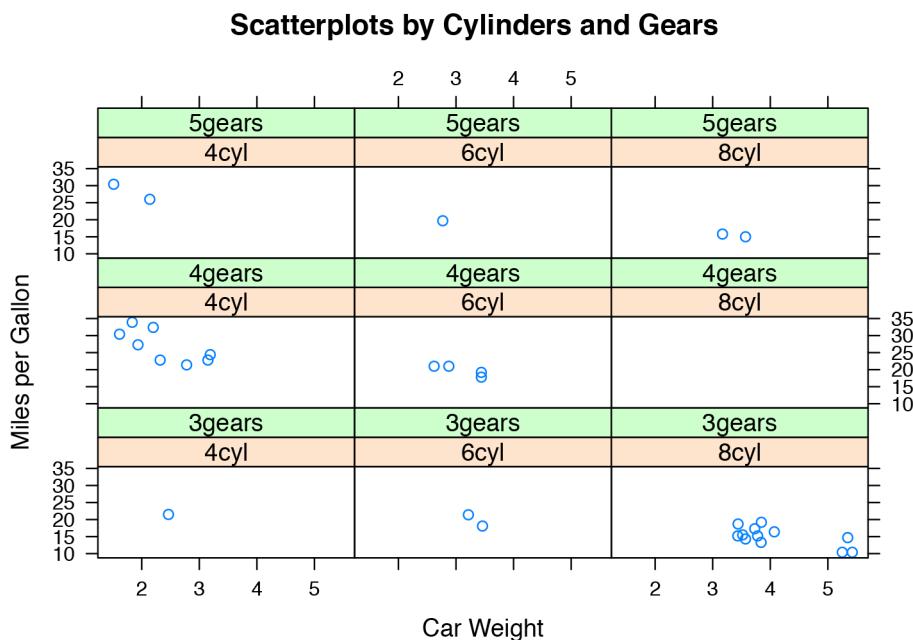
## 1. Introducción a ggplot2

R es un lenguaje para el análisis estadístico de datos. Como tal, una de sus características más destacadas es la de la generación de gráficos. En la mayor parte de los lenguajes de programación, la capacidad de crear gráficos la proporcionan librerías adicionales a su núcleo. Sin embargo, en R, los gráficos son nativos.

Existen dos *motores* gráficos en R. Un motor gráfico es un conjunto de funciones que permiten realizar manipulaciones gráficas básicas: generar lienzos (o *canvas*), trazar líneas, dibujar puntos, etc. Un usuario de R no manipula generalmente esas funciones directamente: utiliza funciones *de alto nivel*, como `plot`. La función `plot` es la encargada de invocar esas funciones de bajo nivel que pintan los segmentos, círculos, etc. que conforman un gráfico estadístico, con sus ejes, sus etiquetas, etc.

Funciones de R tales como `plot`, `hist`, `barplot`, `boxplot` y otras se apoyan en el motor tradicional de R. El motor tradicional de R es suficiente para esos fines. Sin embargo, se queda corto para construir otro tipo de gráficos más avanzados. Por eso, en 2001, Paul Murrell desarrolló un motor gráfico alternativo, `grid`. Uno de los objetivos de Paul Murrell era facilitar la generación en R de un tipo de gráficos conocidos como de Trellis, de celosía o de pequeños múltiples.

```
## The following object is masked from package:ggplot2:  
##  
##     mpg  
  
## The following objects are masked from mtcars (pos = 10):  
##  
##     am, carb, cyl, disp, drat, gear, hp, mpg, qsec, vs, wt
```



Los gráficos de Trellis permiten seguir el comportamiento de unas variables de interés a través de los distintos niveles de otras, disponiendo la información en una retícula que facilita el descubrimiento de patrones por inspección visual. El gráfico anterior muestra la relación entre el peso y el consumo de gasolina de una serie de vehículos en función de su número de cilindros y el número de marchas. La relación es más evidente usando un gráfico de Trellis que, por ejemplo, usando colores (o formas) para representar el número de cilindros o marchas en un único gráfico de dispersión.

Hay muchas funciones y paquetes que crean gráficos apoyándose en el motor gráfico tradicional. Otros, utilizan `grid`. Dos de los más conocidos son `lattice` (con el que está generado el gráfico anterior) y `ggplot2`. De hecho, `lattice` y `ggplot2` se solapan funcionalmente y la mayoría de los usuarios de R se decantan por uno u otro y lo usan de manera predominante.

`ggplot2` es unos años posterior a `lattice` pero, a pesar de ello, más popular. `ggplot2` es una implementación de las ideas recogidas en el artículo de "The Language of Graphics"\*, escrito por Leland Wilkinson y otros en 2000. Este artículo recogía una serie de ideas novedosas sobre qué es la representación gráfica de información estadística y cómo debería hacerse. Los gráficos tradicionales de R tienen un importante elemento de adhoquismo: las funciones para representar diagramas de dispersión, de cajas, de barras, etc. utilizan datos con distintos formatos, tienen parámetros no siempre coincidentes en nombre, etc. Lo revolucionario del planteamiento del artículo es poner de manifiesto que todos esos tipos de gráficos (y otros) pueden generarse mediante un *lenguaje* más o menos regular, con su sintaxis, su semiótica, etc. De la misma manera que el lenguaje natural organiza sonidos mediante ciertas reglas comunes, conocidas y regulares para generar mensajes con significado, es posible construir una serie de reglas comunes, conocidas y regulares para crear representaciones visuales de datos de interés estadístico.

\*<https://www.cs.uic.edu/~wilkinson/Publications/gpl.pdf>

De ese lenguaje, implementado en el paquete `ggplot2`, se ocupan los siguientes subapartados.

## 1.1. Instalación de ggplot2

El paquete `ggplot2` se instala como cualquier otro paquete de R: o bien desde línea de comandos con un `install.packages("ggplot2")` o bien utilizando los menús (Tools > Install Packages, etc.) de la interfaz de RStudio. `ggplot2` depende, como ha quedado claro más arriba, del paquete `grid`, pero este último viene instalado en R por defecto siempre.

El paquete tiene otras dependencias, que R sabe instalar por su cuenta si no se dispone de ellas. Dos de estas, los paquetes `plyr` y `reshape2`, tienen cierta relevancia: son del mismo autor que `ggplot2`, Hadley Wickham, y su uso para organizar convenientemente los datos para su representación gráfica encaja en la filosofía de `ggplot2`. Es recomendable familiarizarse con ellos.

Obviamente, para utilizar `ggplot2` una vez instalado es necesario importarlo así:

```
library(ggplot2)
```

## 1.2. Elementos de un gráfico en ggplot2

Un gráfico en `ggplot2` se construye combinando una serie de elementos básicos y comunes a muchos tipos de gráficos distintos mediante una sintaxis sencilla. Este subapartado describe esa sintaxis y los elementos que articula.

### 1.2.1. Datos

Uno de los elementos más importantes de un gráfico son los datos que se quieren representar. Una particularidad de `ggplot2` es que solo acepta un tipo de datos: `data.frames`. Otras funciones gráficas (por ejemplo, `hist`) admiten vectores, listas u otro tipo de estructuras. `ggplot2` no.

```
p <- ggplot(iris)
```

El código anterior crea un objeto, `p`, que viene a ser un protográfico: contiene los datos que vamos a utilizar, los del conjunto de datos `iris` (en `?iris` hay una descripción de este). Obviamente, el código anterior es insuficiente para crear un gráfico: aún no hemos indicado qué queremos hacer con `iris`.

### 1.2.2. Estéticas

En un conjunto de datos hay columnas: edad, altura, ingresos, temperatura, etc. En un gráfico hay, en la terminología de `ggplot2`, *estéticas*. Estéticas son, por ejemplo, la distancia horizontal o vertical, el color, la forma (de un punto), el tamaño (de un punto o el grosor de una línea), etc. Igual que al hablar asociamos a un conjunto de sonidos (por ejemplo, m-e-s-a) un significado (el objeto que conocemos como mesa), al realizar un gráfico asociamos a elementos sin significado propio (por ejemplo, los colores) uno: el que corresponde a una columna determinada de los datos.

En `ggplot2`, dentro del lenguaje de los gráficos que implementa, es muy importante esa asociación explícita de significados a significantes, es decir, de columnas de datos a *estéticas*.

