

Gráficos con R

Olivier Nuñez

2019-09-23

Índice

Preámbulo	1
1. Gráficos básicos en el análisis de datos	2
1.1. Distribución de una variable	2
1.2. Relación entre dos variables	8
2. Gráficos avanzados	15
2.1. Elementos de un gráfico	16
2.2. Mapas	31
2.3. Árboles filogenéticos	40
3. Rmarkdown y shiny	49
3.1. Rmarkdown	49
3.2. Shiny	54
4. Referencias	56

```
knitr::opts_chunk$set(warning = FALSE,message = FALSE, cache.lazy = FALSE)
```

Preámbulo

“Una imagen vale más que mil palabras” (“Un bon croquis vaut mieux qu’un long discours”)

— Napoleon

- La representación gráfica de los datos es una potente herramienta de síntesis de la información estadística.
- Cualquiera que sea la etapa de su trabajo (exploración, modelización, confirmación o comunicación de resultados), el usuario de métodos estadísticos encuentra en los gráficos un aliado para ahorrar tiempo.
- El objetivo de esta sesión es manejar el paquete `ggplot2` de R y sus extensiones que permiten un diseño eficiente de gráficos para explorar la distribución de un conjunto de datos. Se abordarán los distintos pasos, que van desde la preparación de los datos hasta su exportación.
- La estructura de esta sesión tiene tres partes:
 - La primera parte está dedicada a la definición y elaboración de las representaciones gráficas más comunes (diagrama de barras, de caja, de dispersión, histograma) para extraer la información contenida en un conjunto de datos.
 - La segunda parte describe la “gramática” general de confección de gráficos utilizada en `ggplot2`. Esta gramática permite tratar de manera sistemática y flexible, aspectos básicos como la elección de los ejes, los colores y la leyenda. Pero también, temas avanzados como el problema de solapamiento de puntos, la superposición de varios elementos gráficos (capas) o el desglose de una representación en varios sub-gráficos (facetas).
 - En la ultima parte, se presentarán las herramientas para insertar esta información gráfica en informes estáticos, dinámicos o automatizados, mediante los paquetes `Rmarkdown` y `Shiny`.

1. Gráficos básicos en el análisis de datos

Esta sección es una breve introducción a los gráficos más comunes para analizar conjuntos de datos. Para ello, utilizaremos el paquete `ggplot2` (“gg” para “Grammar of Graphics”).

```
#install.packages("ggplot2")
library(ggplot2) #carga la librería ggplot2
```

Empezaremos con la función básica `qplot` (“quick plot”) de este paquete. Una descripción abreviada de esta función es

```
qplot(x, y=NULL, data, geom="auto", xlim = c(NA, NA), ylim =c(NA, NA))
```

Nota

- `x` : valores en el eje de abscisas.
- `y` : valores en el de ordenadas (opcional).
- `data` : `data.frame` de donde salen los datos (opcional).
- `geom` : elementos gráficos o geometrías (“`point`”,`line`”,`bar`”,...). Por defecto, “`point`” si `y` viene especificado, e “`histogram`” si sólo se especifica `x`.
- `xlim`, `ylim`: límites en los ejes de `x` e `y`.

Otros argumentos relacionados con los ejes y el título del gráfico son: `main`: título del gráfico; `xlab`, `ylab`: etiquetas los los ejes; `log`: ejes en escala log. Los valores permitidos son “`x`”, “`y`” o bien “`xy`”.

1.1. Distribución de una variable

Para intentar ver algo en un conjunto de datos, lo primero que se puede hacer es averiguar como se distribuyen sus valores. En R, hay funciones básicas (`summary`, `stem`, `table`, ...) que permiten tener una idea de esta distribución.

El tipo de representación gráfica de la distribución cambia según la naturaleza de la variable en estudio. Una variable cuantitativa toma valores numéricos. Para una variable cuantitativa, se suele recurrir a un **histograma** o bien a un **diagrama de caja** para describir su distribución. Mientras que para variables cualitativas (o categóricas), se utilizará un *diagrama de barras*.

1.1.1. Histograma

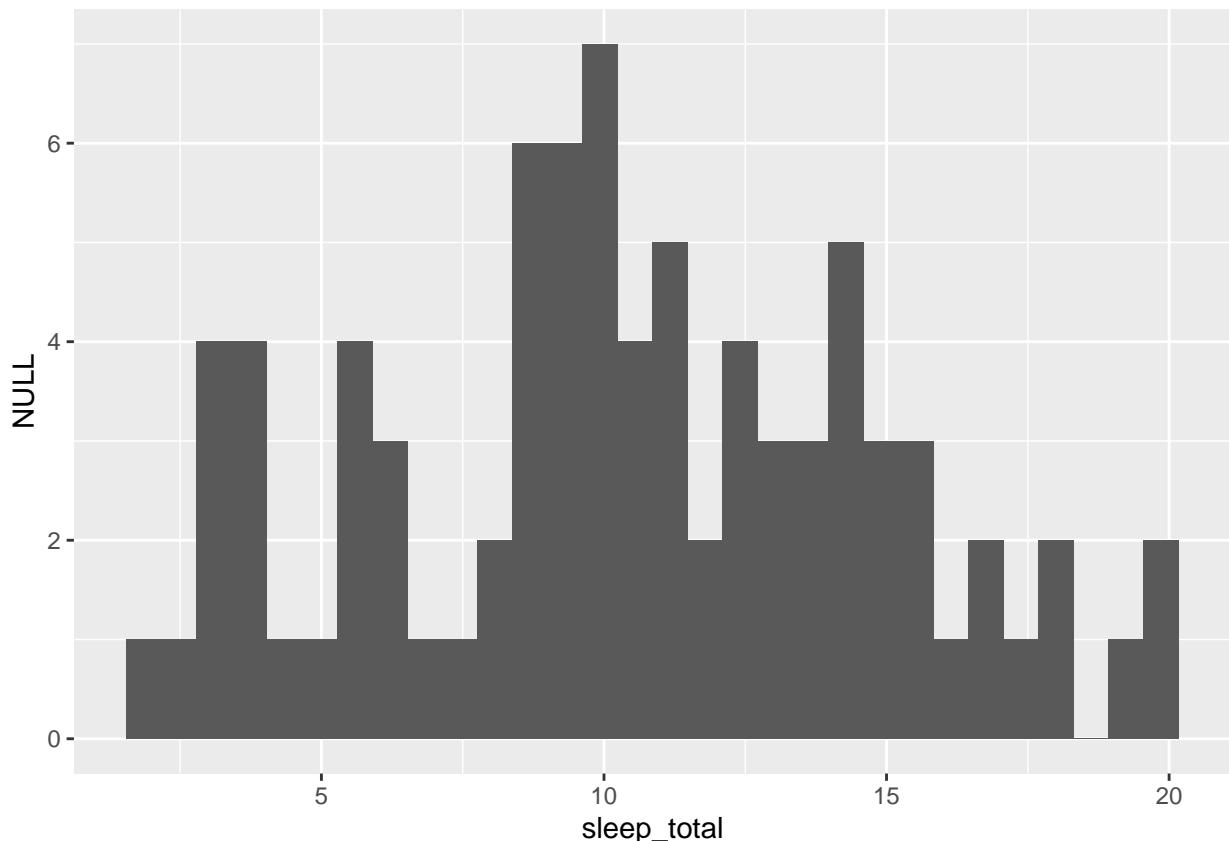
Para ilustrar la descripción gráfica de la distribución de una variable numérica, se utiliza la base de datos `msleep` que contiene información sobre el tiempo de sueño (en horas) de mamíferos:

```
msleep # ?msleep para más detalles
```

```
## # A tibble: 83 x 11
##   name  genus vore  order conservation sleep_total sleep_rem sleep_cycle
##   <chr> <chr> <chr> <chr> <chr>      <dbl>      <dbl>      <dbl>
## 1 Chee~ Acin~ carni Carn~ lc        12.1       NA       NA
## 2 Owl ~ Aotus omni Prim~ <NA>      17         1.8       NA
## 3 Moun~ Aplo~ herbi Rode~ nt       14.4       2.4       NA
## 4 Grea~ Blar~ omni Sori~ lc       14.9       2.3     0.133
## 5 Cow   Bos   herbi Arti~ domesticated 4         0.7     0.667
## 6 Thre~ Brad~ herbi Pilo~ <NA>     14.4       2.2     0.767
## 7 Nort~ Call~ carni Carn~ vu       8.7        1.4     0.383
## 8 Vesp~ Calo~ <NA> Rode~ <NA>      7         NA       NA
## 9 Dog   Canis carni Carn~ domesticated 10.1      2.9     0.333
## 10 Roe ~ Capr~ herbi Arti~ lc        3         NA       NA
## # ... with 73 more rows, and 3 more variables: awake <dbl>, brainwt <dbl>,
## #   bodywt <dbl>
```

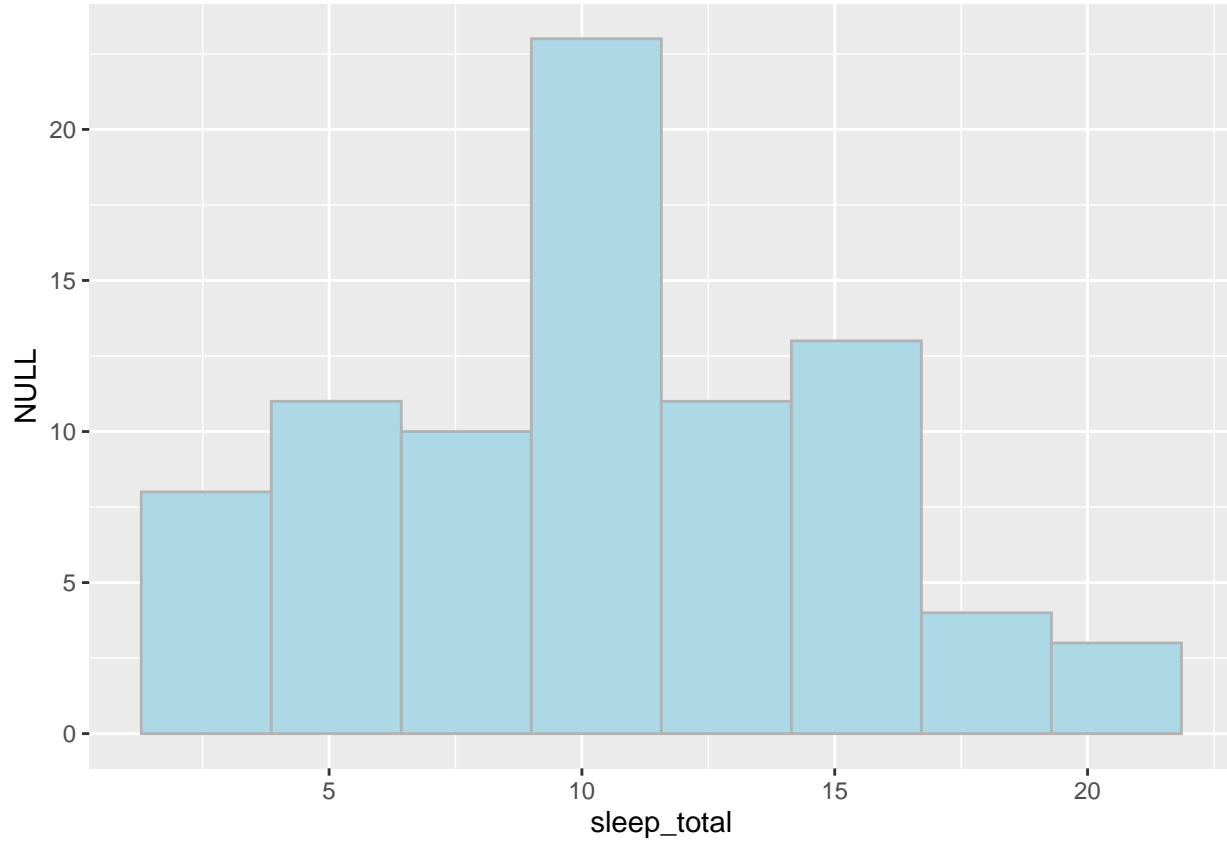
```
#summary(msleep$sleep_total) #tiempo total de sueño (en horas)
stem(msleep$sleep_total)
```

```
## 
##   The decimal point is at the |
## 
##   0 | 9
##   2 | 79013589
##   4 | 0423346
##   6 | 23307
##   8 | 03446779114456788
##  10 | 01113346900135
##  12 | 15555880578
##  14 | 234456996889
##  16 | 604
##  18 | 01479
qplot(sleep_total,data=msleep) #histograma
```



La altura de cada barra en el histograma es proporcional a la frecuencia de datos que caen en el intervalo correspondiente. Por defecto, en la función `qplot` el número de barras es igual a `bins=30`. Este valor es muy arbitrario. Otra alternativa consiste en elegir un número k de barras en función del tamaño muestral n , como por ejemplo, el criterio de Sturges ($k = 1 + \log_2(n)$) o el criterio de Rule ($k = 2n^{1/3}$). Abajo, un histograma con un número de barras que sigue este último criterio:

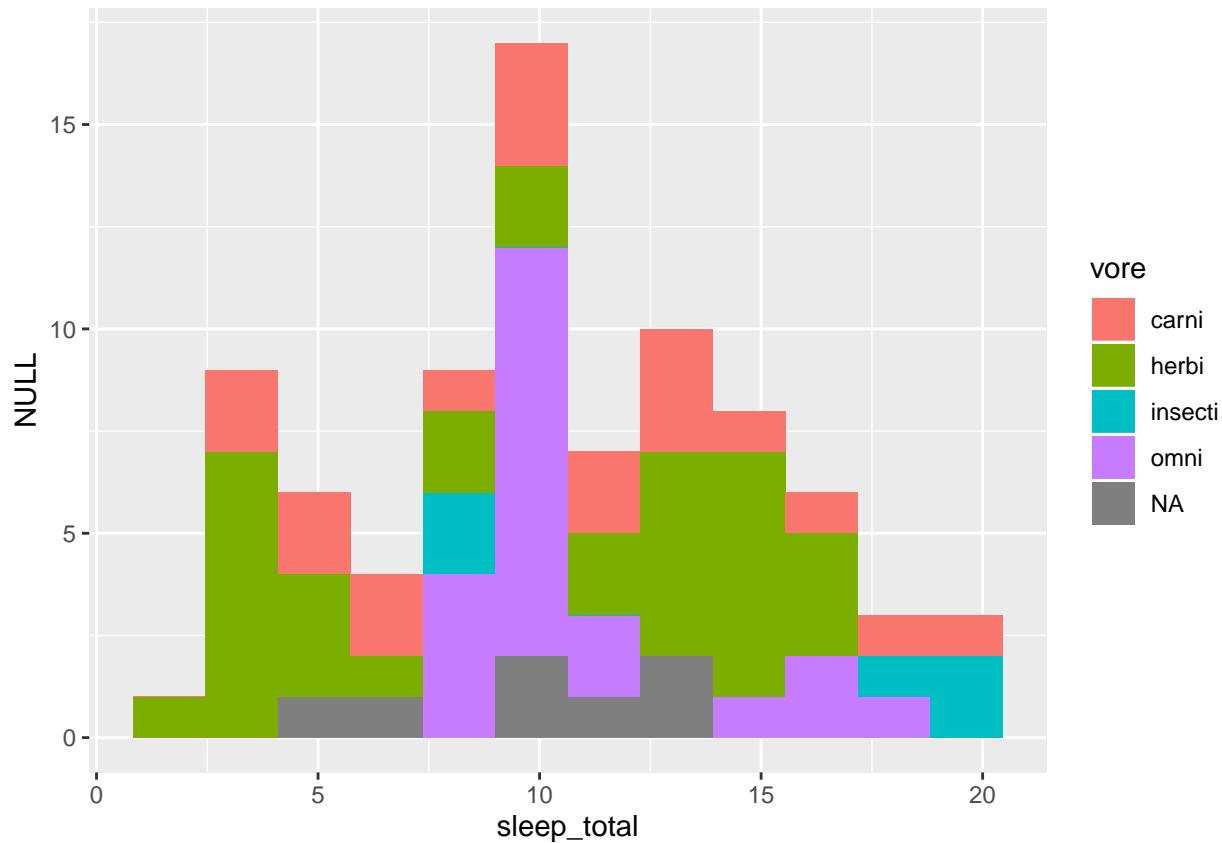
```
qplot(sleep_total,data=msleep,bins=8,color=I("grey70"),fill=I("lightblue"))
```



Nota

El argumento `fill` controla el color de relleno de las barras y el argumento `color` el color del borde. Para especificar un color concreto se utiliza la función `I()`. Si el color varía con otra variable `z`, se especifica esta dependencia escribiendo `fill=z`.

```
qplot(sleep_total, data=msleep, bins=12, fill=vore) #distribución del tiempo de sueño según dieta del mamífero
```

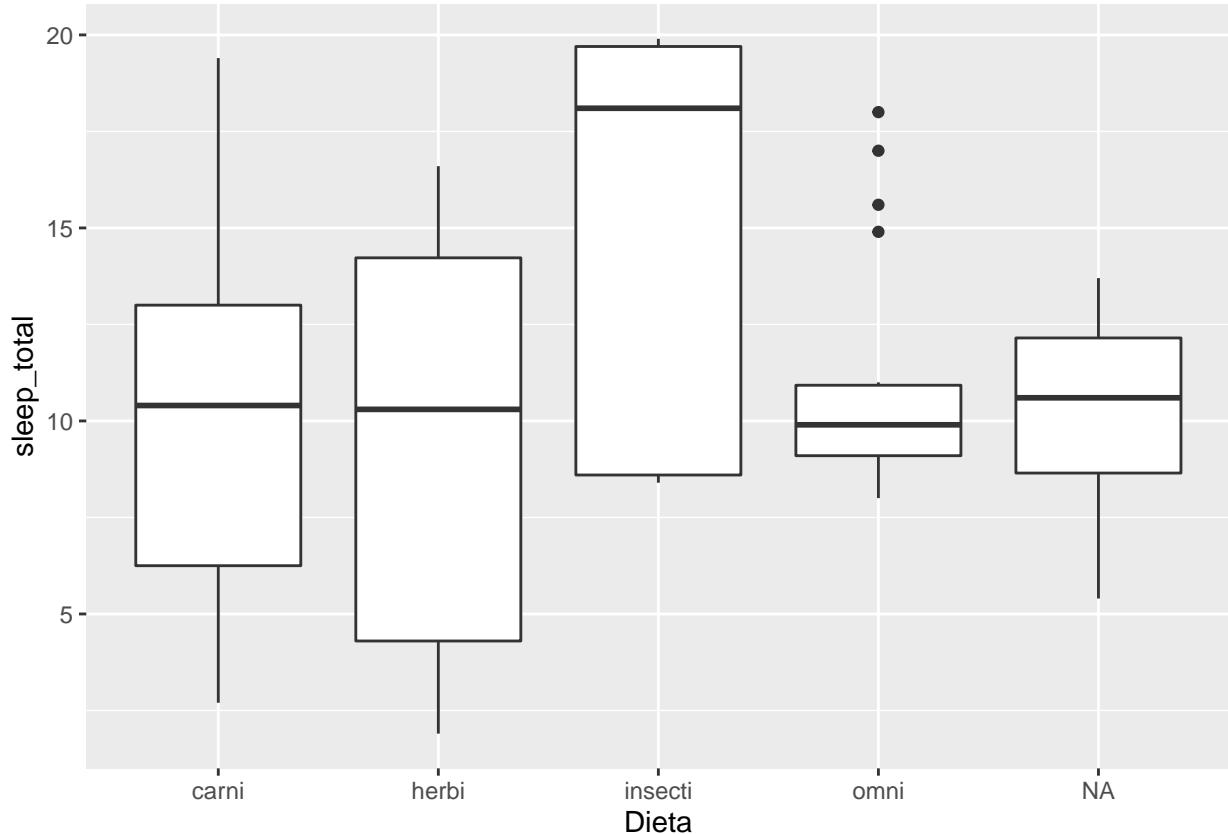


1.1.2. Diagrama de caja (boxplot)

Otra representación similar es el diagrama de caja. Este diagrama describe la distribución de una variable numérica mediante una caja y unos segmentos que acotan las regiones donde la variable tiene el grueso de sus valores. Esta representación es menos fina que la del histograma pero es más robusta (menos sensible a valores extremos).

Esta representación es especialmente adecuada cuando se quiere describir como varía la distribución de una variable numérica en función de una variable categórica. Así, la distribución del tiempo de sueño según la dieta del mamífero se puede representar de la siguiente manera:

```
qplot(vore,sleep_total,data=msleep,geom="boxplot",xlab="Dieta")
```



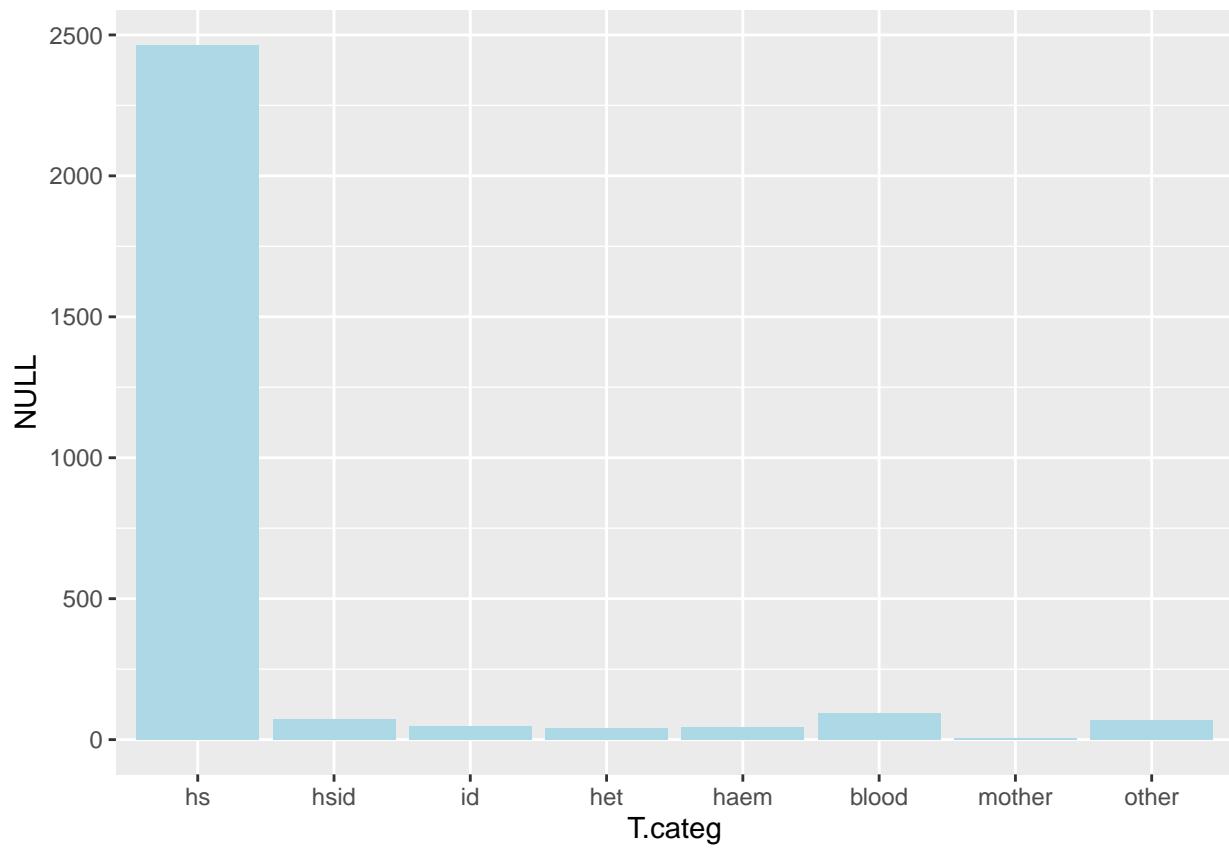
Ejercicio 1.1 Cargar la base de datos de la encuesta nacional americana `nhs` y representar la distribución del índice de masa corporal (imc) según el sexo o la raza.

```
load("data/nhs.RDA")
#View(nhs)
```

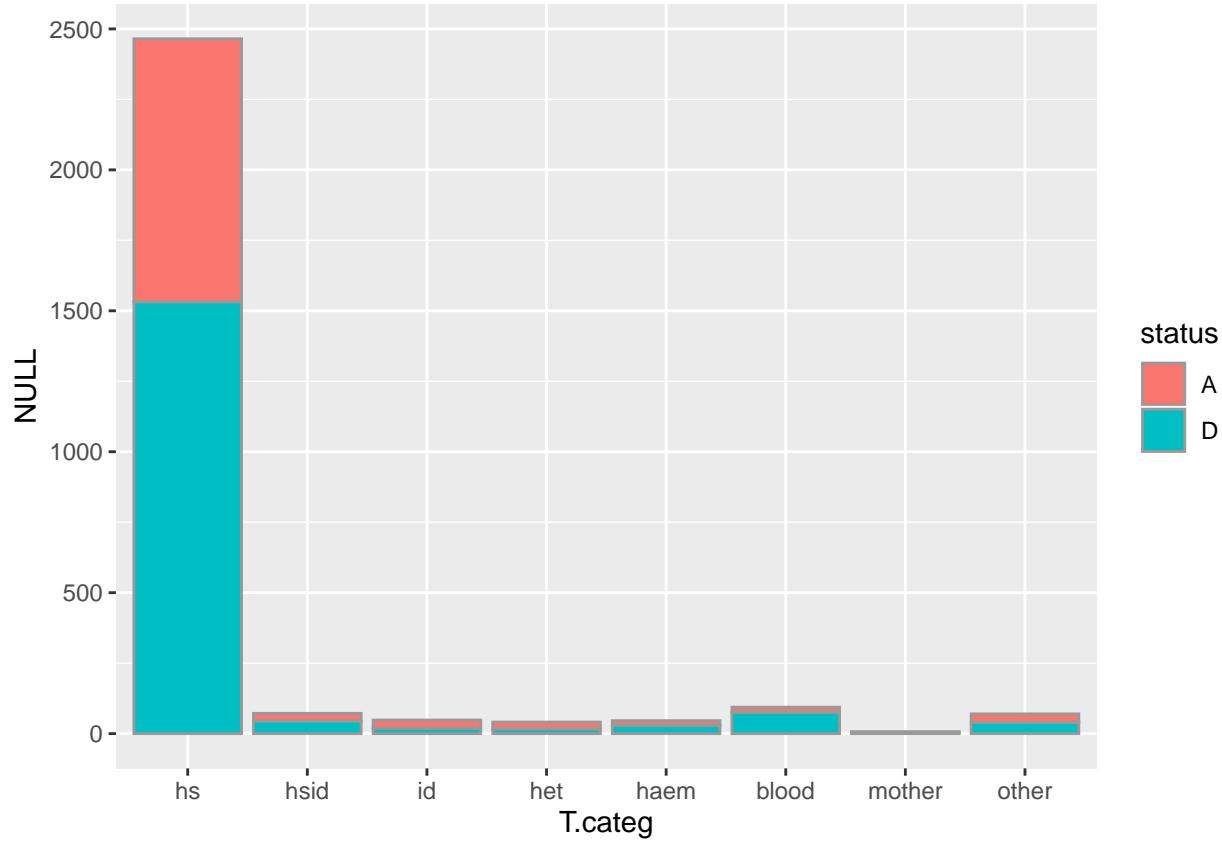
1.1.3. Diagrama de barras

Los diagramas de barras permiten representar la distribución de una variable categórica. En esta representación, cada categoría viene representada por una barra cuya altura es proporcional a su frecuencia en la muestra.

```
require(MASS)
qplot(T.categ,data=Aids2,fill=I("lightblue")) #Distribución de las categorias de transmisión en una muestra
```



```
qplot(T.categ, data=Aids2, fill=status, color=I("gray60")) #Distribución del estado vital del caso según la categoría
```



Utilizando el argumento `fill` se puede ver como varia esta distribución de acuerdo a otra variable (aquí la categoría de transmisión). El gráfico obtenido resulta poco claro y veremos más adelante como mejorarlo.

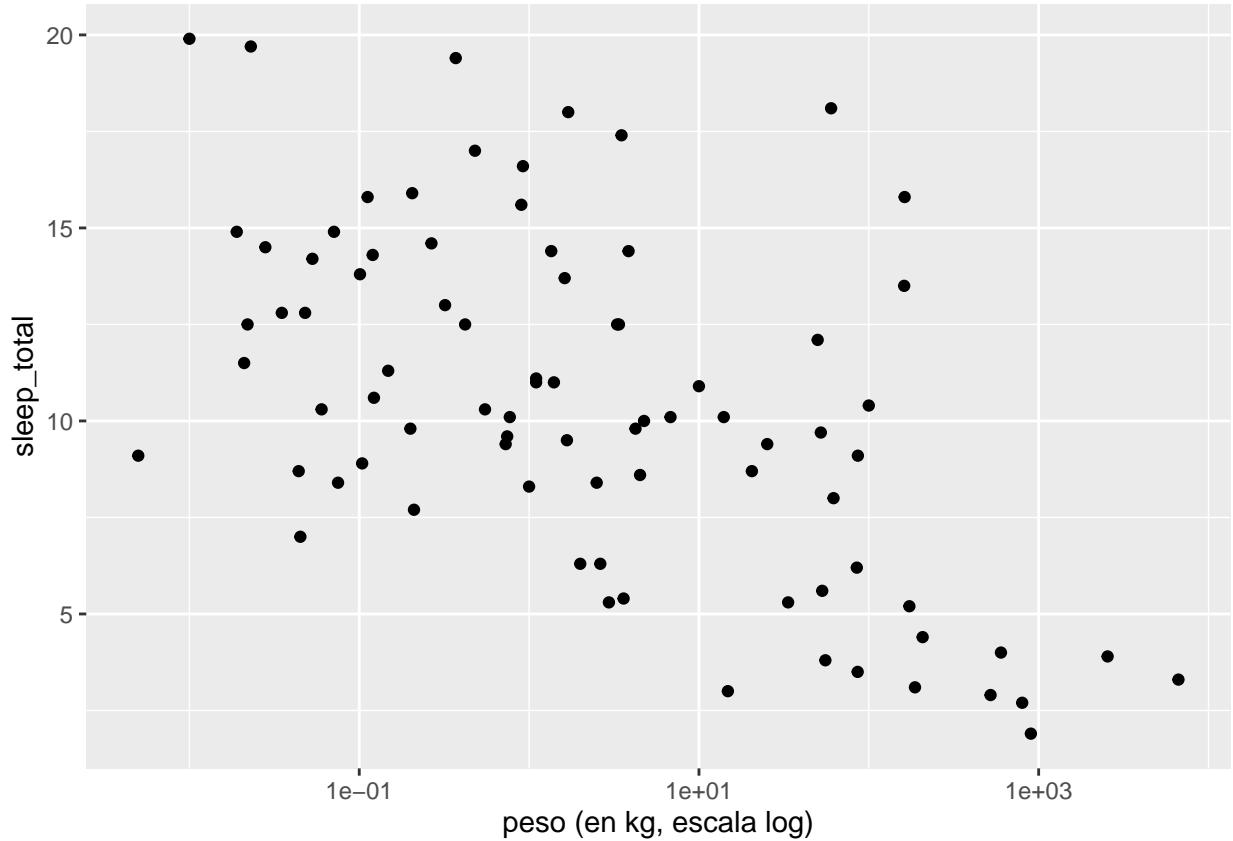
Ejercicio 1.2 Representar la distribución de los hábitos con el tabaco según el sexo, utilizando la muestra de la encuesta nacional americana.

1.2. Relación entre dos variables

1.2.1. Diagrama de dispersión

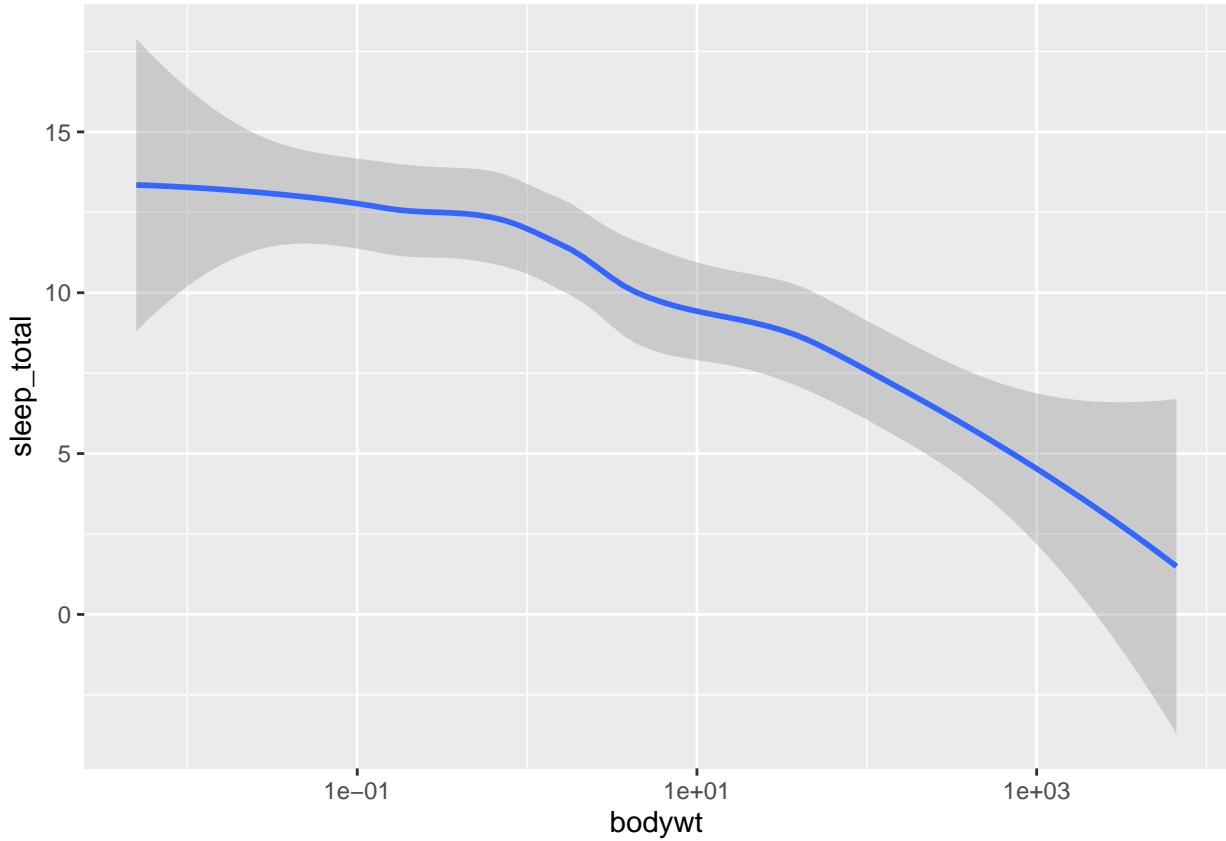
Para describir la relación entre dos variables cuantitativas se suele utilizar gráficos de dispersión. Estos gráficos describen esta relación mediante una nube de puntos en un plano cartesiano. Cada punto de la nube corresponde a una fila de la base de datos y cada una de las variables corresponde a un eje. En el gráfico siguiente se describe la relación entre las horas de sueño y el peso del animal:

```
qplot(bodywt,sleep_total,data=msleep,xlab="peso (en kg, escala log)",log="x")
```



Ajustando una curva suave (“smooth”) a la nube de puntos, se puede apreciar mejor la tendencia en esta relación:

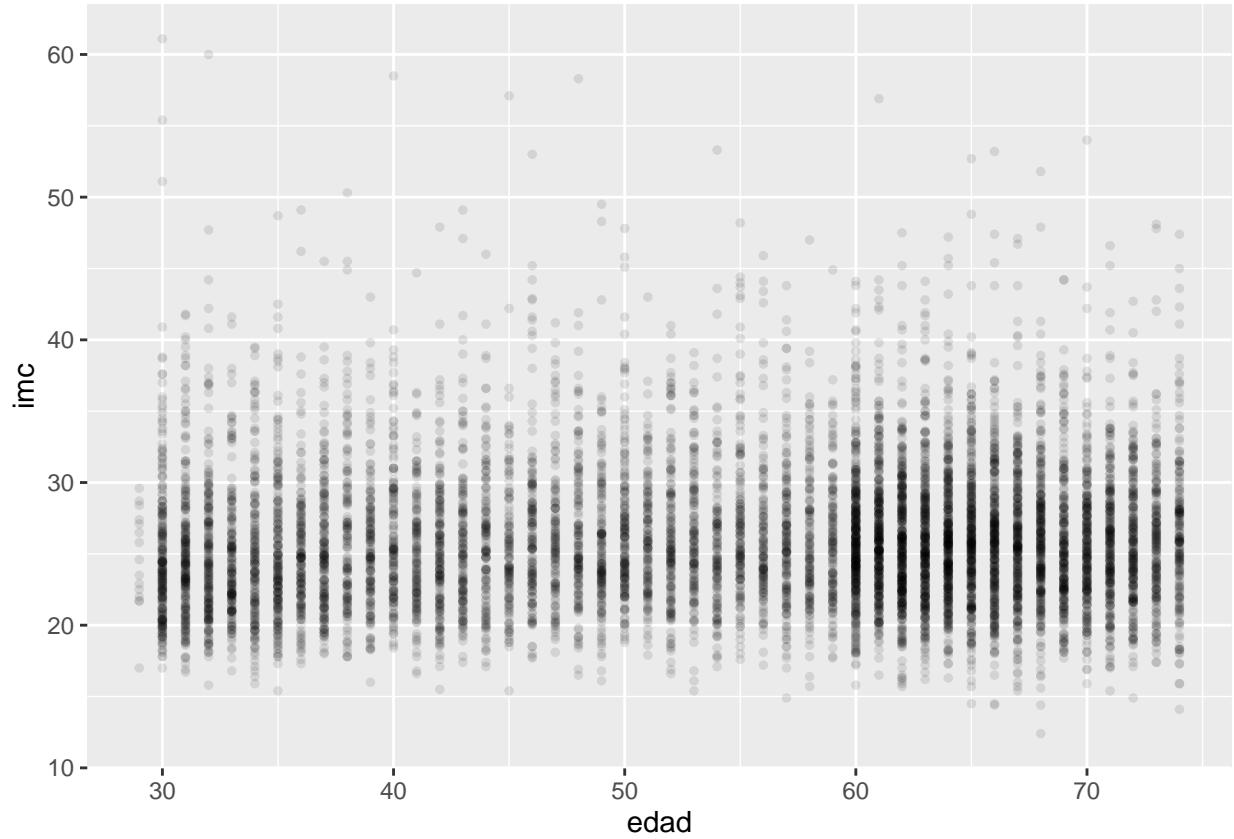
```
qplot(bodywt,sleep_total,data=msleep,log="x",geom="smooth")
```



```
#qplot(bodywt,sleep_total,data=msleep,log="x",geom="smooth", method="lm") ## para ayudar una recta; "lm"
```

Para evitar los problemas de solapamiento de puntos en la representación se puede poner algo de ruido en los datos (`geom="jitter"`), jugar con el tamaño de los puntos (`size`) o utilizar el parámetro de transparencia (`alpha`):

```
qplot(edad, imc,data=nhs,alpha=I(.1),size=I(1))
```



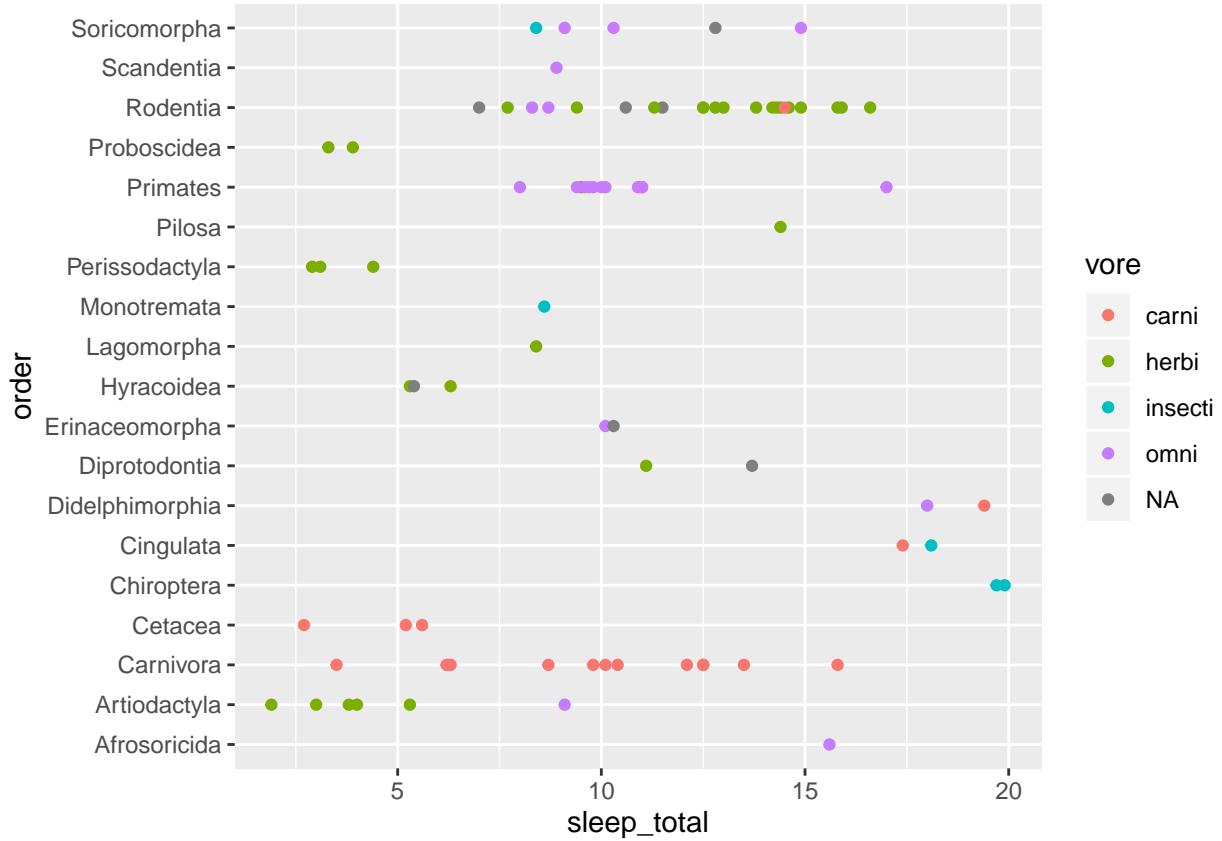
```
#qplot(edad, imc, data=nhs, alpha=I(.1), size=I(1), geom="jitter") #Mejor resultado añadiendo algo de ruido
```

Ejercicio 1.3 Utilizando la encuesta nacional americana, describir la relación entre la edad y el IMC. En un mismo gráfico, describir como esta relación cambia con el sexo.