En el código

```
p <- p + aes(x = Petal.Length, y = Petal.Width, colour = Species)
```

se están añadiendo a `p` información sobre las estéticas que tiene que utilizar y qué variables de `iris` debe utilizar:

- La distancia horizontal, `x`, vendrá dada por la longitud del pétalo.
- La distancia vertical, `y`, por su anchura.
- El color, por la especie.

Hay que hacer notar la sintaxis del código anterior, bastante particular y propia del paquete `ggplot2`. Al *protográfico* se le han sumado las estéticas. En los subapartados siguientes se le *sumarán* otros elementos adicionales. Lo importante es recordar cómo la suma es el signo que combina los elementos que componen el lenguaje de los gráficos.

De todos modos, es habitual combinar ambos pasos en una única expresión:

```
p <- ggplot(iris, aes(x = Petal.Length, y = Petal.Width, colour = Species))
```

El objeto `p` resultante aún no es un gráfico ni se puede representar. Le faltan capas, que es el objeto del siguiente subapartado. No obstante, se puede inspeccionar así:

```
summary(p)
```

```
## data: Sepal.Length, Sepal.Width, Petal.Length, Petal.Width,
##   Species [150x5]
## mapping: x = Petal.Length, y = Petal.Width, colour = Species
## facetting: facet_null()
```

Ahí están indicados los datos que va a utilizar y la relación (o *mapeo*) entre estéticas y columnas de los datos.

¿Cuántas estéticas existen? Alrededor de una docena, aunque se utilizan, generalmente, menos:

- `x` e `y`, distancias horizontal y vertical.
- `colour`, para el color.
- `size`, para el tamaño.
- `shape`, que indica la forma de los puntos (cuadrados, triángulos, etc.) de los puntos o del trazo (continuo, punteado) de las líneas.
- `alpha` para la transparencia: los valores más altos tendrían formas opacas y los más bajos, casi transparentes. De ahí la utilidad del canal alfa: da peso e importancia a las observaciones que la merecen.
- `fill`, para el color de relleno de las formas sólidas (barras, etc.).

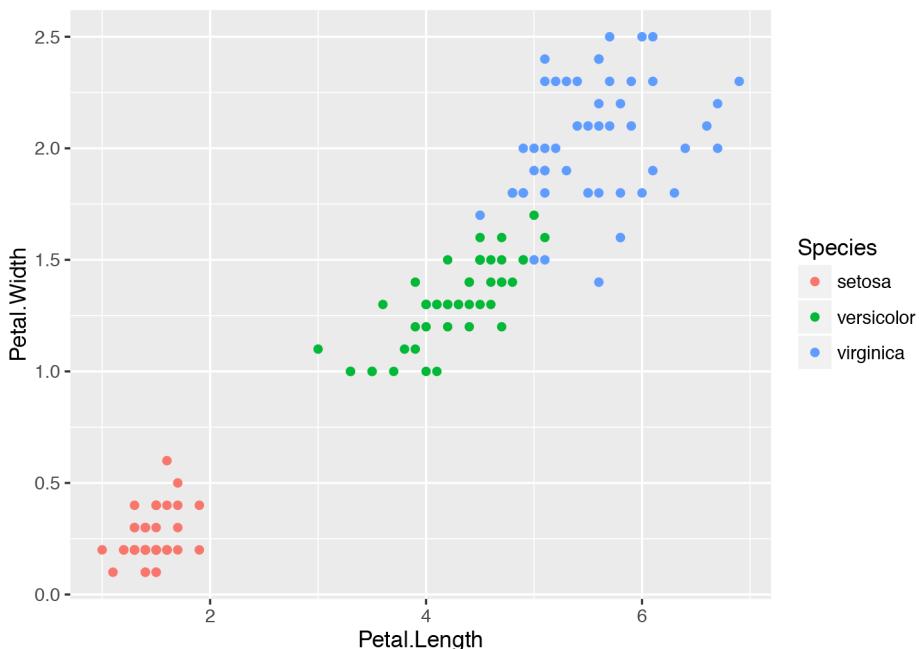
### Estéticas potentes

Hay que advertir que no todas las *estéticas* tienen la misma potencia en un gráfico. El ojo humano percibe fácilmente longitudes distintas, pero tiene problemas para comparar áreas (que es lo que regula la estética `size`) o intensidades de color. Se recomienda usar las estéticas más potentes para representar las variables más importantes.

### 1.2.3. Capas

Las capas (o `geoms` para `ggplot2`) son los verbos del lenguaje de los gráficos. Indican qué hacer con los datos y las estéticas elegidas, cómo representarlos en un lienzo. Y, en efecto, el siguiente código crea el correspondiente gráfico.

```
p <- p + geom_point()
p
```

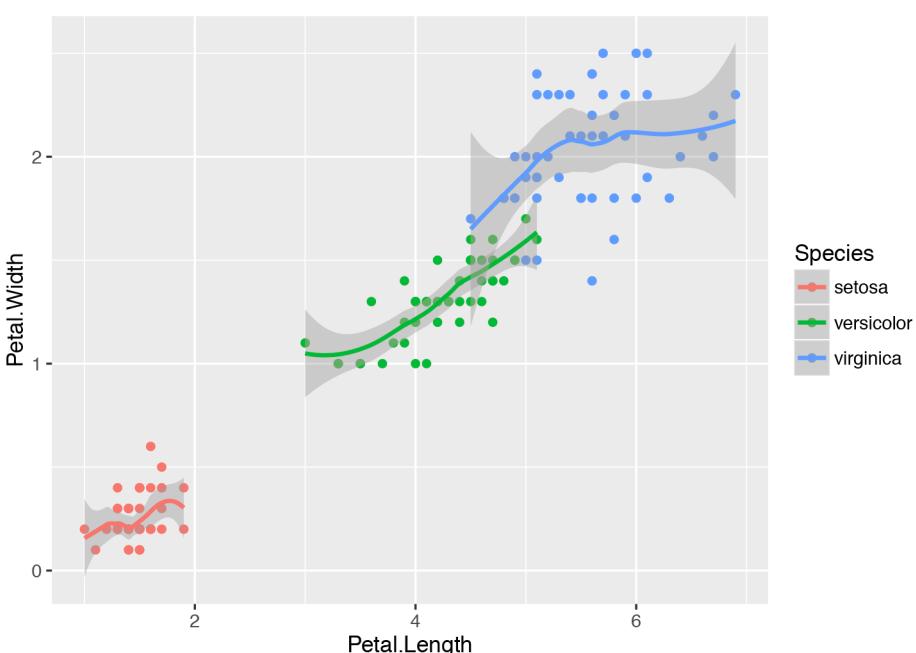


Una vez añadida una capa al gráfico, este puede pintarse (que es lo que ocurre al llamar a `p`). Se obtiene el mismo resultado haciendo, en una única línea,

```
ggplot(iris, aes(x = Petal.Length, y = Petal.Width, colour = Species)) + geom_point()
```

Una característica de las capas, y de ahí su nombre, es que pueden superponerse. Por ejemplo, el siguiente código añade al gráfico una curva suavizada (con sus intervalos de confianza en gris).

```
ggplot(iris, aes(x = Petal.Length, y = Petal.Width, colour = Species)) +
  geom_point() + geom_smooth()
```



Existen muchos tipos de capas. Los más usuales son `geom_point`, `geom_line`, `geom_histogram`, `geom_bar` y `geom_boxplot`, pero hay más.

En la página <http://docs.ggplot2.org/current/> se muestra una lista de los disponibles (en la versión más actualizada de `ggplot2`). En esa página se indica qué `geom` hay que utilizar en función de una representación esquemática del tipo de gráfico que se quiere construir. Además, hay capas específicas que exigen estéticas especiales. Para algunas tiene sentido, por ejemplo, `shape`. Para otras, no. Esas especificidades están indicadas en dicha página, que es más útil que la ayuda general de R.

Una vez creado un gráfico, es posible exportarlo a PNG, JPG, etc. La función `ggsave` guarda en un fichero el último gráfico generado con `ggplot2`. Lo hace, además, en el formato indicado en el nombre del fichero que se quiere generar. Así,

```
ggplot(iris, aes(x = Petal.Length, y = Petal.Width, colour = Species)) + geom_point()
ggsave("mi_grafico.png")
```

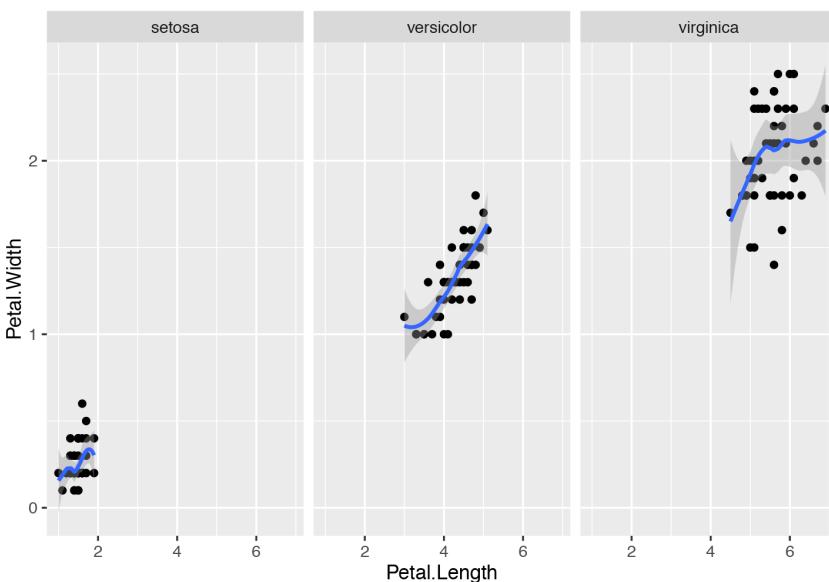
guarda la figura creada en la primera línea en formato PNG en el fichero mi\_grafico.png del directorio de trabajo.

### 1.2.4. Facetas

Muchos de los gráficos que pueden generarse con los elementos anteriores pueden reproducirse sin mucho esfuerzo (exceptuando, tal vez, cuestiones de aspecto) usando los gráficos tradicionales de R. Los que permiten el uso de facetas, no.

Las facetas implementan los gráficos de Trellis mencionados antes. Por ejemplo, el siguiente código crea tres gráficos dispuestos horizontalmente que comparan la relación entre la anchura y la longitud del pétalo de las tres especies de iris. Una característica de estos gráficos, que es crítica para poder hacer comparaciones adecuadas, es que comparten ejes.

```
ggplot(iris, aes(x = Petal.Length, y = Petal.Width)) +
  geom_point() + geom_smooth() +
  facet_grid(~ Species)
```



Los gráficos podrían disponerse verticalmente reemplazando `facet_grid(~ Species)` por `facet_grid(Species ~ .)` en el código anterior. En caso de haber muchas categorías (por ejemplo, provincia), puede usarse la función `facet_wrap` para distribuir las subgráficas en una cuadrícula.

### 1.2.5. Más sobre estéticas

Las estéticas se pueden etiquetar con la función `labs`. Además, se le puede añadir un título al gráfico usando la función `ggtitle`. Por ejemplo, en el gráfico anterior (subapartado 1.2.3) se pueden reetiquetar los ejes y la leyenda haciendo

```
p <- p + ggtitle("Petal length and width") +
  labs(x = "Petal length",
       y = "Petal width",
       colour = "Species")
```

### 1.2.6. Temas

Los *temas* de ggplot2 permiten modificar aspectos estéticos del gráfico que no tienen que ver con los datos en sí. Eso incluye los ejes, las etiquetas, los colores de fondo, el tamaño de los márgenes, etc. No es habitual (y se desaconseja a los usuarios menos expertos) tener que alterar los temas que ggplot2 usa por defecto. Solo se hace necesario cuando los gráficos tienen que adecuarse a una imagen corporativa o hay que atenerse a algún criterio de publicación exigente.

Un tema es una colección de elementos (por ejemplo, `panel.background`, que indica el color, transparencia, etc., del lienzo sobre el que se representa el gráfico) modificables. El tema que usa ggplot2 por defecto es `theme_grey`. Al escribir `theme_grey()` en la consola de R, se muestran alrededor de cuarenta elementos modificables y sus atributos tal y como los define dicho tema.

¿Qué se puede hacer con los temas? Una primera opción es elegir otro. Por ejemplo, se puede reemplazar el habitual por otros disponibles en el paquete como `theme_bw` (o `theme_classic`) haciendo

```
p <- p + theme_bw()
```

Es posible usar tanto los temas que incluye ggplot2 por defecto como otros creados por la comunidad. Algunos, por ejemplo, tratan de imitar el estilo de publicaciones reconocidas, como *The Economist* o similares. Algunos están recogidos en paquetes como, por ejemplo, `ggthemes`.

De manera alternativa (o adicional), es posible modificar un tema dado en un gráfico. Por ejemplo, haciendo

```
p <- p +
  theme_bw() +
  theme(
    panel.background = element_rect(fill = "lightblue"),
    panel.grid.minor = element_line(linetype = "dotted")
  )
```

se está modificando el atributo de color del lienzo de un gráfico y el tipo de la línea con que se dibuja la malla.

#### Personalización de temas

Es posible construir temas propios y personalizados. Aunque no es un proceso complicado, los detalles quedan fuera del alcance de este material.

### 1.3. Ejemplos

En este subapartado se van a explorar algunos de los gráficos estadísticos más básicos.

#### 1.3.1. Diagramas de cajas (y de violín)

Los diagramas de caja (*boxplots*) describen de manera cruda la distribución de una variable continua en función de una discreta. En el ejemplo que aparece a continuación, se explora el conjunto de datos `iris`, que contiene 50 observaciones de características métricas de cada una de las tres subespecies de iris, una flor. Es un conjunto de datos de larga tradición en estadística y se recopiló para ilustrar algoritmos de clasificación, es decir, cómo crear criterios para distinguir los tres tipos de iris.