1.2.2. Dotchart

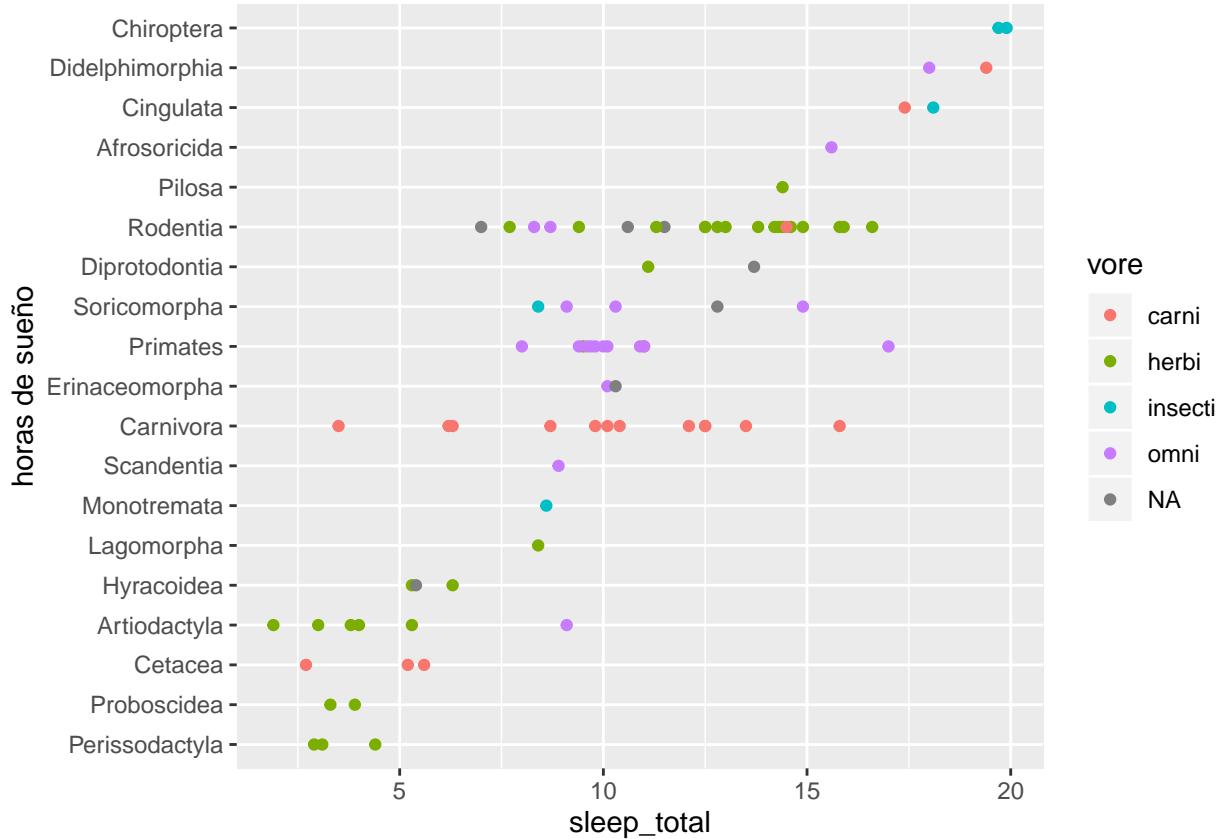
Si una de las variables es categórica y tiene muchas categorías, el gráfico de dispersión puede ser también apropiado:

```
qplot(sleep_total, order, data=msleep, col=vore)
```



Pero, es recomendable para mayor claridad ordenar la variable categórica de acuerdo a la otra variable:

```
qplot(sleep_total,reorder(order,sleep_total),data=msleep,col=vore,ylab="horas de sueño")
```

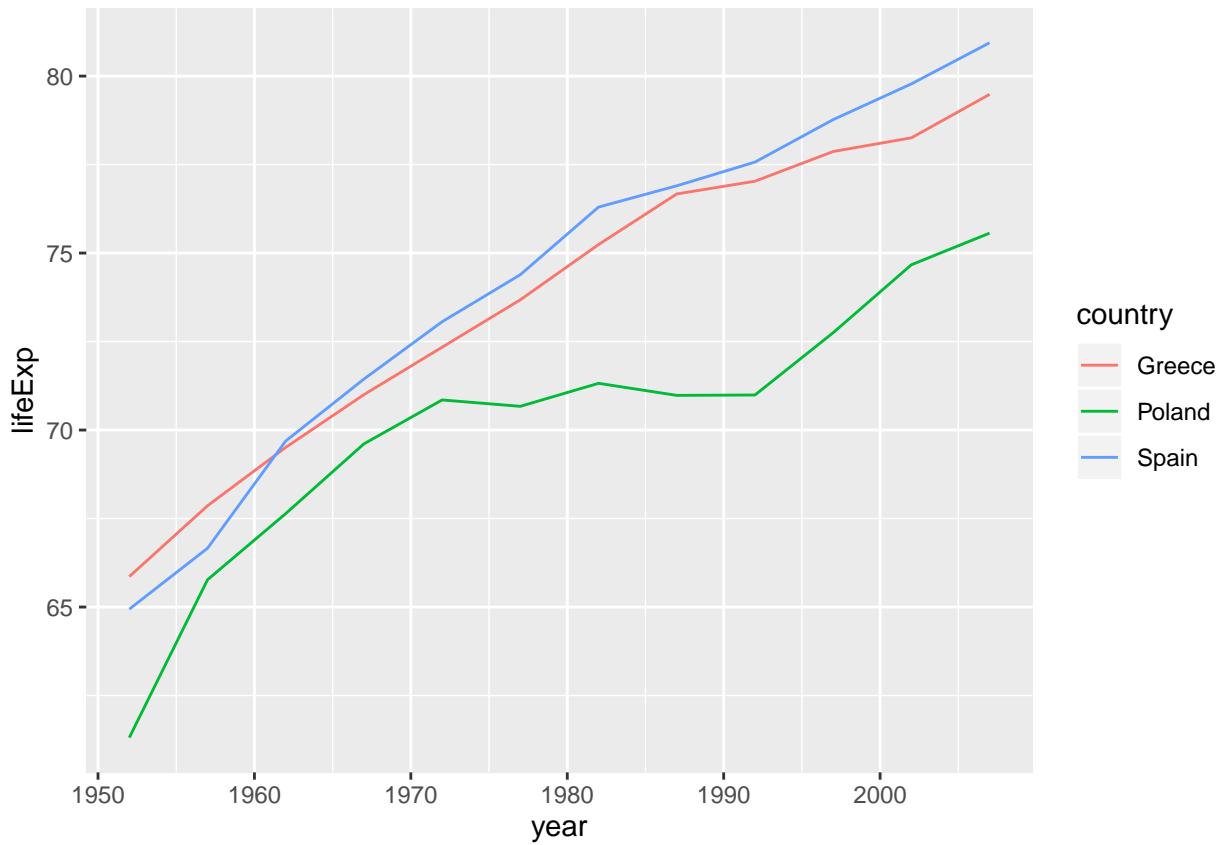


Ejercicio 1.4 Describir con un gráfico similar al anterior, los datos de la base de datos `islands` sobre superficies de islas. Puede ser oportuno recurrir a una escala log.

1.2.3. Relación con una variable temporal

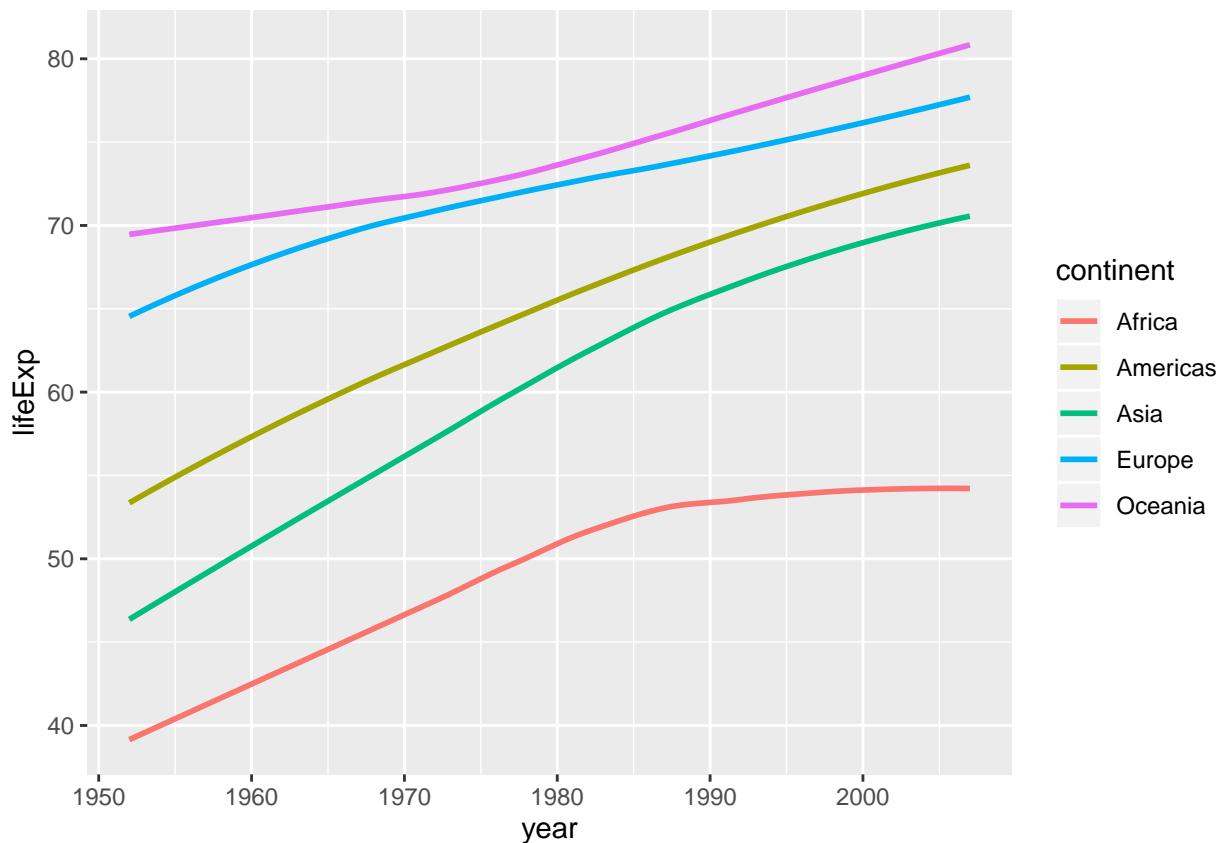
Si una de las variables es el tiempo, a menudo es conveniente recurrir a líneas (`geom="line"`) en vez de puntos para representar la evolución de la otra variable.

```
require(gapminder) #base de datos sobre esperanza de vida según características socio-demográficas del mundo
datos=subset(gapminder,country %in% c("Spain","Greece","Poland")) #sólo se consideran España, Grecia y Polonia
qplot(year,lifeExp,data=datos,geom="line",color=country)
```



Sin embargo para representar la tendencia se utilizará la opción `geom="smooth"`:

```
qplot(year,lifeExp,data=gapminder,geom="smooth",color=continent,se=FALSE) #se=FALSE para quitar intervalos
```



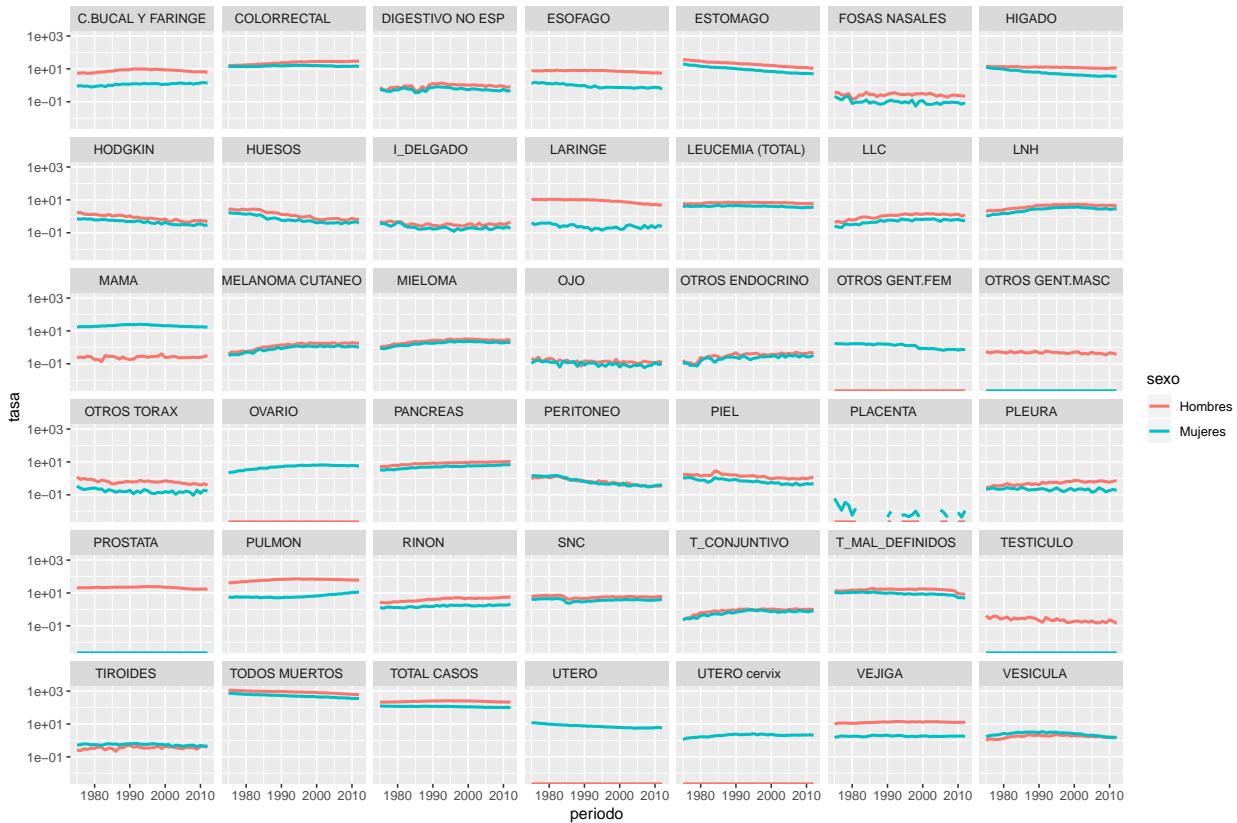
Ejercicio 1.5 Describir con gráficos similares a los anteriores, los datos de la base de datos `cancer` sobre la evolución de las tasas de mortalidad por cáncer en España en el periodo 1975-2012. Empezar por ejemplo, representando la evolución de la mortalidad por cáncer de pulmón según el sexo.

2. Gráficos avanzados

Esta sección es una introducción a las ideas del libro *The Grammar of Graphics* de Leland Wilkinson (2005) tal y como vienen implementadas en el paquete `ggplot2` de Hadley Wickham (2009).

A continuación veremos los elementos principales de esta gramática y como nos permiten elaborar de manera sencilla representaciones visuales del comportamiento de una variable de interés a través de los distintos niveles de otras.

El gráfico siguiente muestra la evolución de la tasa de mortalidad por cáncer según la localización, en España durante el periodo 1975-2012.



2.1. Elementos de un gráfico

El gráfico anterior, se obtuvo de la siguiente manera. En primer lugar, se carga los datos de mortalidad:

```
require(data.table)
load("data/cancer.RDA") #carga los datos
cancer
```

```
##           sexo periodo      tumor     tasa
## 1: Hombres   1975 C.BUCAL Y FARINGE 5.541879
## 2: Hombres   1976 C.BUCAL Y FARINGE 5.512168
## 3: Hombres   1977 C.BUCAL Y FARINGE 5.827874
## 4: Hombres   1978 C.BUCAL Y FARINGE 5.323451
## 5: Hombres   1979 C.BUCAL Y FARINGE 5.461059
##   ---
## 3188: Mujeres 2008 TODOS MUERTOS 385.294977
## 3189: Mujeres 2009 TODOS MUERTOS 373.009470
## 3190: Mujeres 2010 TODOS MUERTOS 359.444478
## 3191: Mujeres 2011 TODOS MUERTOS 357.461639
## 3192: Mujeres 2012 TODOS MUERTOS 359.691700
```

La expresión de ggplot2 para crear el gráfico fue:

```
ggplot(cancer) +
  aes(x = periodo, y = tasa, col = sexo) +
  geom_line(size=1) + scale_y_log10() +
  facet_wrap(~ tumor)
```

Esta expresión combina varios elementos que discutiremos con detalle más adelante:

- **Datos:** siempre un “data.frame”
- **Estéticas:** elementos representables gráficamente (la posición x e y, el color, la forma, ...) en columnas del data.frame.
- **Geometrías** (o capas): puntos, rectas, histogramas, densidades, etc. También se llaman capas porque pueden superponerse.
- **Facetas:** parten un gráfico en sublienzo preservando las escalas (pequeños múltiplos)

2.1.1. 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`.

Por otro lado, es preferible que los datos estén en un formato “largo” (long format), es decir, una columna para cada dimensión y una fila para cada observación. Para ilustrar esta idea, se considera la base de datos `VADeaths` que proporciona tasas de mortalidad (por 1000 personas/año) en Virginia (1940) por grupos socio-demográficos y de edad.