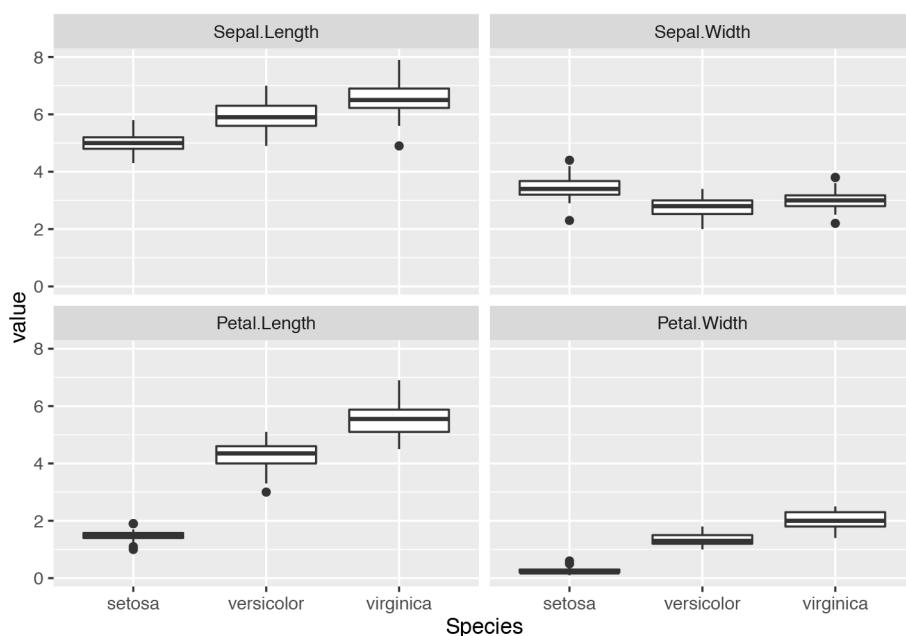
##### Iris dataset

<https://archive.ics.uci.edu/ml/datasets/Iris>

La función `melt()` crea una nueva tabla a partir de unos datos (que están en lo que se conoce como formato ancho, con un campo para cada valor), de forma que existe un registro para cada uno de los campos que se desea recodificar (normalmente todos aquellos que no son factores), usando dos nuevas variables, una llamada `variable`, que contiene el nombre del campo original, y otra llamada `value`, que contiene el valor del campo original.

```
library(reshape2)
tmp <- melt(iris)

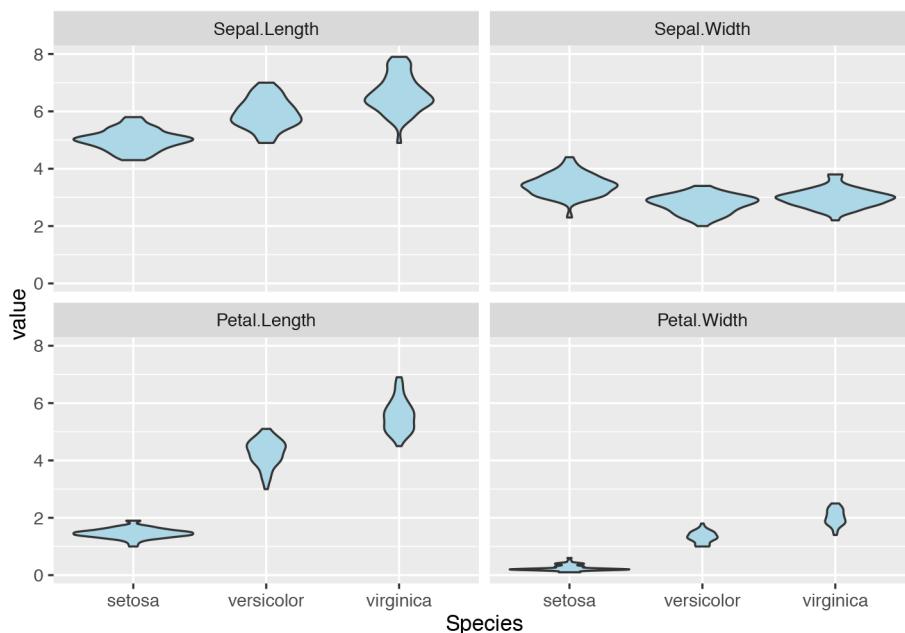
ggplot(tmp, aes(x = Species, y = value)) + geom_boxplot() + facet_wrap(~ variable)
```



El gráfico utiliza las facetas para crear cuatro paneles, uno por variable. Así resume rápidamente la información contenida en el conjunto de datos y revela eficazmente los patrones que encierra: por ejemplo, cómo la especie setosa tiene el pétalo sensiblemente más estrecho y corto que las otras dos. Este tipo de gráficos es fundamental como herramienta exploratoria previa a la aplicación de técnicas de análisis estadístico (discriminante, en este caso).

Los gráficos de cajas son muy básicos: apenas muestran cinco puntos característicos de una distribución: la mediana, los extremos y los cuartiles. Son herencia de una época en la que apenas había recursos, principalmente informáticos, para realizar representaciones más sofisticadas. Una versión moderna de los gráficos de cajas es la de gráficos de violín. Como los de cajas, resumen la distribución de las variables, pero en lugar de una representación sucinta, tratan de dibujar la distribución real de los datos: son verdaderos gráficos de densidad, solo que dispuestos de otra manera para facilitar la comparación.

```
ggplot(tmp, aes(x = Species, y = value)) +
  geom_violin(fill = "lightblue") +
  facet_wrap(~ variable)
```



### 1.3.2. Comparación de dos densidades

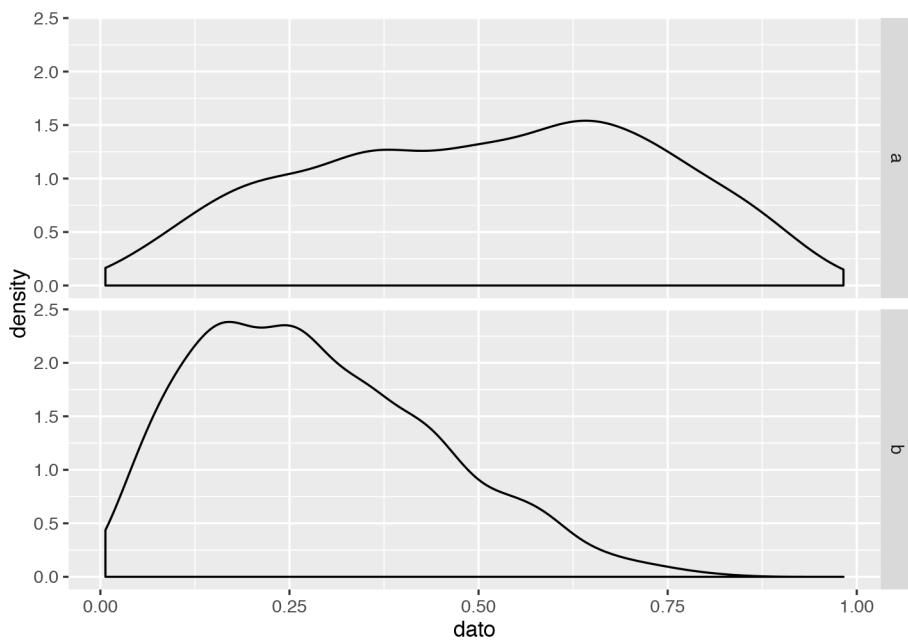
El análisis de datos exige en ocasiones comparar dos distribuciones continuas. Se pueden usar gráficos de cajas o de violín, como arriba, pero también se puede dibujar la distribución completa, como en el siguiente gráfico:

```
# datos (simulados)
a <- rbeta(1000, 2, 2)
```

```
b <- rbeta(2000, 2, 5)

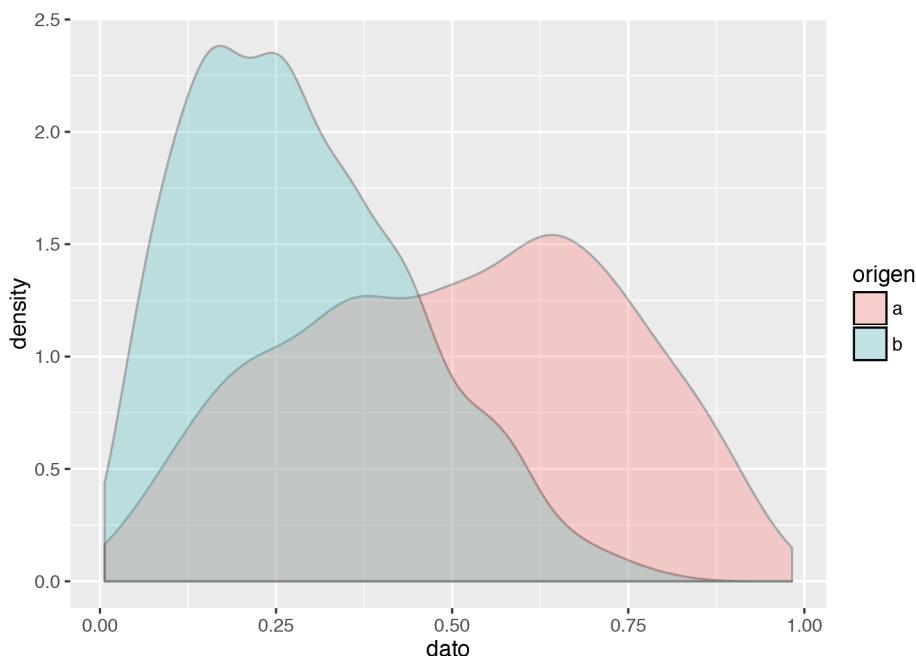
# construcción de un dataframe a partir de ellos
tmp <- rbind(data.frame(origen = "a", dato = a),
              data.frame(origen = "b", dato = b))

ggplot(tmp, aes(x = dato)) + geom_density() + facet_grid(origen ~ .)
```



Alternativamente, se pueden solapar ambas distribuciones. El uso del parámetro alpha, que controla la transparencia, es fundamental en este caso:

```
ggplot(tmp, aes(x = dato, fill = origen)) + geom_density(alpha = 0.3)
```



### 1.3.3. Series temporales

ggplot2 entiende ciertos tipos de datos particulares, como por ejemplo, series temporales. En el siguiente ejemplo se descargan las cotizaciones bursátiles de dos banco españoles directamente de Yahoo! Finance y se representan gráficamente:

```
# instalarlos previamente si es necesario
library(tseries)
library(zoo)
library(reshape2)

# función para descargar las cotizaciones
cotizaciones <- function(valor) {
  res <- get.hist.quote(valor, provider = "yahoo", quote = "AdjClose")
  colnames(res) <- valor
  res
}

# combinación de ambas series temporales
res <- merge(cotizaciones("SAN.MC"),
             cotizaciones("BBVA.MC"))

## time series starts 2000-01-03
## time series starts 2000-01-03

# construcción de un dataframe con un índice de tipo fecha
res <- data.frame(fecha = index(res), as.data.frame(res))
res <- melt(res, id.var = "fecha")

ggplot(res, aes(x = fecha, y = value)) + geom_line() + facet_grid(variable ~ .)
```



Una versión alternativa del gráfico es la que superpone las series usando colores para distinguirlas:

```
ggplot(res, aes(x = fecha, y = value, colour = variable)) + geom_line() +
  labs(colour = "valor", y = "cotización")
```



## 1.4. Datos medianos grandes

Cada vez es más frecuente enfrentarse al problema de la representación gráfica de datos grandes (*big data*). Muchas de las representaciones gráficas tradicionales son imposibles por diversos motivos:

- Exigen un preproceso de datos (por ejemplo, si se quiere usar `geom_smooth`) muy pesado.
- Esos algoritmos no están pensados para datos grandes; es decir, podría haber implementaciones de esos algoritmos que podrían valer para datos grandes, pero las existentes no lo son.
- Pero, sobre todo, porque las funciones gráficas son muy lentas: representar millones de círculos sobre el lienzo (o *canvas*) de un fichero .png es muy costoso computacionalmente.