	Rural Male	Rural Female	Urban Male	Urban Female
50-54	11.7	8.7	15.4	8.4
55-59	18.1	11.7	24.3	13.6
60-64	26.9	20.3	37.0	19.3
65-69	41.0	30.9	54.6	35.1
70-74	66.0	54.3	71.1	50.0

Antes de representar los datos, convertimos la base en un formato alargado.

```
require(data.table)
mortalidad = data.table(melt(VADeaths))
names(mortalidad) <- c("edad", "grupo", "tasa")
mortalidad

##      edad      grupo tasa
## 1: 50-54 Rural Male 11.7
## 2: 55-59 Rural Male 18.1
## 3: 60-64 Rural Male 26.9
## 4: 65-69 Rural Male 41.0
## 5: 70-74 Rural Male 66.0
## 6: 50-54 Rural Female 8.7
## 7: 55-59 Rural Female 11.7
## 8: 60-64 Rural Female 20.3
## 9: 65-69 Rural Female 30.9
## 10: 70-74 Rural Female 54.3
## 11: 50-54 Urban Male 15.4
## 12: 55-59 Urban Male 24.3
## 13: 60-64 Urban Male 37.0
## 14: 65-69 Urban Male 54.6
## 15: 70-74 Urban Male 71.1
## 16: 50-54 Urban Female 8.4
## 17: 55-59 Urban Female 13.6
## 18: 60-64 Urban Female 19.3
## 19: 65-69 Urban Female 35.1
## 20: 70-74 Urban Female 50.0

p <- ggplot(mortalidad)
```

El código anterior crea un objeto, `p` que viene a ser un proto-gráfico: contiene los datos que vamos a utilizar, los del conjunto de datos `mortalidad`. Obviamente, el código anterior es insuficiente para crear un gráfico:

aún no hemos indicado qué queremos hacer con `mortalidad`.

2.1.2. Estéticas (aes)

En un conjunto de datos hay columnas: edad, altura, ingresos, temperatura, etc. En un gráfico hay, en la terminología de `ggplot2`, *aesthetic*. 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.

```
p <- p + aes(x = edad, y = tasa, colour = grupo)
```

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

- La abscisa `x`, vendrá dada por el grupo de edad.
- La ordenada `y`, por la tasa de mortalidad.
- El color, por el grupo socio-demográfico.

Al *protográfico* se le han sumado las estéticas. En las secciones 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(mortalidad, aes(x = edad, y = tasa, colour = grupo))
```

El objeto `p` resultante aún no es un gráfico ni se puede representar. Le faltan capas, que es el objeto de la siguiente sección. No obstante, se puede inspeccionar la relación (o *mapeo*) entre estéticas y columnas de los datos:

```
p$mapping
```

```
## Aesthetic mapping:  
## * `x`      -> `edad`  
## * `y`      -> `tasa`  
## * `colour` -> `grupo`
```

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

- `x` e `y`, coordenadas 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. También muy útil para el solapamiento de puntos.
- `fill`, para el color de relleno de las formas sólidas (barras, etc.).

Nota

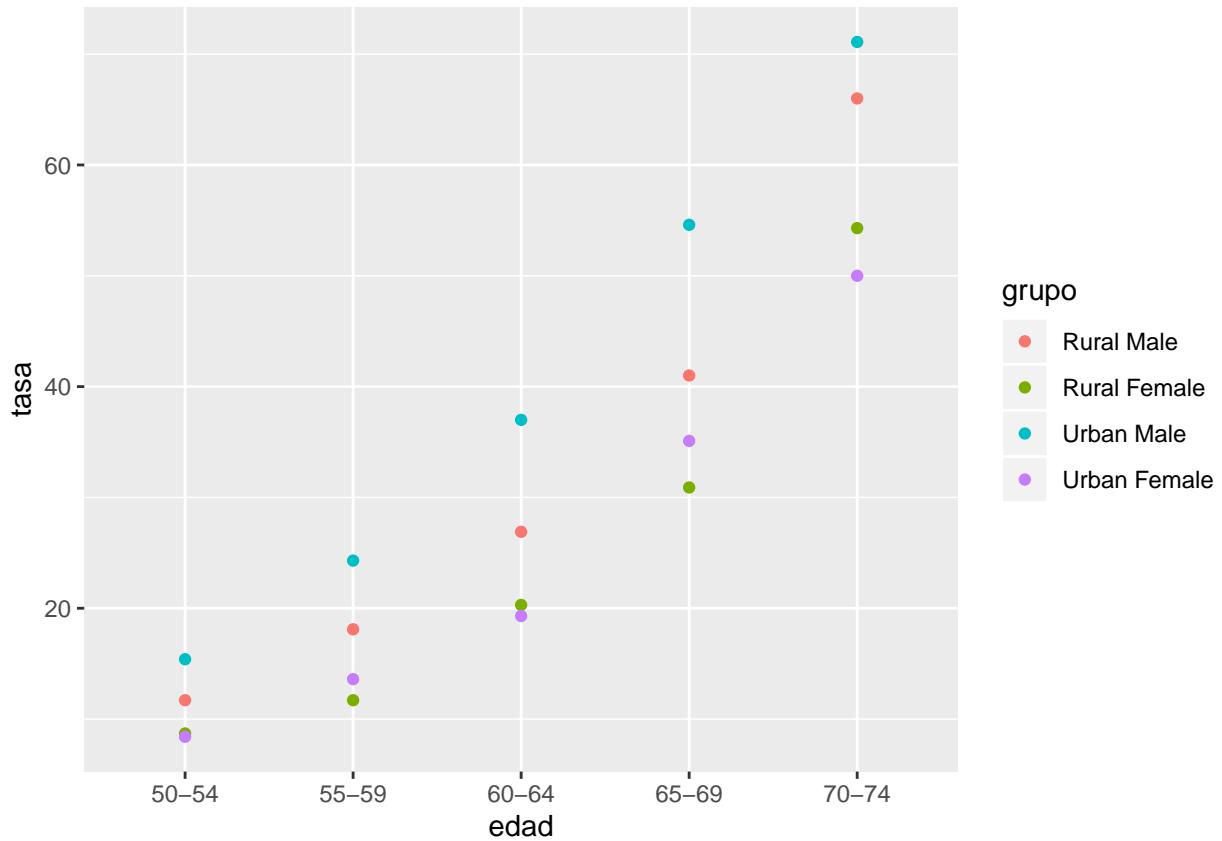
Hay que advertir que no todas las *estéticas* tienen la misma potencia en un gráfico. El ojo humano percibe fácilmente longitudes distintas. Pero tiene problemas para comparar áreas (que es lo que regula la estética `size`) o intensidades de color. Se recomienda usar las estéticas más potentes para representar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

2.1.3. Capas (geoms)

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:

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

```
p
```

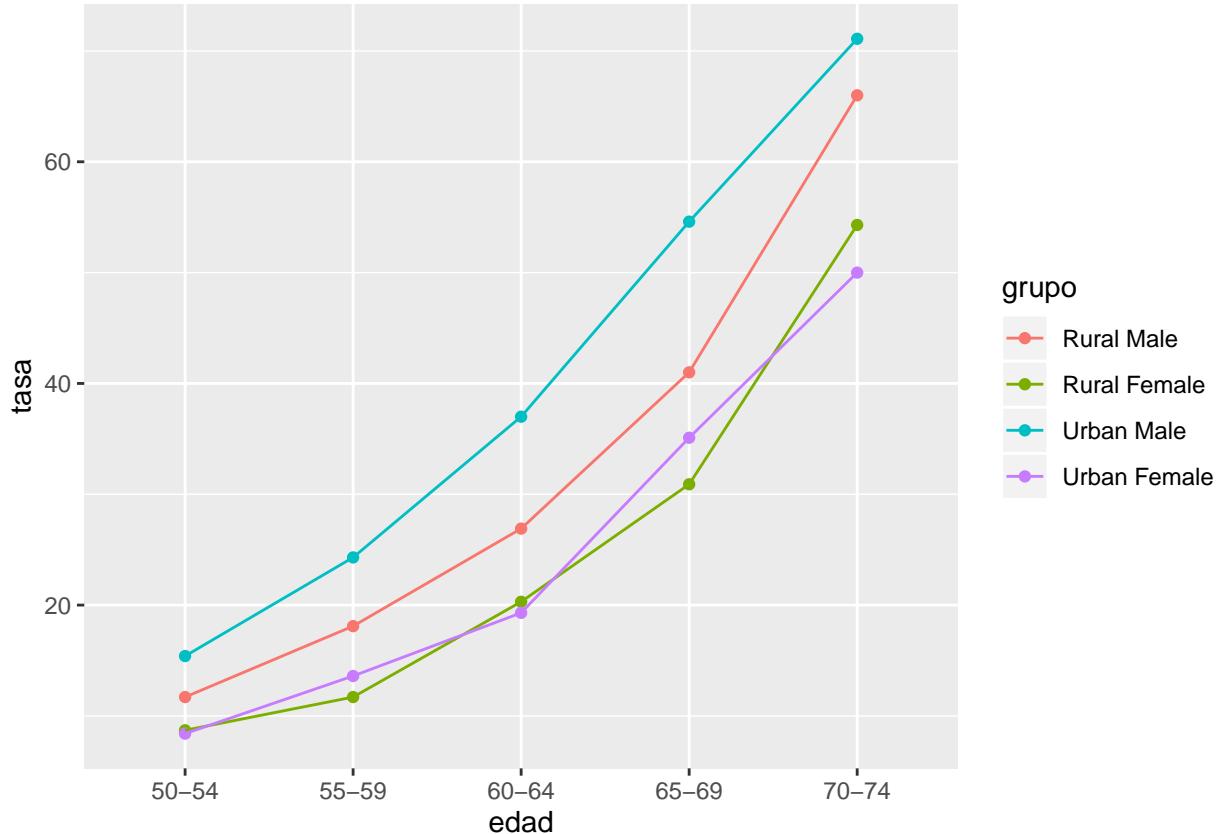


Una vez añadida una capa al gráfico, este puede pintarse (que es lo que ocurre al llamar a p). Se obtiene el mismo resultado haciendo, en una única línea,

```
ggplot(mortalidad, aes(x = edad, y = tasa, colour = grupo)) + geom_point()
```

Una característica de las capas, y de ahí su nombre, es que pueden superponerse. Por ejemplo,

```
ggplot(mortalidad, aes(x = edad, y = tasa, colour = grupo, group= grupo)) +
  geom_point() + geom_line()
```

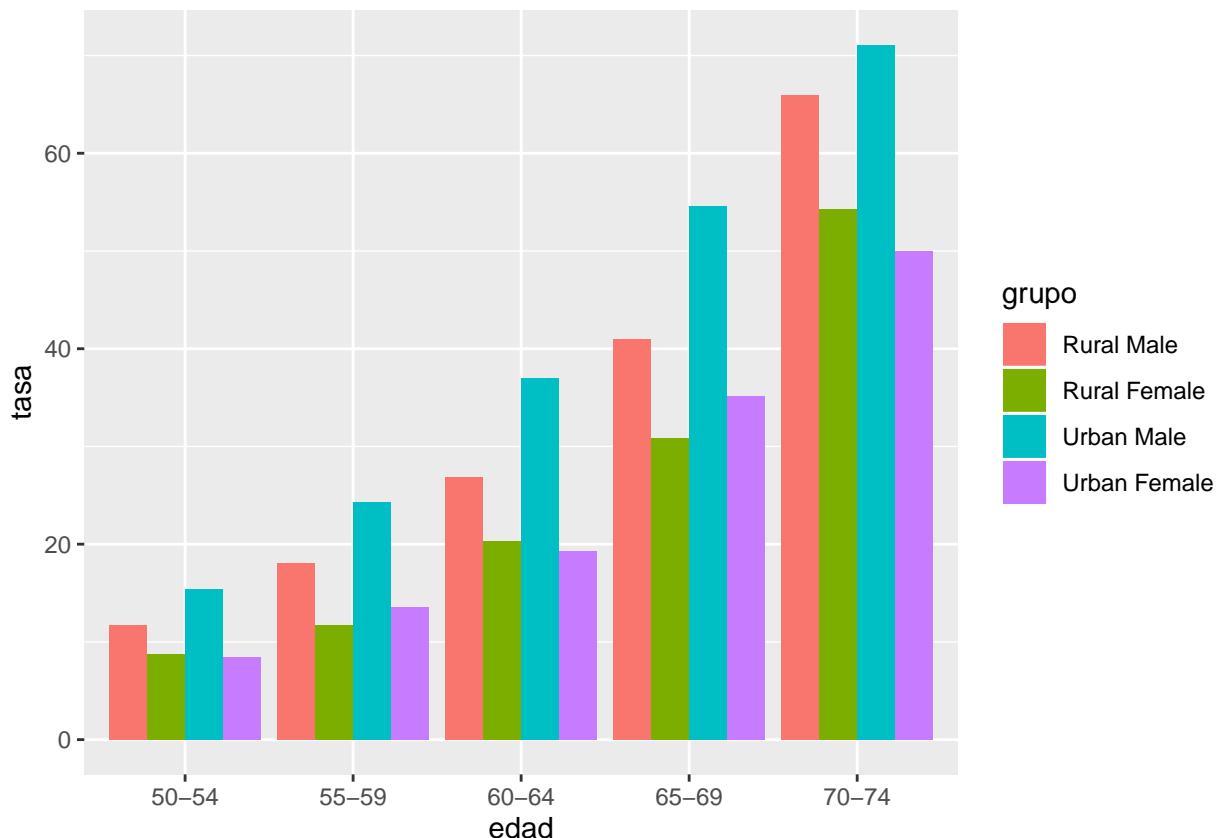


```
# Se requiere la estética `group` para conectar los puntos de una linea cuando la variable en abscisa es
```

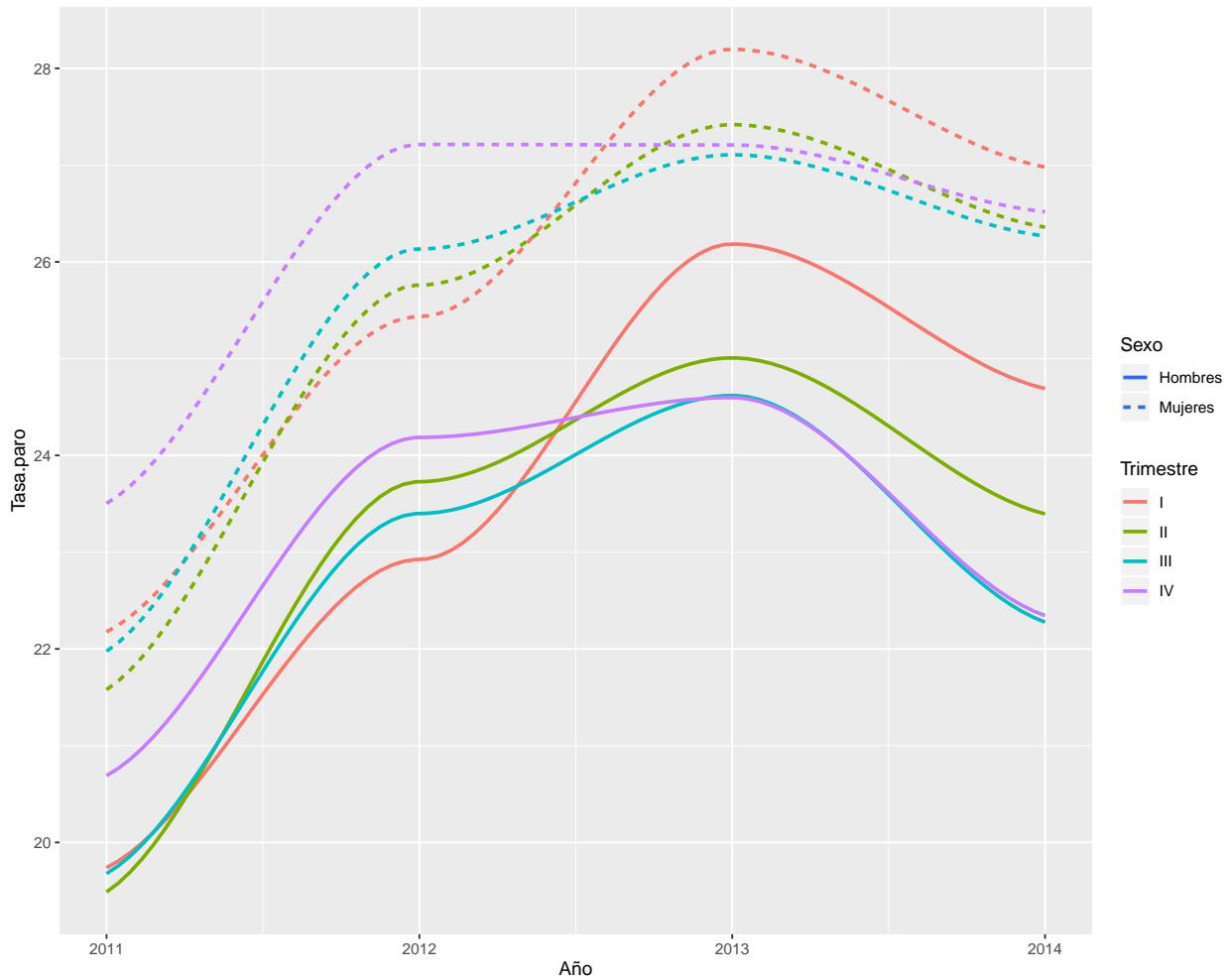
Existen muchos tipos de capas. Los más usuales son `geom_point`, `geom_line`, `geom_histogram`, `geom_bar` y `geom_boxplot` (ver las páginas <http://docs.ggplot2.org/current/>) para una lista actualizada.

Abajo una representación mediante un diagrama de barra de los datos anteriores:

```
ggplot(mortalidad, aes(x = edad, y = tasa, fill = grupo)) +
  geom_bar(stat="identity", position="dodge")
```



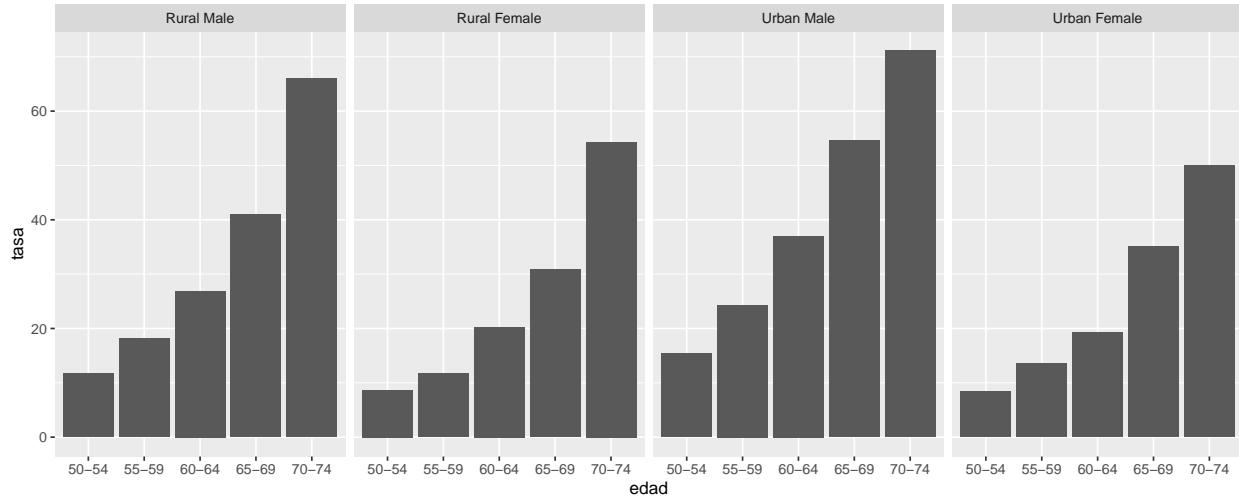
Ejercicio 2.1 Elaborar el siguientes gráfico sobre la evolución del paro en España. Utilizar la capa `geom_smooth` para suavizar la tendencia y la estética `linetype` para distintos tipos de curvas.



2.1.4. Facetas

Las facetas permiten subdividir un gráfico y suele ser un recurso muy eficiente para describir el comportamiento de una variable en función de otra variable categórica. Así por ejemplo, con los datos de mortalidad,

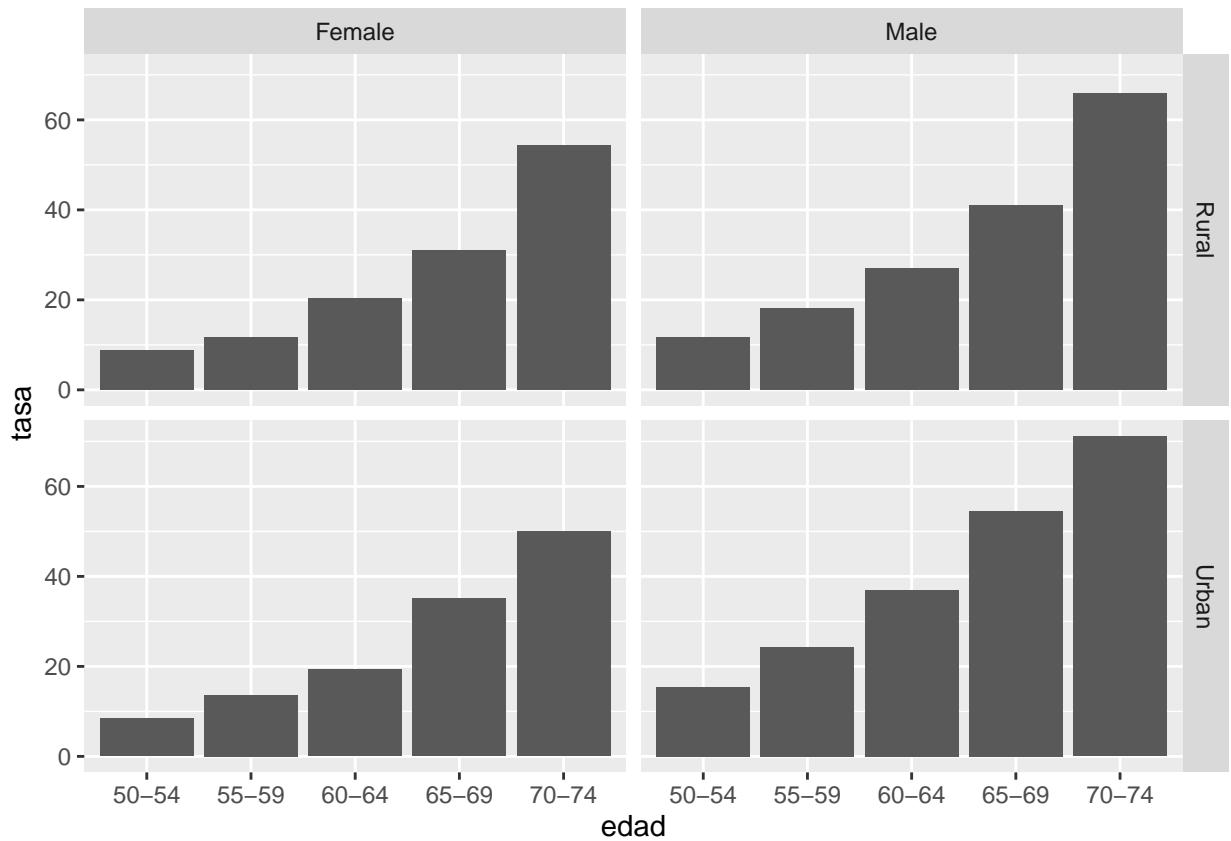
```
ggplot(mortalidad, aes(x = edad, y = tasa)) +
  geom_bar(stat="identity") +
  facet_grid(~grupo)
```



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.

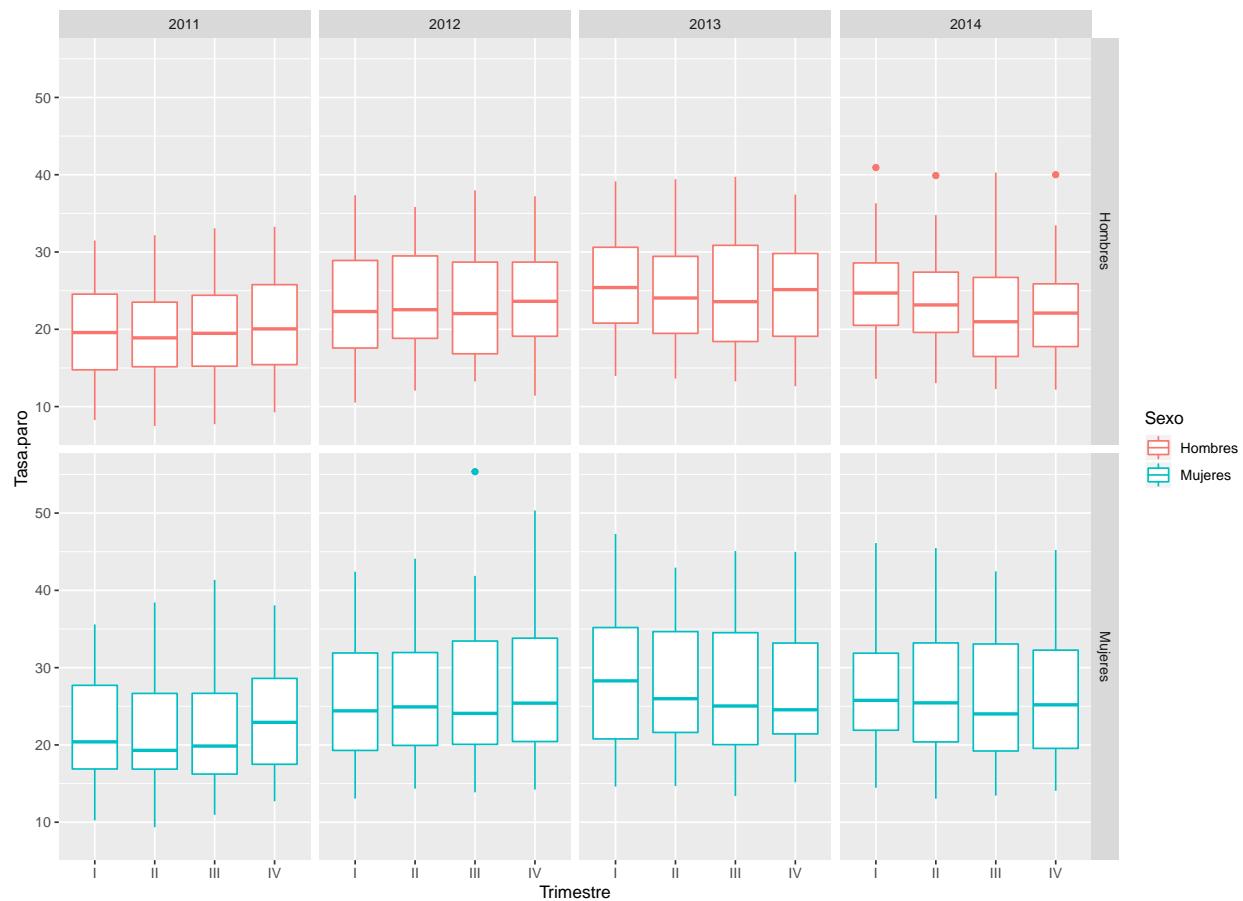
Los gráficos podrían disponerse verticalmente reemplazando `facet_grid(~grupo)` por `facet_grid(grupo~)` en el código anterior. Además, se puede subdividir el lienzo por dos (yo más!) variables así:

```
mortalidad[,c("zone", "sex")]:= tstrsplit(grupo, " ")
ggplot(mortalidad, aes(x = edad, y = tasa)) +
  geom_bar(stat="identity") +
  facet_grid(zone ~ sex)
```



Ejercicio 2.2 Elaborar los siguientes gráficos sobre la evolución del paro. Para el segundo gráfico, se puede filtrar la base original con el comando:

```
paro.ZHT <- subset(paro, Provincia %in% c("Zaragoza", "Huesca", "Teruel"))
```





En caso de haber muchas categorías (p.e., todas las provincias), puede usarse la función `facet_wrap` para distribuir las subgráficas en una cuadrícula (ver gráfico a principio de la sección).

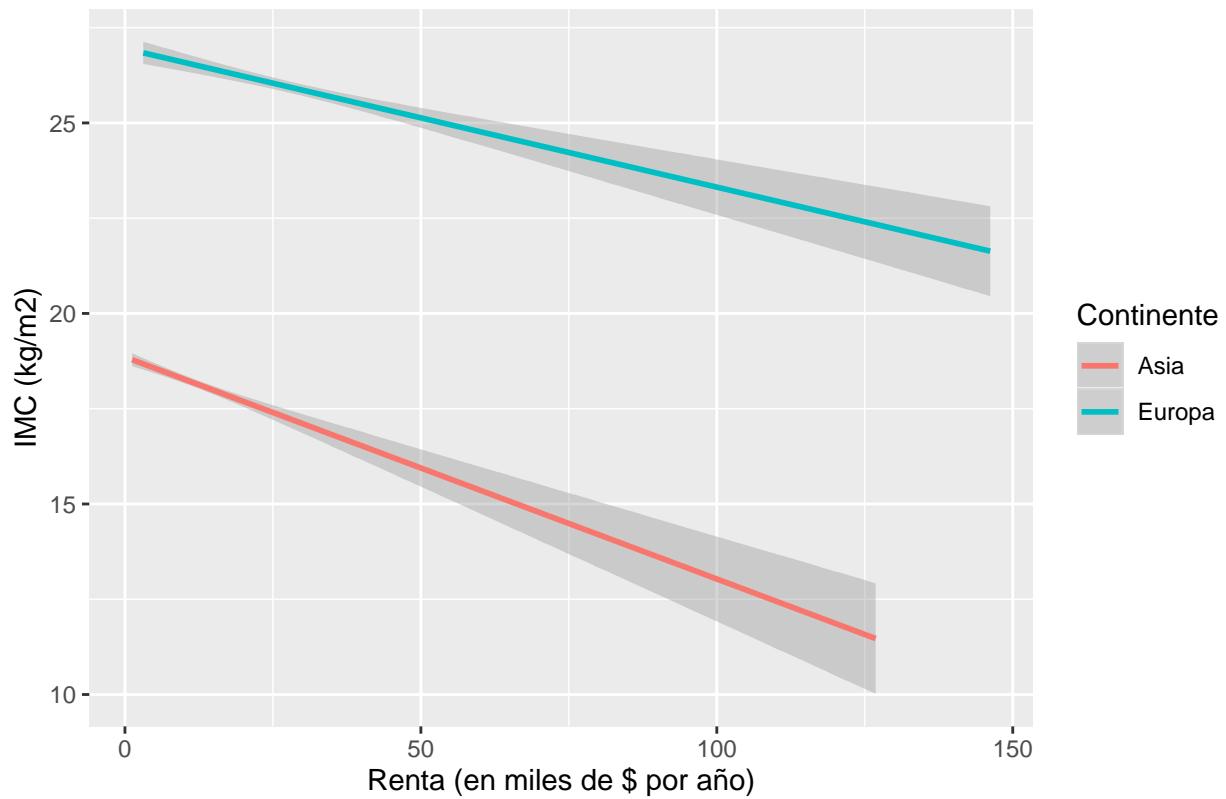
2.1.5. Toque final

2.1.5.1. Etiquetas

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 se pueden re-etiquetar los ejes y la leyenda escribiendo

```
obesidad<-fread("data/obesidad.csv")
p<-ggplot(obesidad,aes(x=renta,y=imc,color=region))+geom_smooth(method="lm")
p + ggtitle("Relación entre Indice de Masa Corporal (IMC) y renta") +
    labs(x = "Renta (en miles de $ por año)", y = "IMC (kg/m2)", color = "Continente")
```

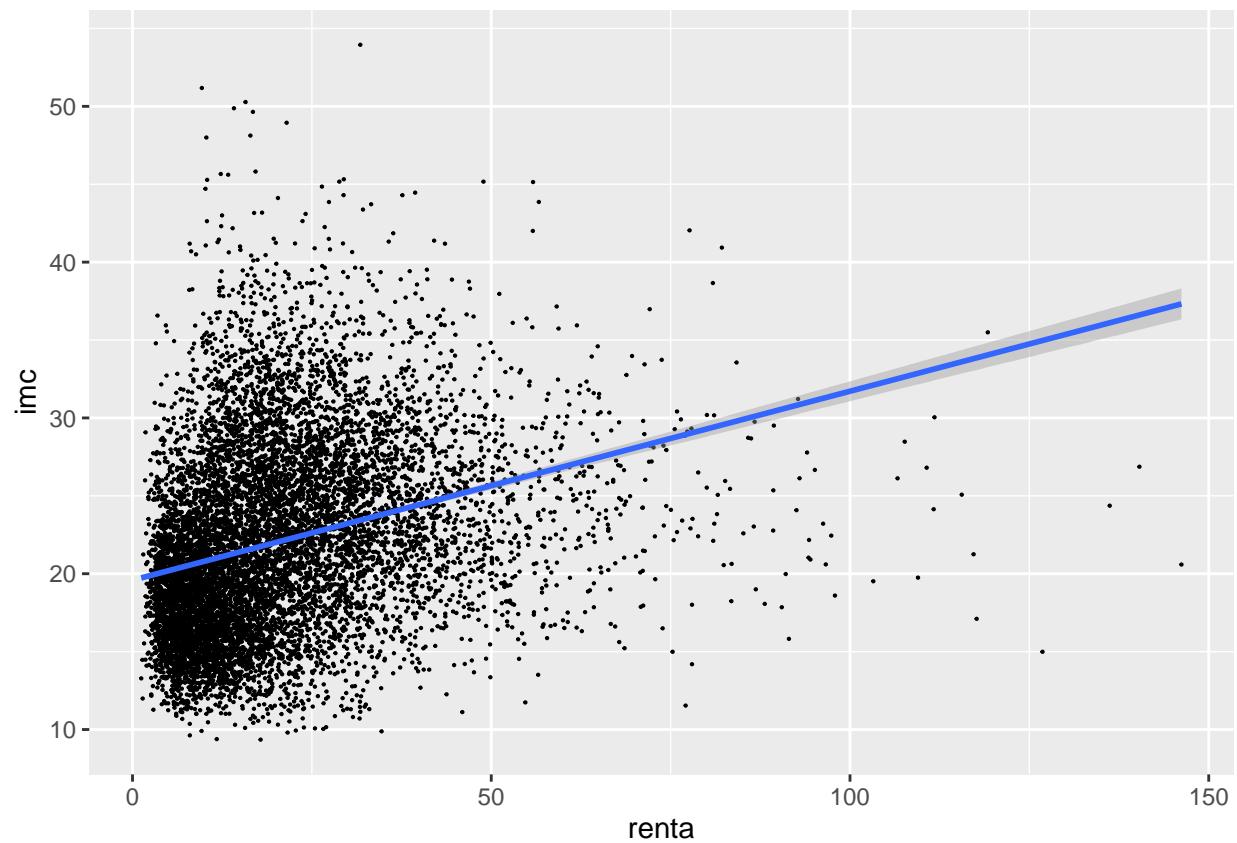
Relación entre Indice de Masa Corporal (IMC) y renta

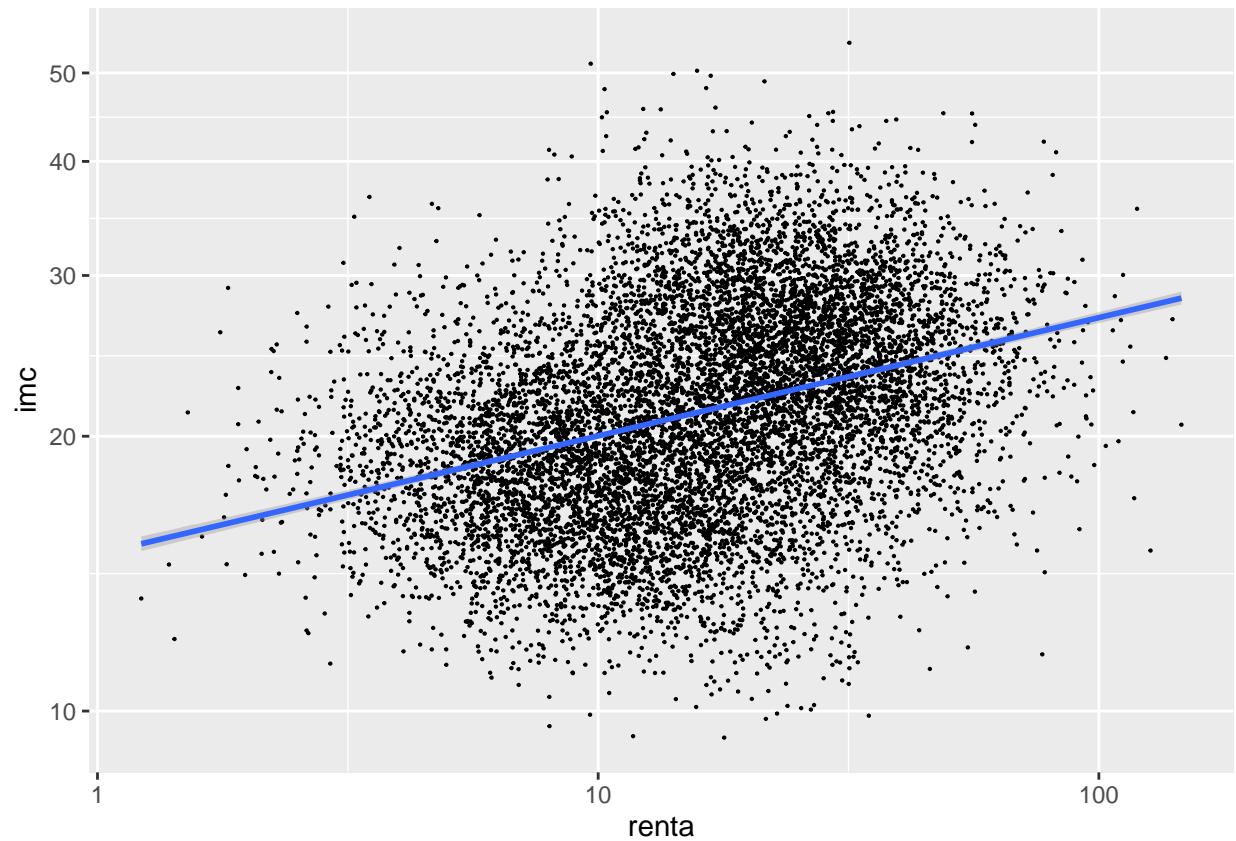


2.1.5.2. Escalas

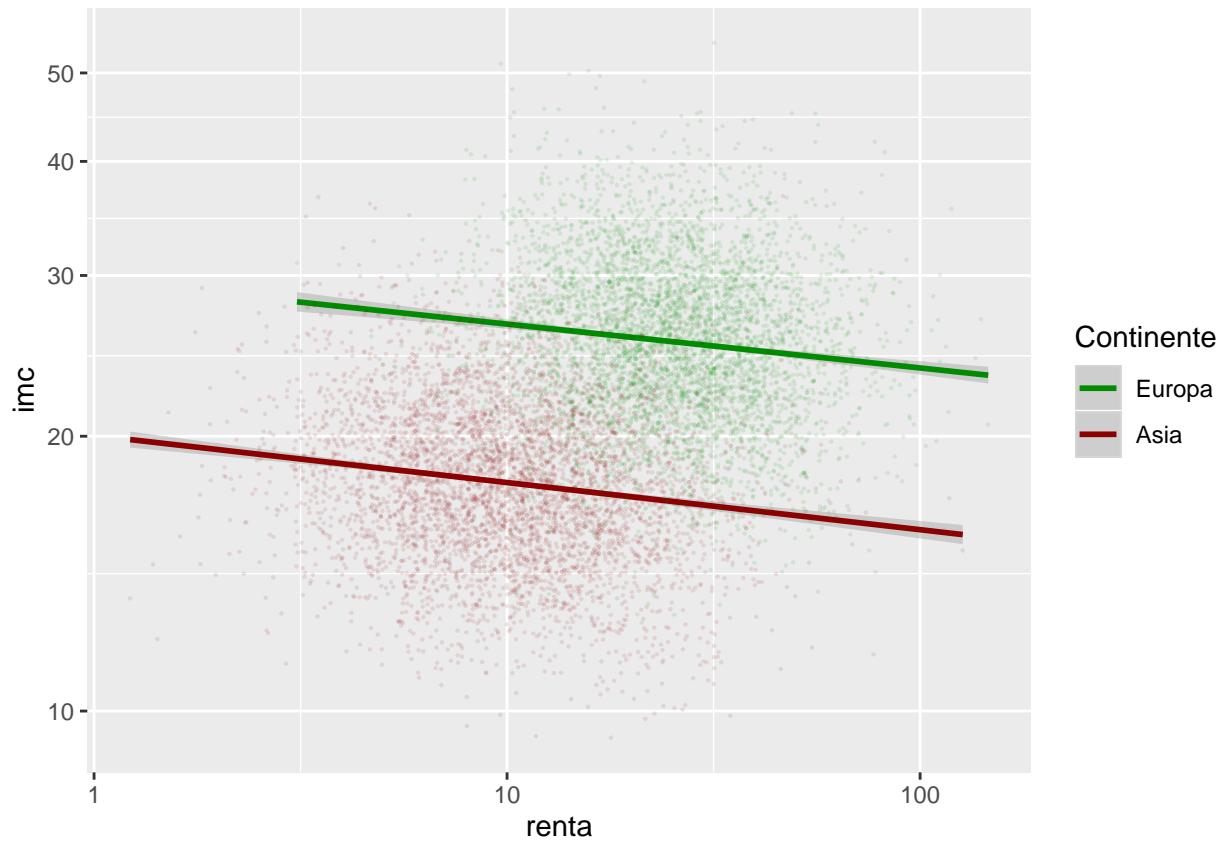
Las escalas de las estéticas pueden ser modificadas para mejorar la claridad del gráfico.