No obstante, siempre se puede recurrir a representaciones gráficas que agreguen datos de alguna manera determinada. O, alternativamente, realizar la preagregación antes de aplicar las funciones gráficas propiamente dichas.

En lo que sigue, se van a realizar algunas exploraciones gráficas sobre parte de un conjunto de datos muy popular para realizar pruebas en entornos de datos grandes: el de los retrasos de las compañías aéreas\*, que contiene información sobre todos los vuelos comerciales en Estados Unidos entre 1998 y 2008, indicando origen, destino, tiempo de vuelo, distancia de vuelo, tiempo de retraso, incidencias, etc. Se utilizará el correspondiente a 2008, que contiene información de algo más de siete millones de vuelos, ocupa 650 MB y puede descargarse del enlace anterior.

\*<http://stat-computing.org/dataexpo/>

En primer lugar, se va a proceder a cargarlo (para lo que se recomienda la función `fread` del paquete `data.table`, mucho más eficiente, aunque también menos flexible, que la habitual `read.table` para datos grandes). También se va a extraer de él un subconjunto *mediano* (100.000 filas) al azar.

```
library(data.table)

raw <- fread("2008.csv")
small <- raw[sample(1:nrow(raw), 100000), ]
```

Es tentador tratar de ejecutar

```
ggplot(small, aes(x=Distance, y = AirTime)) + geom_point()
```

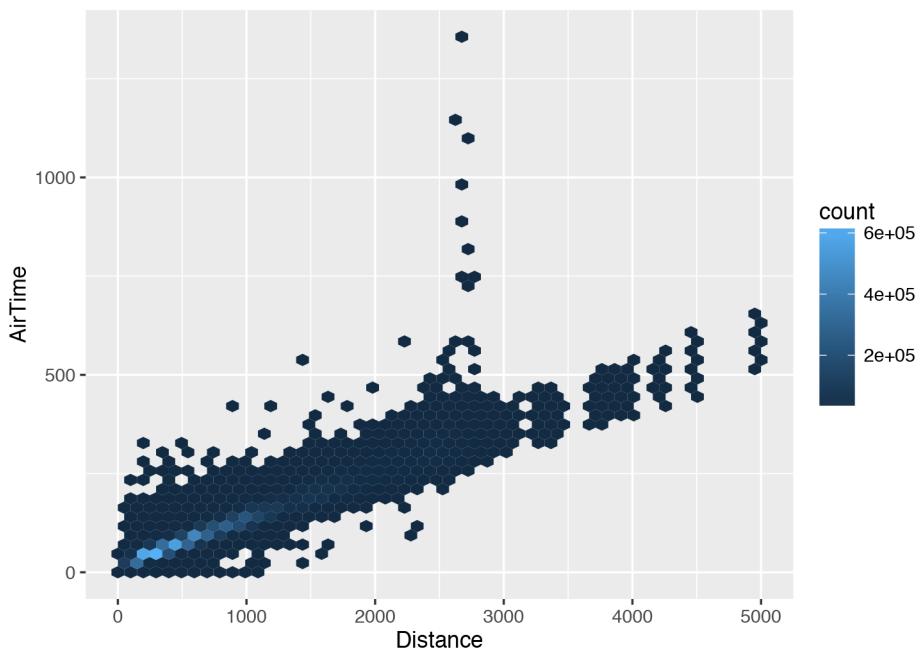
con el conjunto mediano de datos. Sin embargo, a diferencia de lo que ocurre con conjuntos más pequeños, la opacidad de los puntos, junto con el gran número de ellos, producen un efecto de mancha negra que esconde los detalles. En tales casos es recomendable modular la transparencia así

```
ggplot(small, aes(x=Distance, y = AirTime)) +
  geom_point(alpha = 0.01) + geom_smooth()
```

Gracias a la transparencia, se detectan fácilmente las zonas de mayor densidad de datos. Sin embargo, este tipo de subterfugios fallan cuando se quiere representar el conjunto de datos completo con sus siete millones de filas. De todos los *geoms* que ofrece `ggplot2` solo unos pocos, los que realizan preagregaciones de datos, pueden aplicarse efectivamente: histogramas, teselaciones hexagonales (`geom_hex`), diagramas de cajas,...

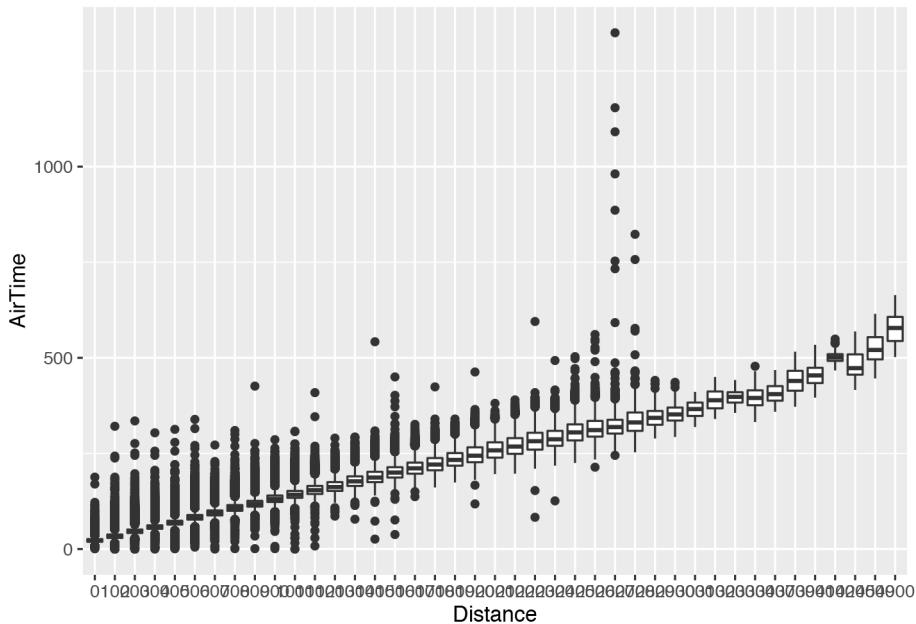
Por ejemplo,

```
ggplot(raw, aes(x=Distance, y = AirTime)) + geom_hex(bins = 50)
```



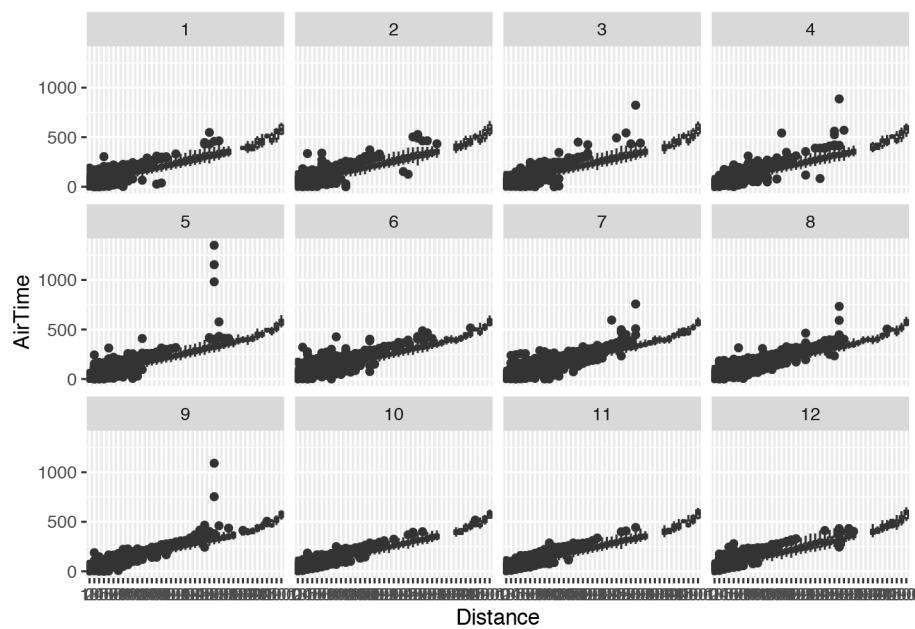
La binarización de una variable continua permite también utilizar diagramas de caja (nótese que cada caja *resume* información relativa a un conjunto potencialmente grande de datos) para realizar visualizaciones efectivas:

```
raw$DistanceBreaks <- factor(100 * floor(raw$Distance / 100))
ggplot(raw, aes(x = DistanceBreaks, y = AirTime)) + geom_boxplot() +
  xlab("Distance")
```



Y, por supuesto, el uso de facetas no supone un incremento inasumible de la complejidad computacional:

```
ggplot(raw, aes(x = DistanceBreaks, y = AirTime)) + geom_boxplot() +
  facet_wrap(~ Month) + xlab("Distance")
```



Nótese que el eje de las X en ambos gráficos resulta ilegible, dada la gran cantidad de datos a mostrar. Con `ggplot2` también es posible determinar el etiquetado de cada uno de los ejes, resolviendo dicho problema.

Existen consideraciones adicionales a la hora de representar conjuntos de datos muy grandes. El caso anterior, por ejemplo, pone de manifiesto cómo unos pocos retrasos aislados (dentro de un conjunto de millones de vuelos) cobran un protagonismo excesivo: ¿merecería, por tanto, la pena filtrarlos previamente para hacer hincapié en la estructura general de los datos? ¿O son precisamente esos valores atípicos los que exigen nuestra atención? Dependiendo de las respuestas a estas preguntas podría plantearse la posibilidad de realizar algún tipo de filtro previo.

## 1.5. Resumen

`ggplot2` es un paquete moderno para crear gráficos estadísticos avanzados. Utiliza una sintaxis particular que trata de unificar la manera de generar gráficos. Exige un cierto esfuerzo por parte de quienes se inician en su uso, pero permite llegar mucho más lejos que los gráficos tradicionales de R.

## 2. Introducción a ggmap

Con `ggplot2` pueden construirse muchos tipos de gráficos de interés estadístico, pero sus autores quisieron trasladar la arquitectura del paquete a otro ámbito: el de la representación de información georreferenciada. `ggplot2` permite representar información geográfica (puntos, segmentos, etc.): basta con que las estéticas `x` e `y` se correspondan con la longitud y la latitud de los datos. Lo que permite hacer `ggmap` es, en esencia, añadir a los gráficos ya conocidos una capa cartográfica adicional. Para eso, emplea recursos disponibles en la *web* a través de API (de Google y otros).

Un ejemplo sencillo ilustra los usos de `ggmap`. En primer lugar, se carga (si se ha instalado previamente) el paquete:

```
library(ggmap)
```

Existen varios proveedores que proporcionan API de geolocalización. Uno de ellos es Google: dado el nombre más o menos normalizado de un lugar, la API de Google devuelve sus coordenadas. Este servicio tiene una versión gratuita, que permite realizar un determinado número de consultas diarias (2.500 actualmente); para usos más intensivos, es necesario adquirir una licencia. La función `geocode` encapsula la consulta a dicha API y devuelve un objeto (un `data.frame`) que contiene las coordenadas del lugar de interés:

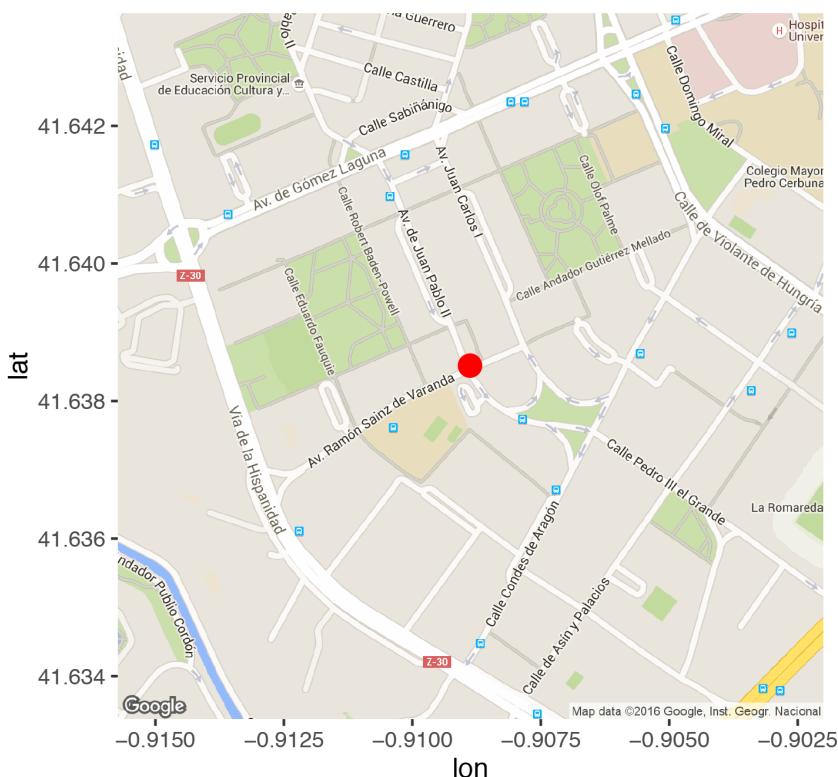
```
unizar <- geocode('Universidad de Zaragoza, Zaragoza, España',
                  source = "google")
```

La función `get_map` consulta otro servicio de información cartográfica (GoogleMaps en el ejemplo siguiente) y descarga un mapa (que es, esencialmente, una imagen *raster*). La función exige una serie de argumentos: el nivel de `zoom`, si se quiere un mapa de carreteras o del terreno, etc. Son, de hecho, los parámetros que uno puede manipular con los controles de la interfaz habitual de GoogleMaps.

```
map.unizar <- get_map(location = as.numeric(unizar),
                      color = "color",
                      maptype = "roadmap",
                      scale = 2,
                      zoom = 16)
```

Es obvio que para poder invocar las dos funciones anteriores hace falta una conexión a internet. Sin embargo, el resto de las operaciones que se van a realizar se ejecutan localmente. Se puede, por ejemplo, representar el mapa directamente (haciendo `ggmap(map.unizar)`). O también se puede marcar sobre él el punto de interés:

```
ggmap(map.unizar) + geom_point(aes(x = lon, y = lat),
                                data = unizar, colour = 'red',
                                size = 4)
```



Como puede apreciarse, la sintaxis es similar a la de `ggplot2`. Una diferencia notable es que, ahora, los datos se pasan en la capa, es decir, en este caso, en la función `geom_point`.

## 2.1. Funciones de `ggmap`

`ggmap` incluye muchas funciones, que pueden clasificarse en tres categorías amplias:

- Funciones para obtener mapas (de diversos tipos y de distintos orígenes: Google, Stamen, OpenStreetMap).
- Funciones que utilizan API de Google y otros. Por ejemplo, `revgeocode`, `geocode` y `route` consultan la información que tienen distintos proveedores de servicios vía API sobre las coordenadas de un determinado lugar; indican el lugar al que se refieren unas coordenadas, y, finalmente, encuen-

tran rutas entre dos puntos. Es conveniente recordar que las consultas a los servicios de Google Maps exige la aceptación de las condiciones de uso y que existe un límite diario en el número de consultas gratuitas.

- Funciones que pintan mapas y que representan determinados elementos adicionales (puntos, segmentos, etc.) en mapas.