```
p<-ggplot(obesidad,aes(x=renta,y=imc))+geom_point(size=.1)+geom_smooth(method="lm")  
p
```





```
ggplot(obesidad,aes(x=renta,y=imc,color=region))+geom_smooth(method="lm") +
  geom_point(size=.1,alpha=.1)+
  scale_x_log10()+scale_y_continuous(breaks=seq(10,50,10),trans="log")+
  scale_color_manual("Continente",values=c("green4","red4"),limits=c("Europa","Asia"))
```



2.1.5.3. 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, etiquetas, colores de fondo, el tamaño de los márgenes, etc. Cambiar el tema por defecto, puede ser útil cuando los gráficos tienen que adecuarse a una imagen corporativa o atenerse a algún criterio de publicación.

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 + facet_grid(~region) + theme_bw()
p + facet_grid(~region) + theme_classic()
```

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`.

2.1.5.4. Exportación de los graficos

Una vez creado un gráfico, es posible exportarlo en diversos formatos:

- Imagen tipo bitmap (`jpeg`,`png`,`bmp`,`tiff`,...)
- Imagen vectorial (`pdf`,`svg`,...)

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(obesidad,aes(x=renta,y=imc,color=region))+geom_smooth(method="lm")
ggsave("obesidad.pdf")
#ggsave("mortalidad.pdf", width = 20, height = 20, units = "cm")
ggsave("obesidad.png")

```

Si no se especifica ruta, las imágenes serán guardadas en el directorio de trabajo.

Nota

Las imágenes vectoriales tienen una resolución “infinita” y suelen ocupar poca memoria. Sin embargo, no todos los editores de texto admiten este tipo de formato.

2.2. Mapas

- Con `ggplot2` se puede también construir representaciones gráficas con 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 usa recursos disponibles en la *web* a través de APIs (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 APIs 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 (2500 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:

```

#ens <- geocode('Calle Monforte de Lemos 5, madrid') #ya no funciona!
ens<-c(lat= 40.47767,lon=-3.691096)
### Una alternativa gratuita sin límites para direcciones en España
#require(caRtociudad)
#ens <- cartociudad_geocode('Calle Monforte de Lemos 5, madrid')
### Otra alternativa OSM
#require(tmaptools)
#ens=as.numeric(geocode_OSM('Monforte de Lemos 5, madrid')$coords)

```

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*).

```

require(caRtociudad)
mapa=get_cartociudad_map(ens,radius=2)
#ens<-c(left=-3.7,bottom=40.47,right=-3.68,top=40.485)
#mapa <- get_stamenmap(ens,zoom = 16,maptype="toner-lite")

```

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 (con la función `ggmap`):

```
ggmap(mapa)
```

O bien se puede marcar sobre él puntos de interés:

```

require(data.table)
# localización de los bares con terrazas de Madrid
# https://www.datanalytics.com/2017/03/02/todas-las-terrazas-de-madrid/
terrazas=fread("data/terrazas.csv")

ggmap(mapa) + geom_point(aes(x = lon, y = lat), data = terrazas, colour = 'red3',size = 3,alpha=.5)

```



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

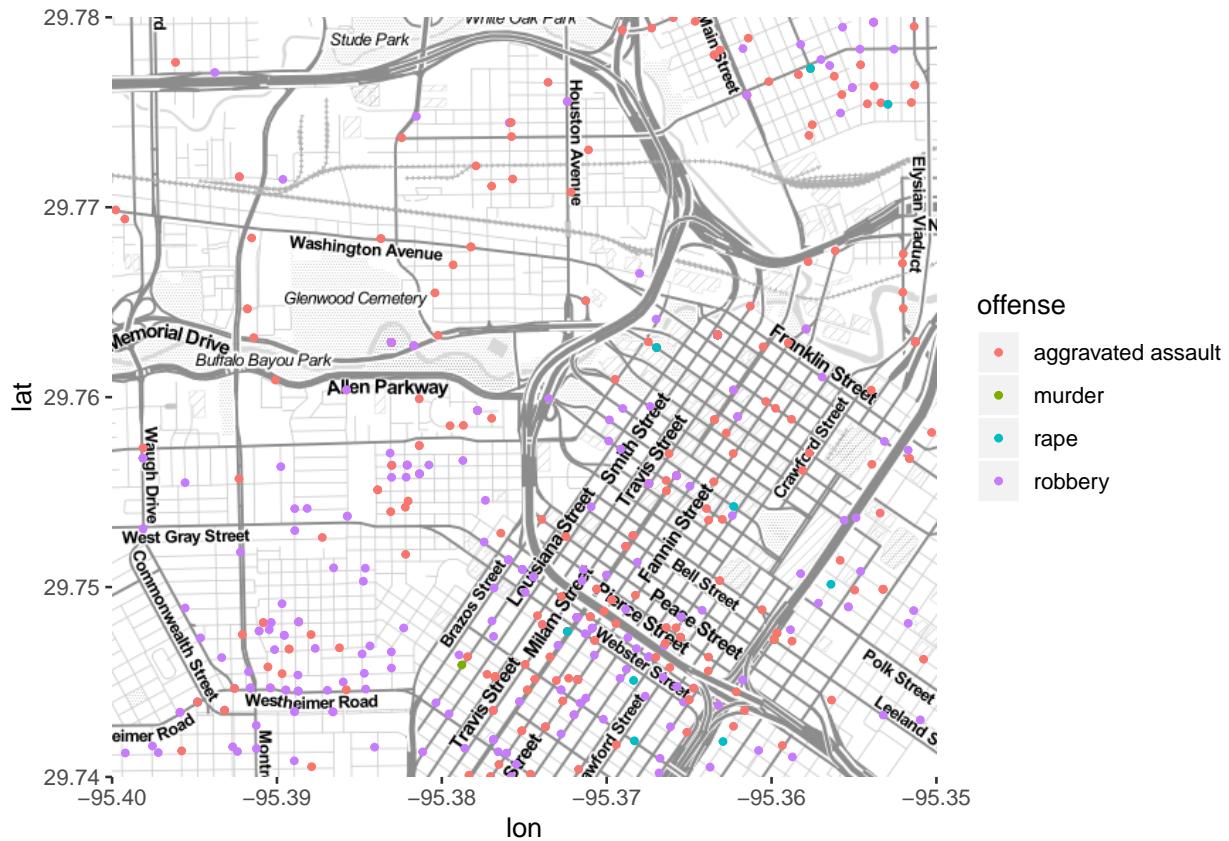
2.2.1. Más ejemplos de mapas con puntos

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,

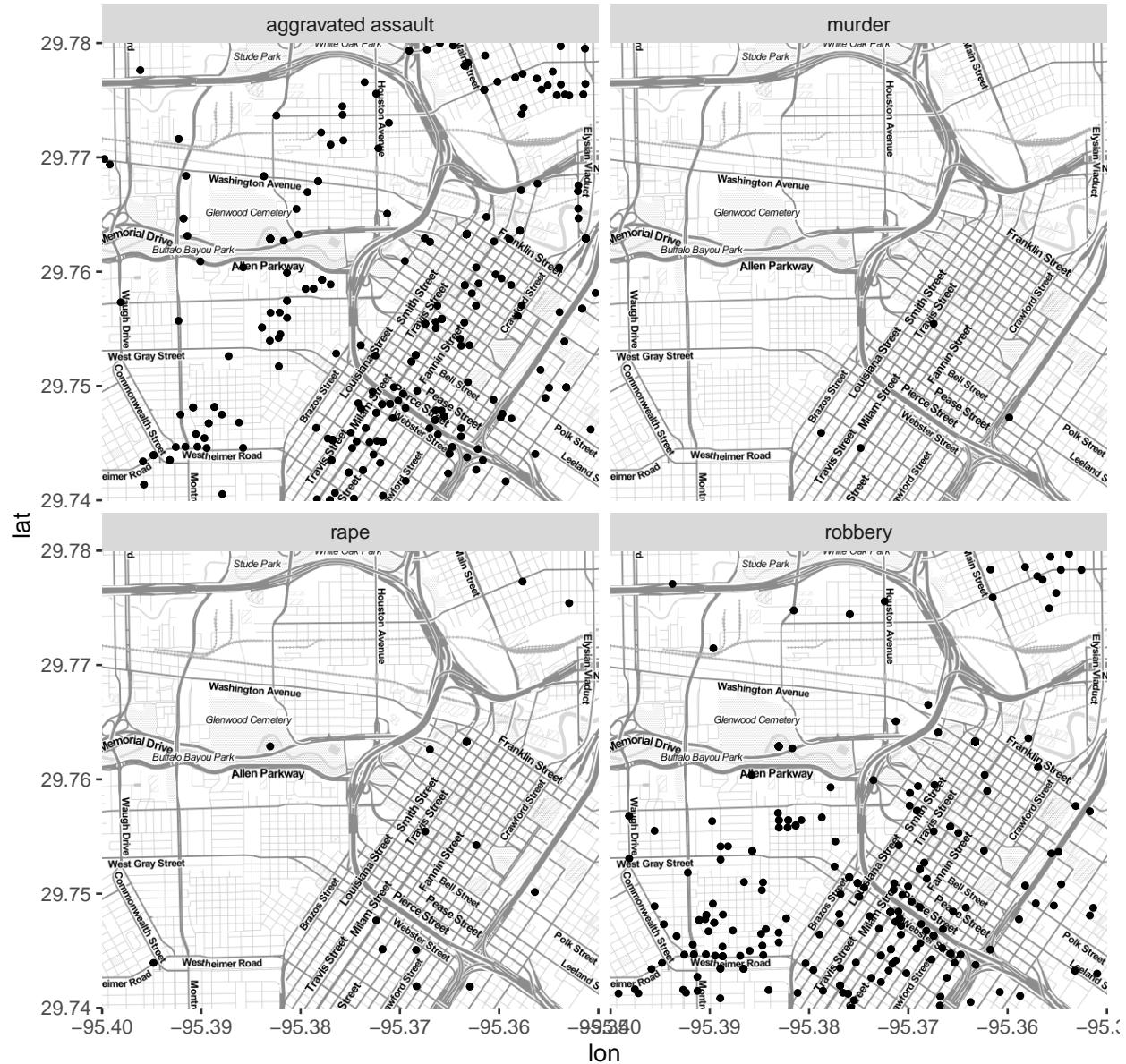
```
crimes.houston <- subset(crime, !crime$offense %in% c("auto theft", "theft", "burglary"))
```

El tipo de mapas más simples son los que se limitan a representar puntos sobre una capa cartográfica.

```
#houston<-geocode('houston',source = "dsk")
houston=c(left = -95.4, bottom = 29.74, right = -95.35, top = 29.78)
HoustonMap <- ggmap(get_stamenmap(houston,zoom=14,maptype="toner-lite"))
HoustonMap +
  geom_point(aes(x = lon, y = lat, colour = offense), data = crimes.houston, size = 1)
```



Ejercicio 2.3 Los mecanismos conocidos de `ggplot2`, como las facetas, están disponibles en `ggmap`. Descomponer el anterior gráfico utilizando `facet_wrap` por tipo de crimen (ver gráfico siguiente). Hacer lo mismo con el día de la semana.



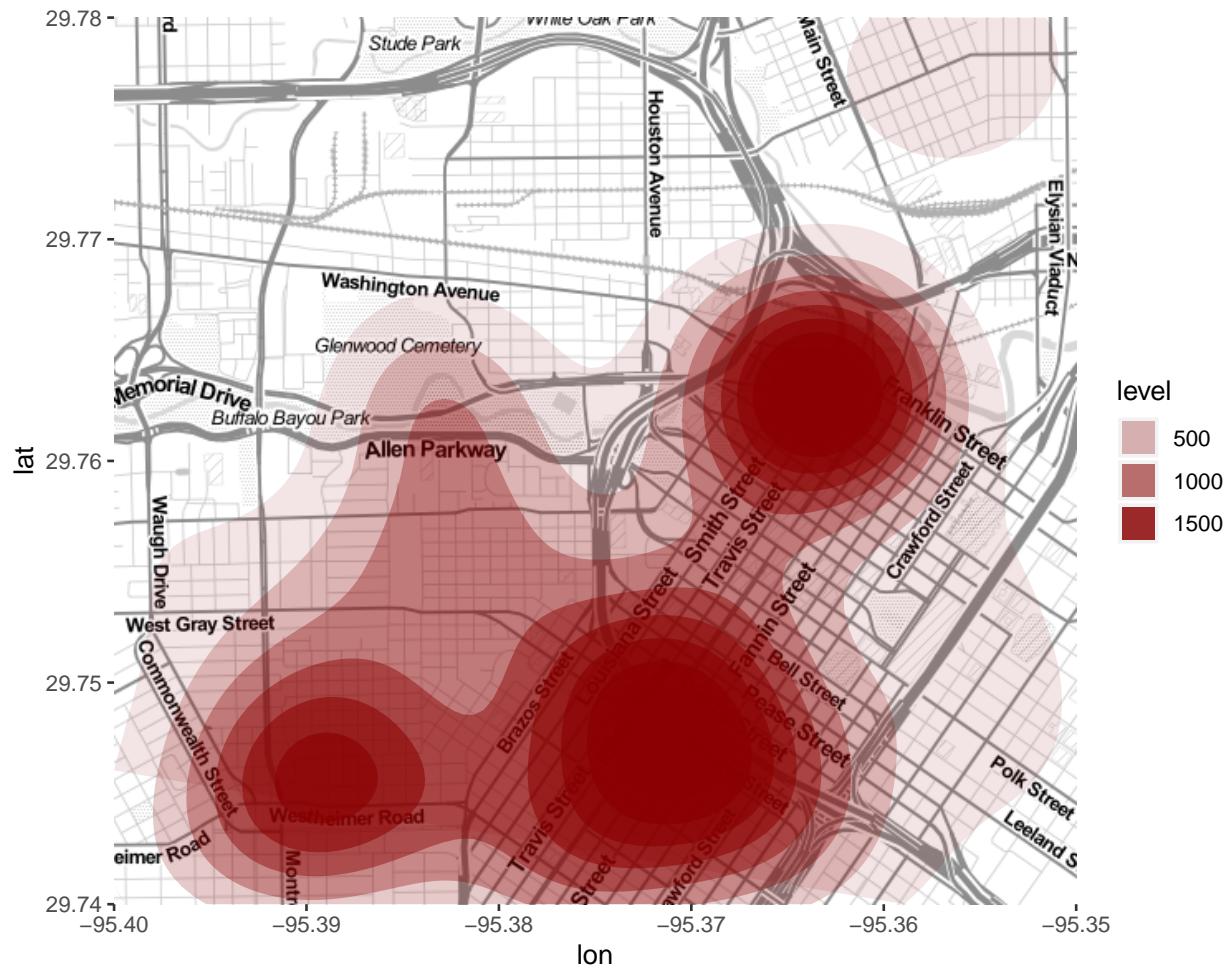
Ejercicio 2.4 Pintar las gasolineras en el mapa de España (o de una provincia o un municipio) utilizando el fichero `data/carburantes.csv`. Modificar el tamaño (o color) de los puntos en función de, por ejemplo, el precio de los carburantes.

2.2.2. Más allá de los puntos: densidades y polígonos

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

Se puede también utilizar `stat_density2d`, que representa densidades, para identificar las zonas de mayor criminalidad.

```
HoustonMap +
  stat_density2d(aes(x = lon, y = lat, alpha = ..level..), fill="red4",
                 size = 2, data = subset(crimes.houston, offense=="robbery"),
                 geom = "polygon")
```



Por otra parte, la información estadística puede ser proporcionada de manera agregada en unidades espaciales (a nivel provincial, municipal, ...), como para los datos del paro.

Lo primero que se necesita para representar estos datos, es el conjunto de polígonos (o “shape”) que definen las secciones geográficas. Esta información se puede descargar desde el servidor GADM mediante el paquete `raster`:

```

require(raster)
shape <- getData("GADM", country= "Spain", level = 2) #mapa administrativo a nivel provincial
peninsula <- subset(shape,!NAME_1=="Islas Canarias") #mapa sin las islas canarias
peninsula

## class      : SpatialPolygonsDataFrame
## features   : 50
## extent     : -9.301806, 4.328195, 35.17058, 43.79153  (xmin, xmax, ymin, ymax)
## crs        : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
## variables  : 13
## names      : GID_0, NAME_0,   GID_1,           NAME_1, NL_NAME_1,   GID_2,   NAME_2, VARNAME_2, NI
## min values  :   ESP, Spain, ESP.1_1,       Andalucía, NA, ESP.1.1_1, A Coruña, Alacant,
## max values  :   ESP, Spain, ESP.9_1, Región de Murcia, NA, ESP.9.1_1, Zaragoza, València,

```

Se puede representar este mapa mediante el comando `plot`

```

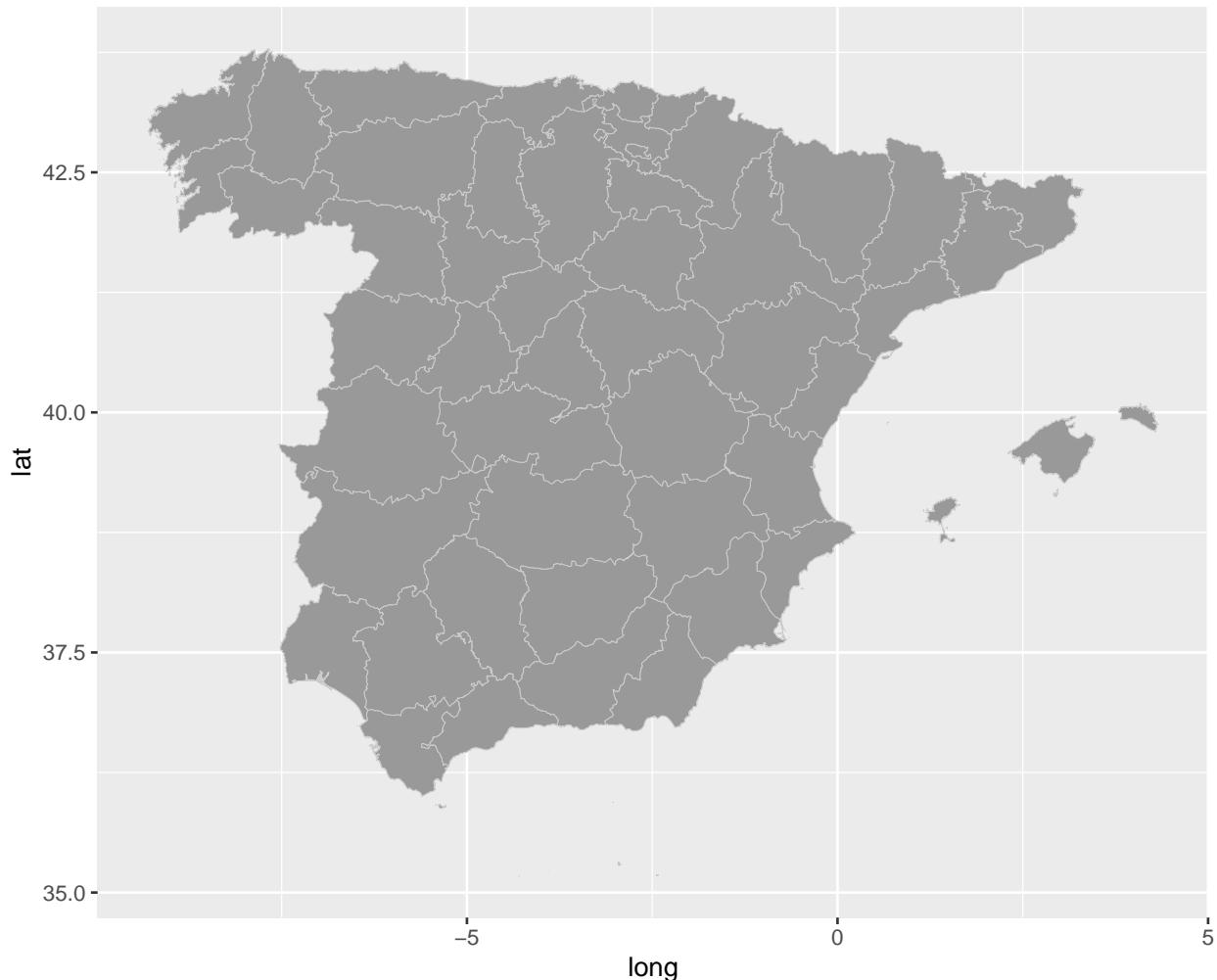
plot(peninsula,col="lightblue")
text(coordinates(peninsula), labels=peninsula$NAME_2,cex=.5)

```



o utilizando el paquete `ggplot2`:

```
#gpclibPermit()
peninsula.df=fortify(peninsula,region="CC_2") #convierte el shape en data.frames
ggplot() + geom_polygon(data = peninsula.df, aes(long, lat, group = group),
                        fill="grey60", colour = "grey80", size = .1)+coord_quickmap()
```



Ahora pintamos en el mapa los datos del paro (Mujeres, 2011, primer trimestre):

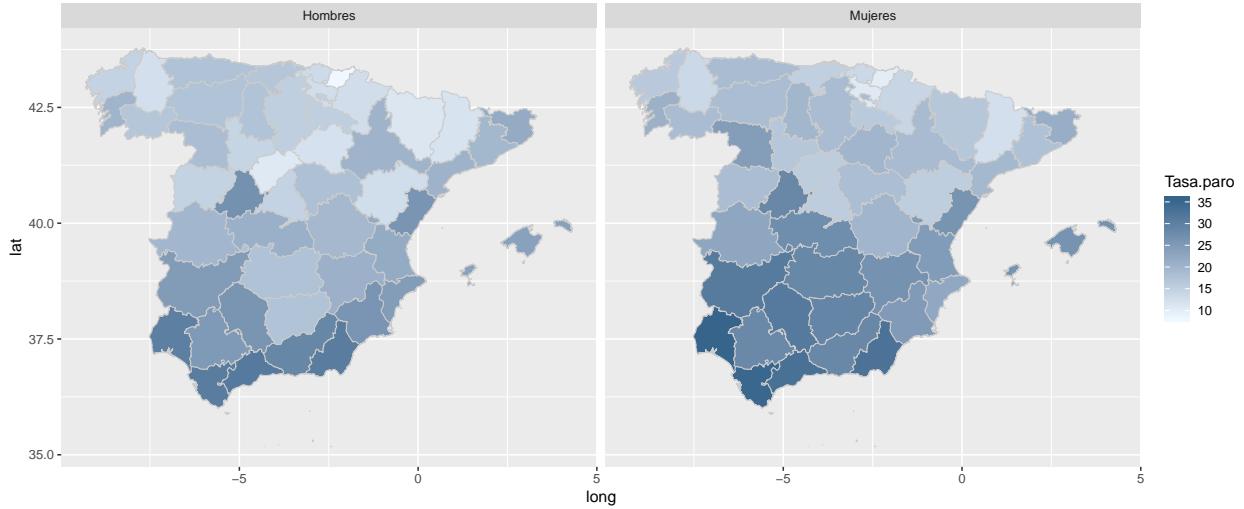
```

paro=fread("data/paro.csv",encoding="UTF-8")
paro[,id:=sub(" ","0",format(Prov.id,width=2))]
Paro <- subset(paro,Año==2011 & Trimestre=="I")

peninsula.paro=merge(peninsula.df,Paro,by="id") #juntamos las dos bases

ggplot() +
  geom_polygon(data = peninsula.paro, aes(long, lat, group = group,fill=Tasa.paro), colour = "grey80",
  facet_grid(~ Sexo) + scale_fill_gradient(low="aliceblue",high="steelblue4") + coord_quickmap()

```

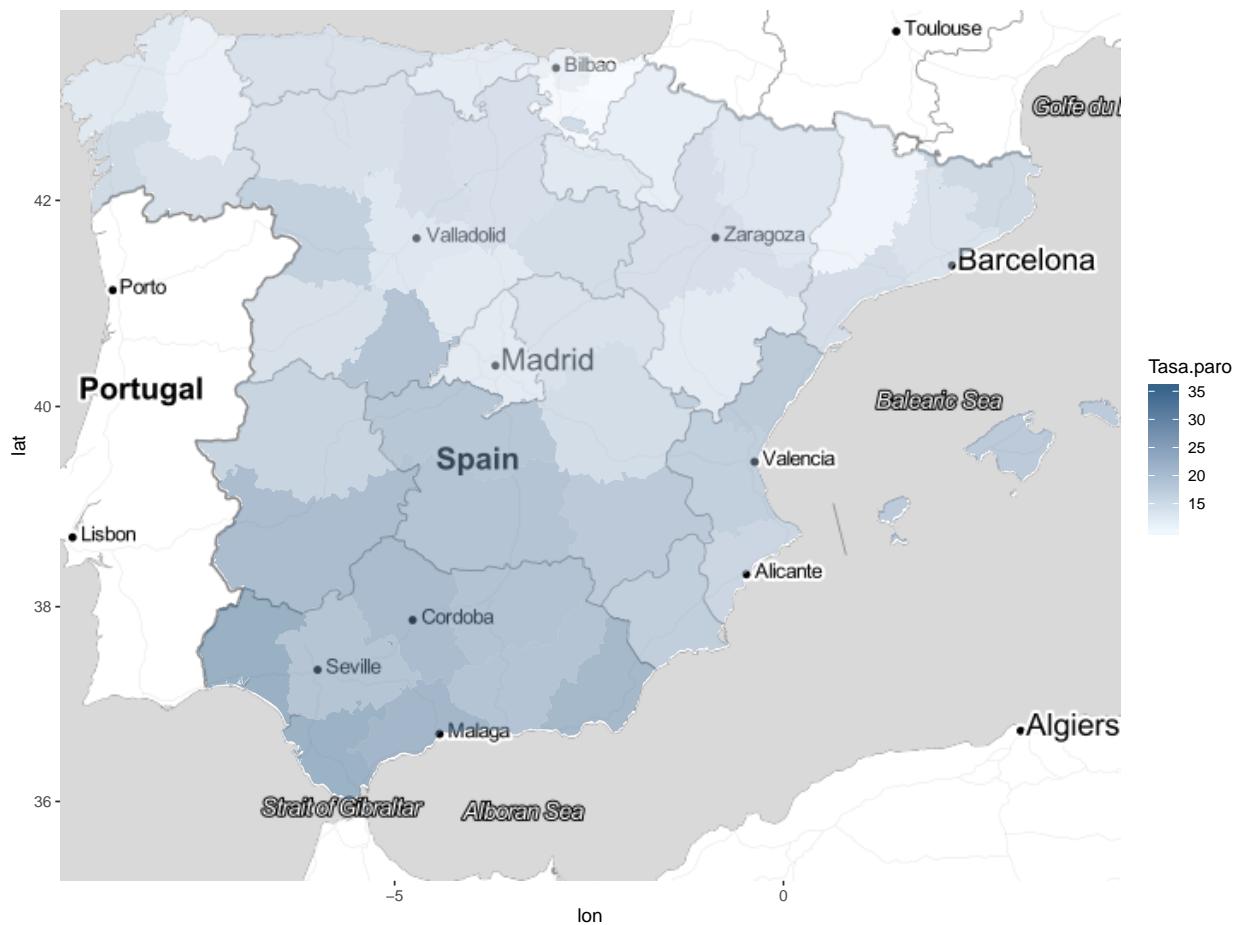


Podemos incluso dibujar este mapa, sobre un lienzo obtenido mediante `get_map`:

```
bb=bbox(peninsula)
españa <- get_stamenmap(c(bb),zoom=6,maptype="toner-lite")

mujeres=subset(peninsula.paro,Sexo=="Mujeres")

ggmap(españa) + geom_polygon(data = mujeres, aes(long, lat, group = group,fill=Tasa.paro),alpha=.5) +
  scale_fill_gradient(low="aliceblue",high="steelblue4")
```



2.3. Árboles filogenéticos

2.3.1. Importación de un árbol

- Un árbol filogenético es un esquema arborescente que muestra las relaciones evolutivas entre varias especies u otras entidades que se cree que tienen una ascendencia común (Wikipedia).
- El método de construcción de árboles filogenéticos consiste en analizar rasgos heredables (secuencias de ADN) o caracteres morfológicos con el fin de establecer relaciones de parentesco entre especies (o taxones). Casi todos los métodos filogenéticos comienzan con una matriz de distancia en la cual las diferencias entre taxones se estiman sumando discrepancias en nucleótidos o caracteres morfológicos cuantitativos. Cada nodo del árbol representa el ancestro común.
- Sin embargo, aquí no entraremos en la construcción de los arboles, sino simplemente en su representación gráfica. Suponemos pues que ya disponemos de un árbol filogenético. El paquete `treeio` sirve como interfaz para importar varios arboles en varios formatos (Newick, Nexus, NHX, jplace,).

EL siguiente comando permite instalar el paquete `ggtree` y sus dependencias (incluye el paquete `treeio`) desde el repositorio de Bioconductor.

```
if (!requireNamespace("BiocManager", quietly = TRUE))
  install.packages("BiocManager")

BiocManager::install("ggtree")
```

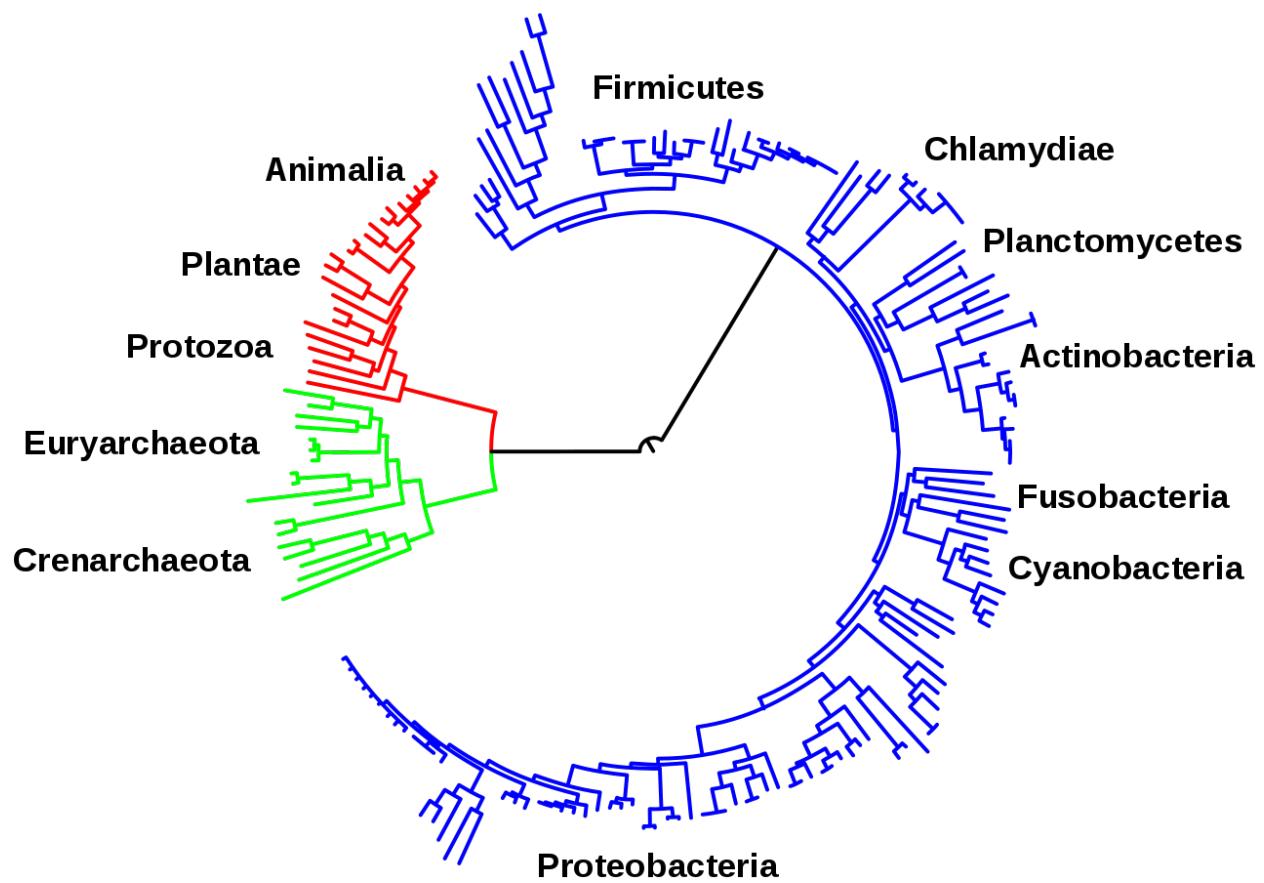


Figura 1: El árbol de la vida

A continuación importamos un árbol sobre filogenia de primates:

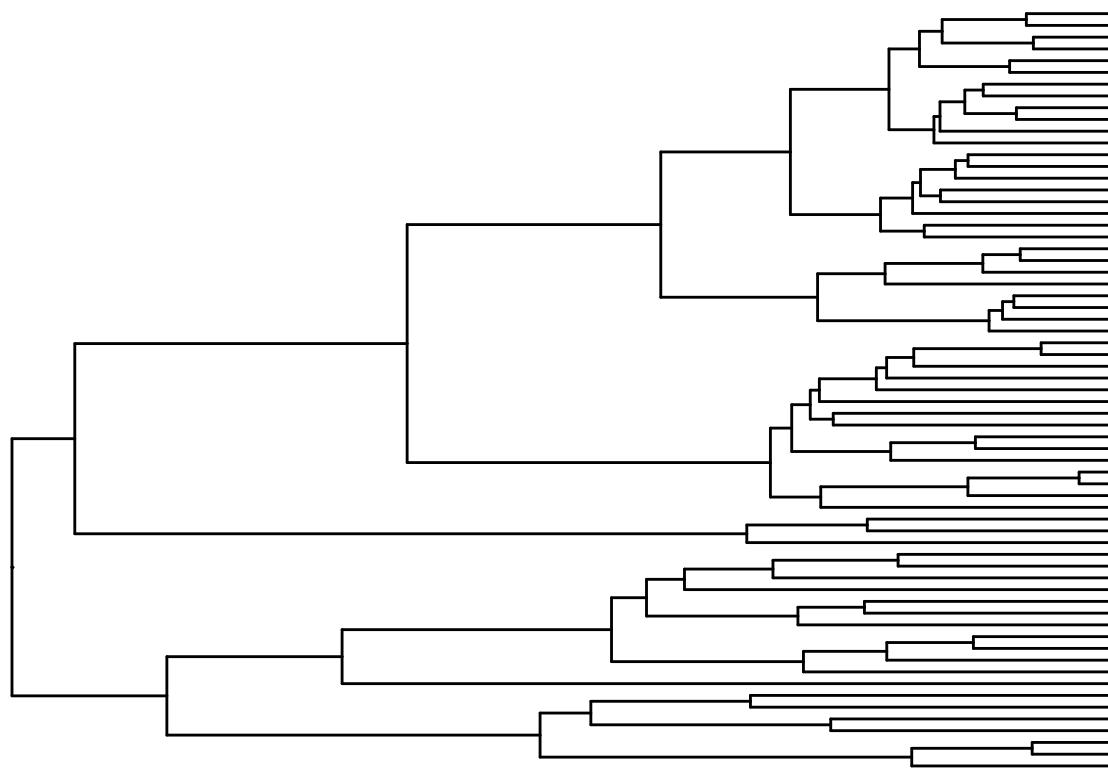
```
require(ggplot2)
require(ggtree)
url="https://raw.githubusercontent.com/rgriff23/Dissertation/master/Chapter_2/data/tree.nex"
arbol <- read.nexus(url)
summary(arbol)

##
## Phylogenetic tree: arbol
##
##   Number of tips: 65
##   Number of nodes: 64
##   Branch lengths:
##     mean: 10.01834
##     variance: 67.22004
##   distribution summary:
##     Min.   1st Qu.    Median   3rd Qu.    Max.
## 0.401789  4.710731  8.266731 14.271122 51.124065
##   No root edge.
##   First ten tip labels: Allenopithecus_nigroviridis
##                           Cercopithecus_mitis
##                           Cercopithecus_petaurista
##                           Chlorocebus_sabaeus
##                           Erythrocebus_patas
##                           Miopithecus_ogouensis
##                           Avahi_laniger
##                           Cheirogaleus_major
##                           Daubentonnia_madagascarensis
##                           Eulemur_fulvus
##   No node labels.
```

2.3.2. Representación básica

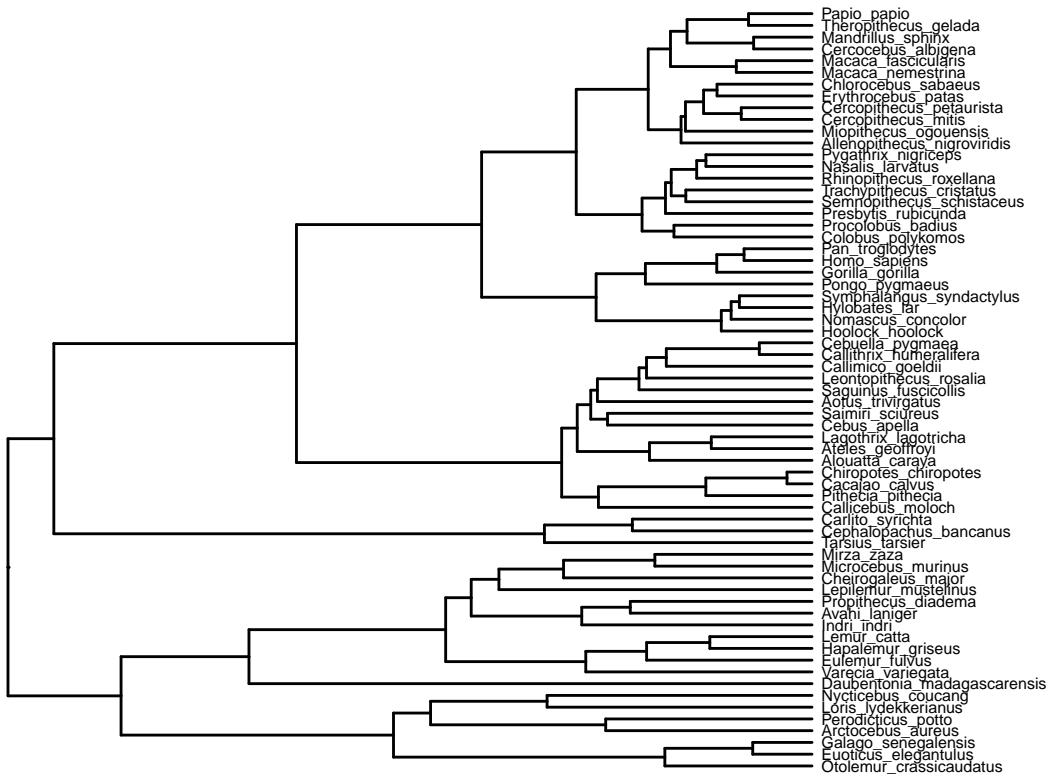
Vamos ahora a Explorar la estructura del árbol mediante el comando `ggtree`