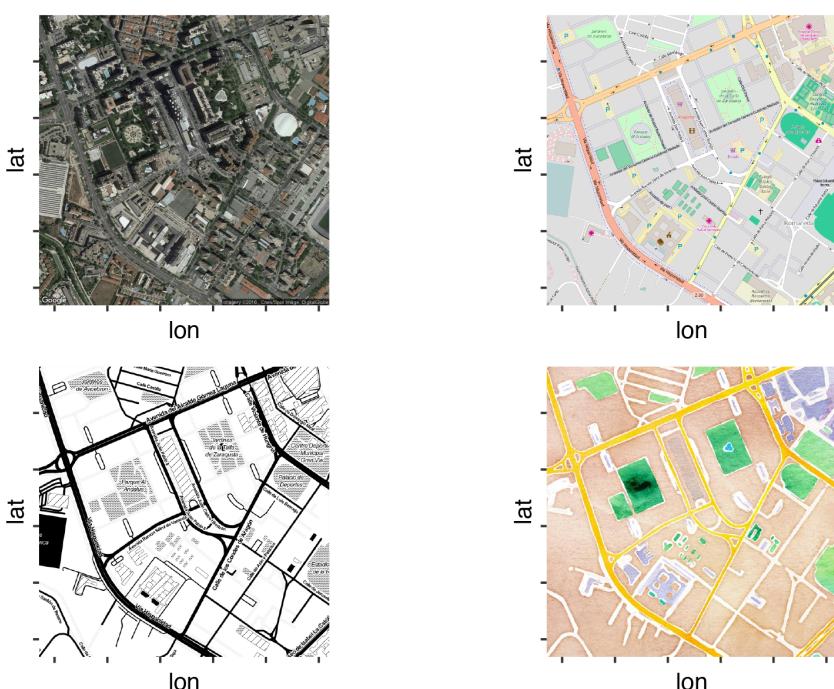
En este subapartado se van a presentar los tres tipos de funciones de `ggmap`.

### 2.1.1. Funciones para obtener mapas

`ggmap` obtiene sus mapas, por defecto, de GoogleMaps. Sin embargo, hay otros proveedores de mapas libres, como OpenStreetMap (OSM) o Stamen. Cada proveedor exige una serie de parámetros distintos y, por ejemplo, un *zoom* de 8 puede significar una escala distinta en GoogleMaps que en OSM. No obstante, los autores de `ggmap` se han tomado la molestia de homogeneizar los argumentos de llamada para que sean aproximadamente equivalentes en todos los proveedores.

`ggmap` incluye funciones específicas para cada proveedor, como son por ejemplo `get_googlemap` o `get_stamenmap`, pero salvo para usos avanzados, es recomendable usar la función `get_map`, que ofrece un punto de entrada único y homogéneo para el resto.

El gráfico siguiente muestra cuatro mapas obtenidos de distintos proveedores y con diversas opciones. En la fila superior, una capa de Google en modo imagen de satélite y otra de OSM. En la inferior, dos mapas de Stamen, uno en modo *toner* y otro en modo *watercolor* o acuarela. Son solo cuatro de los muchos a los que la función `get_map` puede acceder.



## Funciones para consultar API cartográficas

Muchos servicios de información cartográfica proporcionan API para realizar consultas. Las API se consultan, típicamente, con URL convenientemente construidas.

Por ejemplo, la siguiente URL consulta el servicio de geolocalización de GoogleMaps y devuelve las coordenadas de la Universidad de Zaragoza (así como otra información relevante en formato JSON):

`http://maps.googleapis.com/maps/api/geocode/json?address=Universidad+de+Zaragoza`  
 La función `geolocate` de `ggmap` facilita la consulta a dicho servicio: toma su argumento (el nombre de un lugar), construye internamente la URL, realiza la consulta (para lo que es necesario conexión a internet), lee la respuesta y le da un formato conveniente (en este caso, un `data.frame` de R).

La de geolocalización no es la única API que permite consultar `ggmap`. También permite invertir la geolocalización, es decir, dadas unas coordenadas, devolver el nombre del lugar al que se refieren:

```
revgeocode(as.numeric(unizar))
```

```
## [1] \comillas Av. Ramón Sainz de Varanda, 112, 50009 Zaragoza,  

Zaragoza, Spain\textquotedblright
```

Finalmente, `route` permite obtener la ruta entre dos puntos distintos:

```
mapa <- get_map("Madrid", source = "stamen", maptype = "toner", zoom = 12)  

ruta <- route(from = "Puerta del Sol, Madrid", to = "Plaza de Castilla, Madrid")  

ggmap(mapa) +  

  geom_path(aes(x = startLon, y = startLat, xend = endLon, yend = endLat),  

    colour = "red", size = 2, data = ruta)
```



En el mapa anterior la ruta elegida por GoogleMaps para ir de la Puerta del Sol hasta la plaza de Castilla (dos plazas de Madrid) está marcada en rojo sobre un mapa de Stamen de tipo *toner*.

### 2.1.2. Funciones para representar mapas

Como se ha visto en los subapartados anteriores, la función `ggmap` permite representar un mapa descargado previamente. Además, a esa capa subyacente se le pueden añadir elementos (puntos, segmentos, densidades etc.) usando las funciones ya conocidas de `ggplot2`: `geom_point`, etc.

En `ggplot2` existe una función, `geom_path` que dibuja caminos (secuencias de segmentos). Se puede utilizar en `ggmap` para dibujar rutas, aunque este paquete proporciona una función especial, `geom_leg`, que tiene la misma finalidad aunque con algunas diferencias menores: por ejemplo, los segmentos tienen las puntas redondeadas para mejorar el resultado gráfico.

## 2.2. Ejemplos

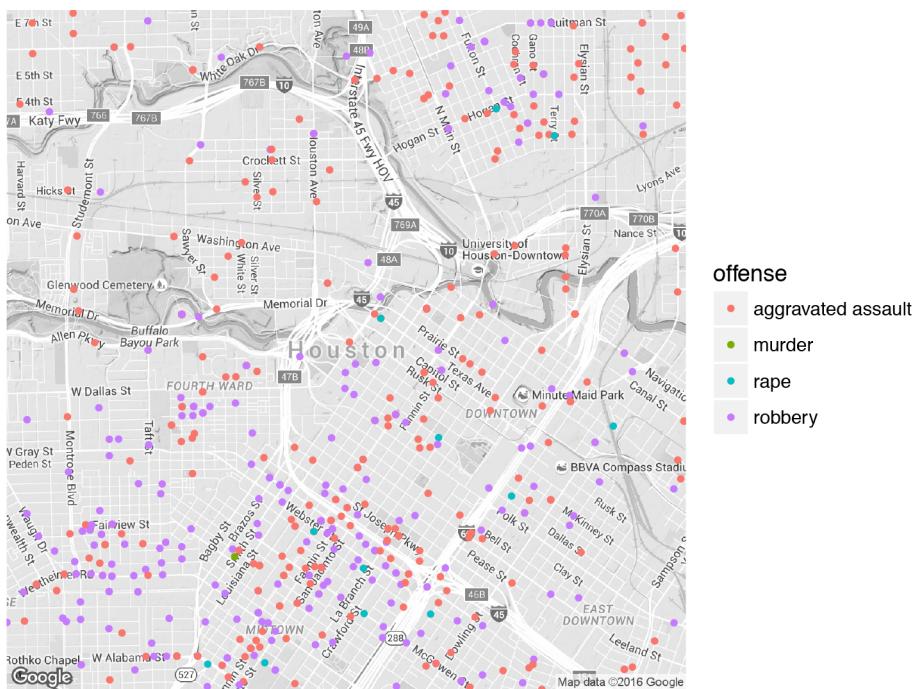
En los ejemplos que siguen se va a utilizar el conjunto de datos `crimes`, que forma parte del paquete `ggmap` y que incluye información geolocalizada de crímenes cometidos en la ciudad de Houston. En realidad, solo consideraremos los crímenes *serios*, es decir, descontaremos los robos:

```
crimes.houston <- subset(crime, ! crime$offense %in% c("auto theft", "theft", "burglary"))
```

### 2.2.1. Puntos sobre mapas

El tipo de mapas más simple es el que se limita a representar puntos sobre una capa cartográfica.

```
HoustonMap <- qmap("houston", zoom = 14, color = "bw")
HoustonMap +
  geom_point(aes(x = lon, y = lat, colour = offense), data = crimes.houston, size = 1)
```



Los mecanismos conocidos de `ggplot2`, como las facetas, están disponibles en `ggmap`: es posible crear una retícula de mapas usando `facet_wrap`. En este primer caso, descomponiendo el gráfico anterior por tipo de crimen.

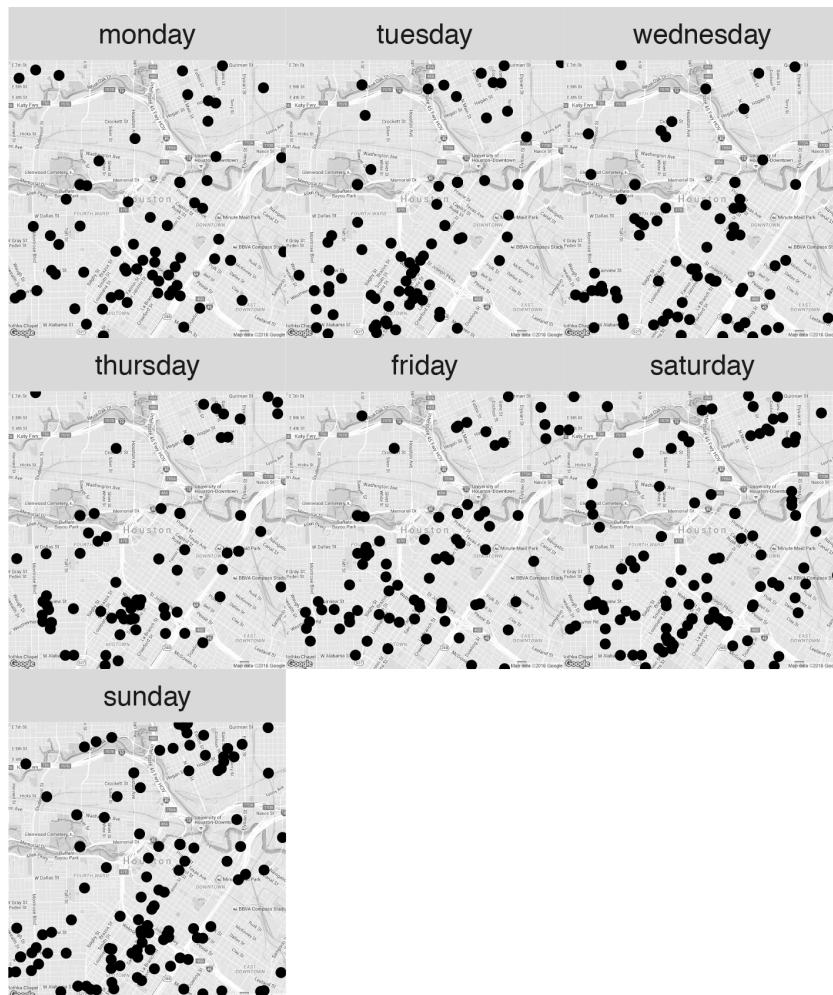
```
HoustonMap +
  geom_point(aes(x = lon, y = lat), data = crimes.houston, size = 1) +
  facet_wrap(~ offense)
```



O, alternativamente, por día de la semana.

HoustonMap +

```
geom_point(aes(x = lon, y = lat), data = crimes.houston, size = 1) +
facet_wrap(~ day)
```

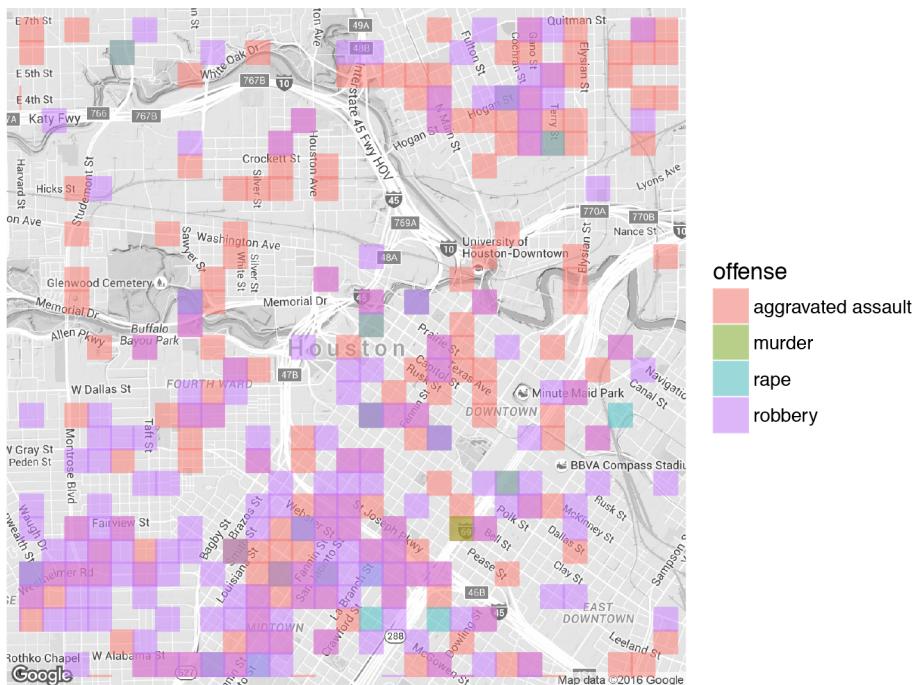


### 2.2.2. Más allá de los puntos: densidades y retículas

Además de `geom_point`, también están disponibles otros tipos de capas de `ggplot2`, como `stat_bin2d`, que cuenta el número de eventos que suceden en regiones cuadradas de un tamaño predefinido.

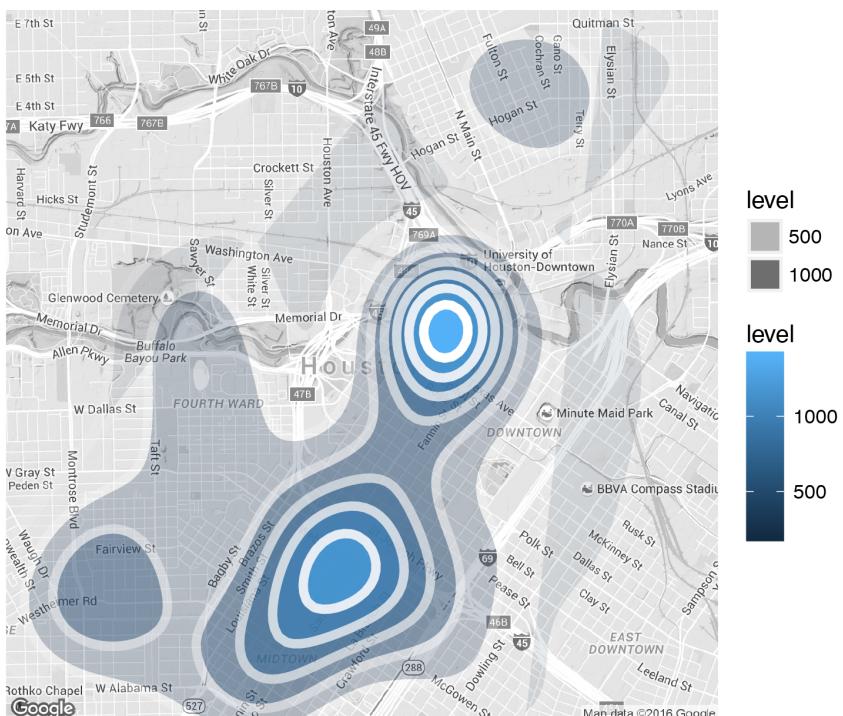
HoustonMap +

```
stat_bin2d(
  aes(x = lon, y = lat, colour = offense, fill = offense),
  size = .5, bins = 30, alpha = 1/2,
  data = crimes.houston
)
```



O se puede usar `stat_density2d`, que representa intensidades, para identificar las zonas de mayor criminalidad.

```
HoustonMap +
  stat_density2d(aes(x = lon, y = lat, fill = ..level.., alpha = ..level..),
                 size = 2, data = crimes.houston,
                 geom = "polygon"
  )
```



### 2.3. Resumen

`ggmap` es una extensión del paquete `ggplot2` para representar información cartográfica. Además de funciones meramente gráficas, dispone de otras que permiten realizar operaciones de interés geográfico a través de consultas a API de proveedores externos, como Google Maps.