```
ggtree(arbol)
```



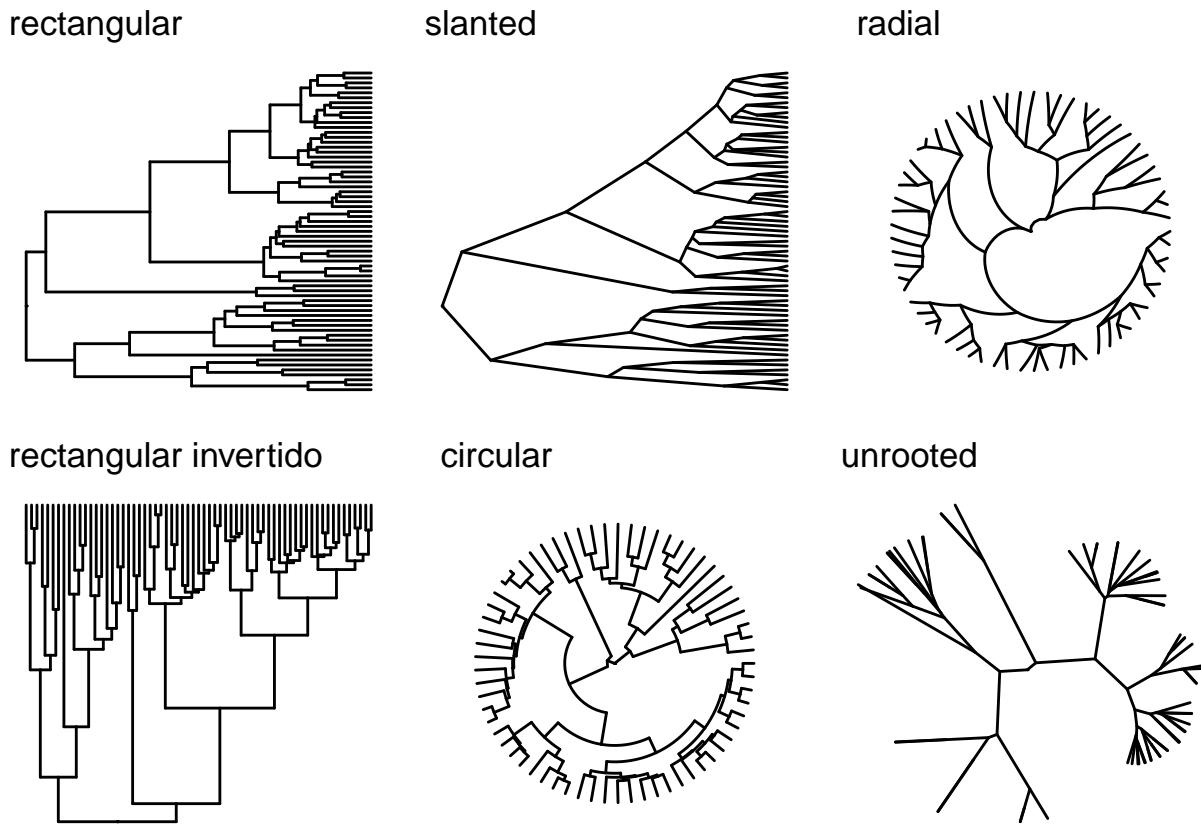
Para poder añadir las etiquetas de los nodos finales utilizamos el comando `geom_tiplab`:

```
ggtree(arbol)+geom_tiplab(size=2,offset=.5)+xlim(0,100)
```



Además de la representación rectangular, otros sistemas de coordenadas (circular, radial, ...) son disponibles utilizando la opción `layout`:

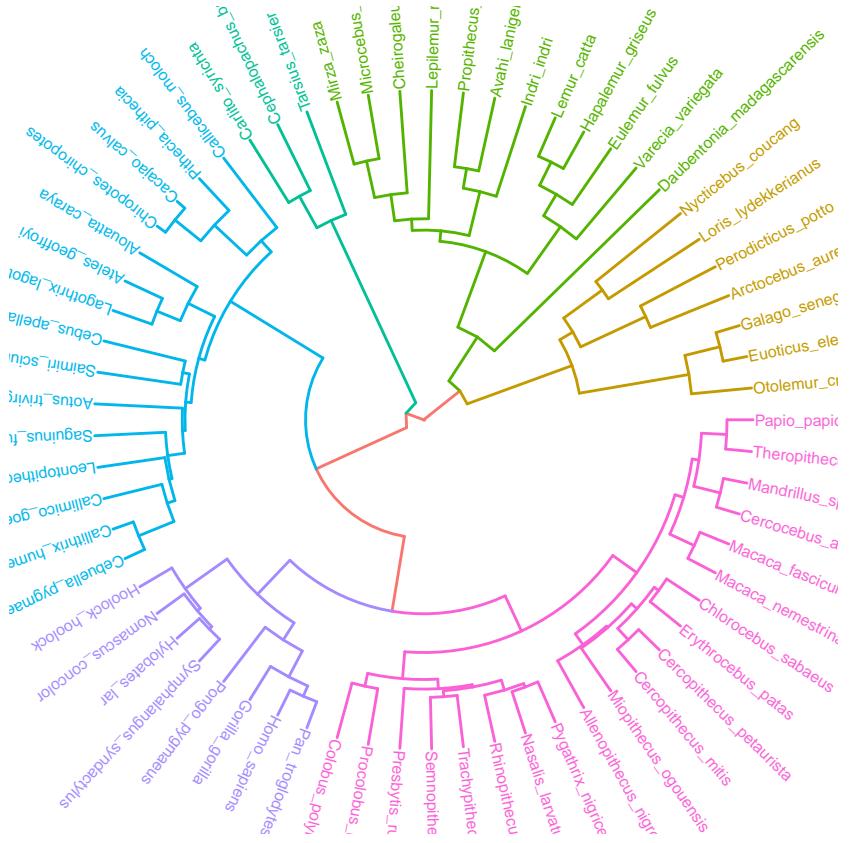
```
require(ggplot2)
pa <- ggtree(arbol) + ggtitle("rectangular")
pa.bis <- ggtree(arbol) + coord_flip() + ggtitle("rectangular invertido")
pb <- ggtree(arbol, layout="slanted") + ggtitle("slanted")
pc <- ggtree(arbol, layout="circular") + ggtitle("circular")
pd <- ggtree(arbol, layout="radial") + ggtitle("radial")
pe <- ggtree(arbol, layout="unrooted") + ggtitle("unrooted")
multiplot(pa, pa.bis, pb, pc, pd, pe, ncol=3)
```



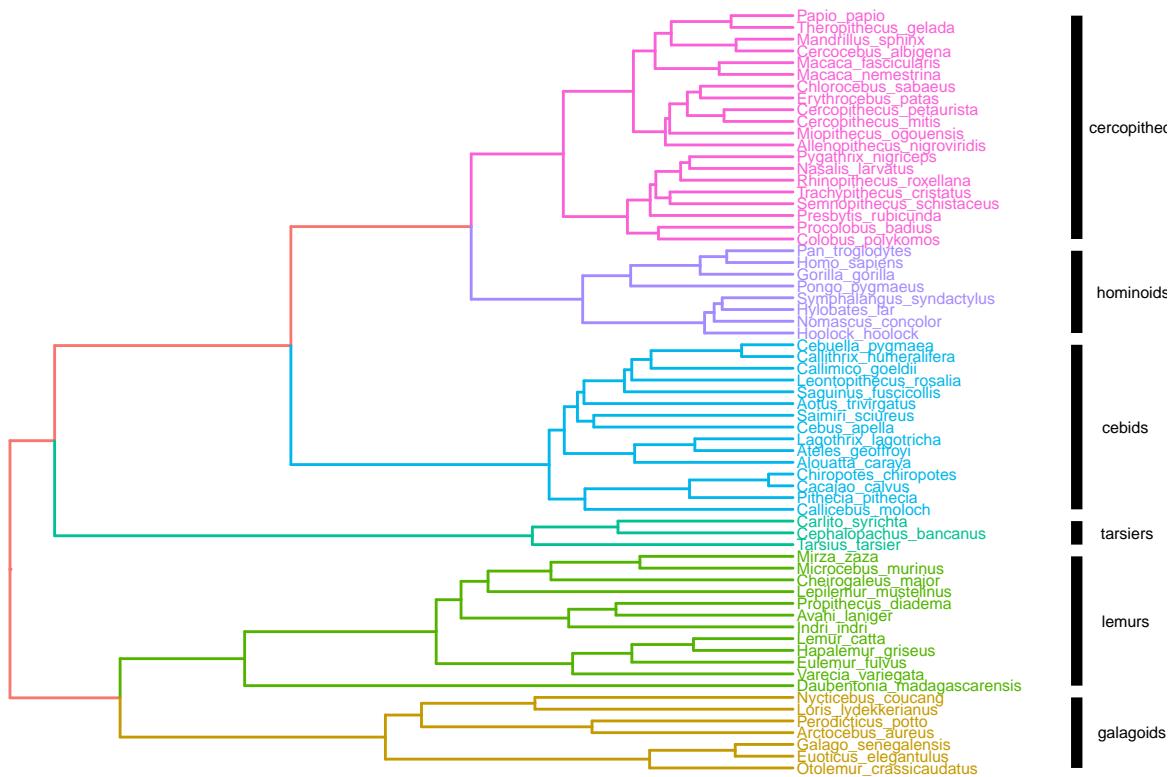
2.3.3. Agrupación de nodos

La función `groupClade` permite agrupar nodos que comparten un mismo ancestro. A continuación agrupamos los nodos por familia de primates: - Galagoidea (124) - Lemuroidea (113) - Tarsioidea (110) - Ceboidea (96) - Hominoidea (89) - Cercopithecoidea (70)

```
nodos=c(124, 113, 110, 96, 89, 70)
arbol<-groupClade(arbol,.node=nodos) #agrupación por especie
familias=c("galagooids","lemurs","tarsiers","cebids","hominoids","cercopithecoids")
ggtree(arbol, aes(color=group), layout='circular') +
  geom_tiplab(size=2, aes(angle=angle))
```



```
code="ggtree(arbol, aes(color=group)) + geom_tiplab(size=2)"
for(i in 1:6){
  mas=paste0("geom_cladelabel(",nodos[i],",\"",familias[i],"\",offset=25, barsize=2,offset.text=2.5, font",
  code=paste(code,mas,sep="+")
}
eval(parse(text=code))
```



```
##### Explorar la estructura de los siguientes arboles  
arbol<-read.tree("http://www.phytools.org/eqp2015/data/anole.tre") #filogenia de lagartijas  
arbol <- read.nexus("https://raw.githubusercontent.com/rgriff23/Dissertation/master/Chapter_2/data/tree
```

2.3.4. Añadir Heatmap

Simulamos cinco caracteres continuos (ejemplo: tamaño del cuerpo, ...) de acuerdo al árbol filogenético:

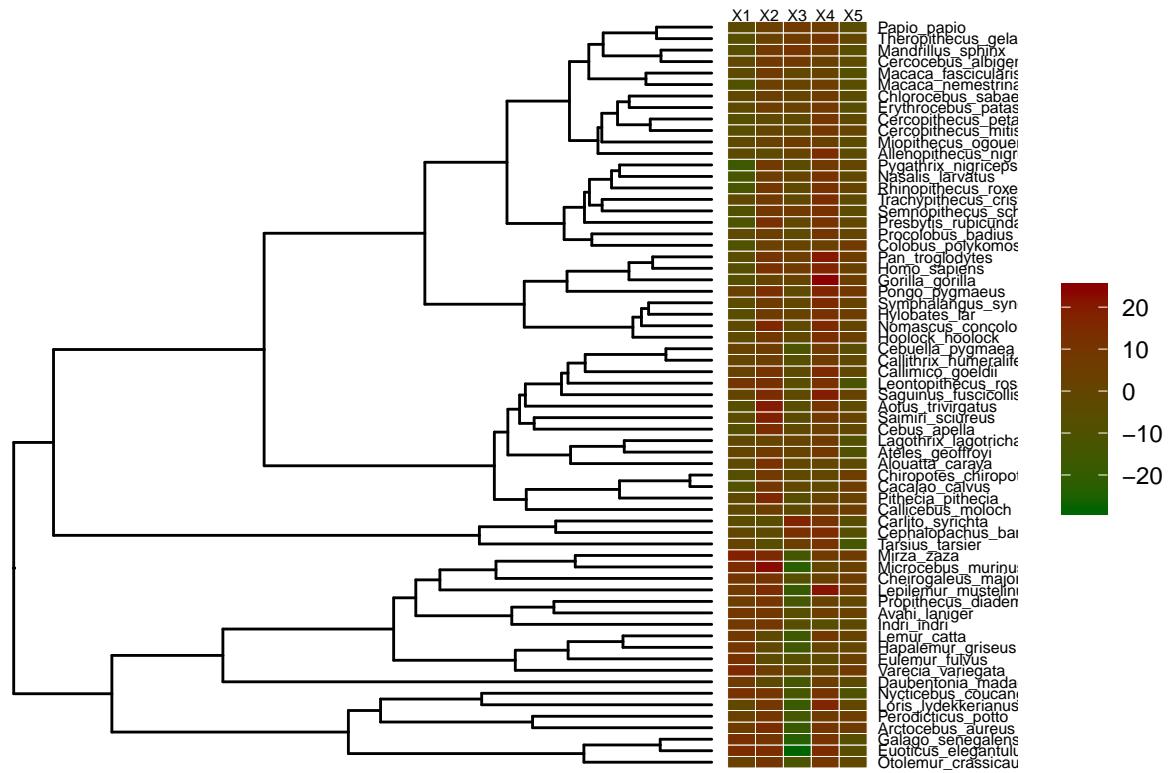
```
require(phytools)
set.seed(1234) # simulacion reproducible
traits <- data.frame(fastBM(arbol, nsim=5))
```

utilizando la función `gheatmap` para adjuntar un `heatmap` a la representación del árbol:

```
p <- ggtree(arbol) + xlim(0, 100) + geom_tiplab(size=2, offset=17)
```

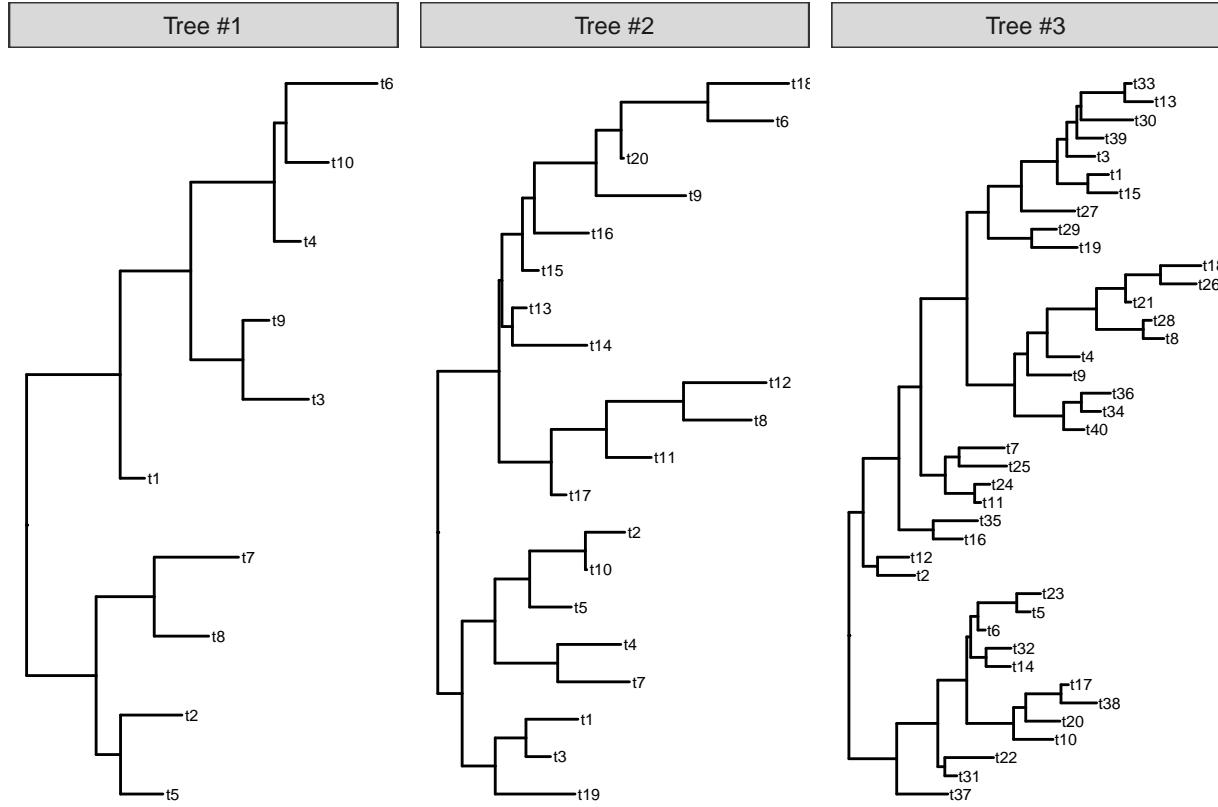
```
# añade el heatmap
```

```
gheatmap(p, traits, offset=0.2, width=0.2, low="darkgreen", high="darkred", colnames_position = "top",
```



2.3.5. Visualización de múltiples árboles

```
trees <- lapply(c(10, 20, 40), rtree)
class(trees) <- "multiPhylo"
ggtree(trees) + facet_wrap(~.id, scale="free") + geom_tiplab(size=2)
```



3. Rmarkdown y shiny

Esta sección está dedicada a dos extensiones de R: **shiny** y **rmarkdown**. El primero de ellos permite crear cuadros de mando interactivos. El segundo, documentos automatizados que combinan texto con código, tablas y gráficos generados directamente por R.

Estos dos paquetes no son complejos sino, más bien, extensos y llenos de detalles. El objetivo de esta sesión es recorrer sus posibilidades.

3.1. Rmarkdown

Rmarkdown permite generar documentos dinámicos al mezclar texto formateado y resultados generados por R. Los documentos generados pueden estar en HTML, PDF, Word y muchos otros formatos.

Las ventajas de esta herramienta son numerosas:

- El código y sus resultados no están separados de los comentarios asociados a ellos
- El documento final es reproducible
- El documento se puede actualizar fácilmente, por ejemplo, si los datos de origen se han modificado.

Por lo tanto, es una herramienta muy práctica para exportar, comunicar y difundir resultados estadísticos.

Este documento se ha generado a partir de archivos R Markdown [^] [Más precisamente gracias a la extensión **bookdown** que permite generar documentos de tipo libro].

Aprender Rmarkdown implica aprender dos cosas distintas:

- Markdown, un *formato* para escribir documentos simples en modo texto. Tiene la ventaja de ser fácilmente legible por humanos pero, a la vez, procesable programáticamente para volcarlos en otros

- formatos: pdf, html, ...
- La integración entre R y markdown

Aquí, un documento de R Markdown básico:

```
---
```

```
title: "Ejemplo de R Markdown"
```

```
output: pdf_document
```

```
---
```

R Markdown permite mezclar :

- texto libre puesto en formato
- bloques de código de R

Los bloques de código se pueden ejecutar para incluir sus resultados en el documento, así por ejemplo :

```
```{r}
```

```
mean(airquality$Ozone,na.rm=TRUE)
```

```
```
```

Gráficos

Se puede también incluir __gráficos__ :

```
```{r}
```

```
plot(Ozone~Temp,data=airquality,bg="lightblue",pch=21)
```

```
```
```

Al “compilar” el documento, el texto se formatea, los bloques de código se ejecutan, sus resultados se agregan al documento y todo se transforma en uno de los diferentes formatos posibles (html, pdf, word, ...).

Aquí, la representación del documento anterior en formato HTML

Ejemplo de R Markdown

R Markdown permite mezclar :

- texto libre puesto en formato
- bloques de código de R

Los bloques de código se pueden ejecutar para incluir sus resultados en el documento, así por ejemplo :

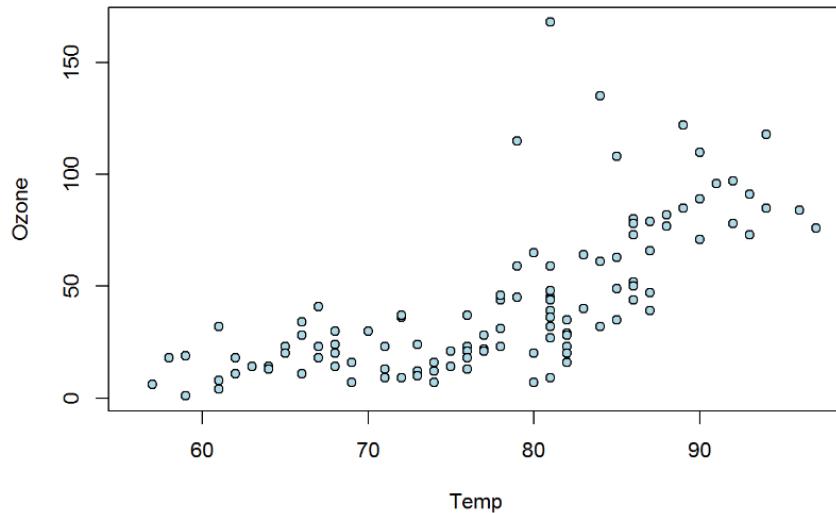
```
mean(airquality$Ozone, na.rm=TRUE)
```

```
## [1] 42.12931
```

Gráficos

Se puede también incluir gráficos :

```
plot(Ozone~Temp, data=airquality, bg="lightblue", pch=21)
```



3.1.1. Rmarkdown en 15 mn

Para aprender Markdown, se recomiendan los dos siguientes ejercicios :

Ejercicio 3.1 Crear un fichero .Rmd usando File > New File > R Markdown.

Al crear un nuevo fichero de tipo R Markdown, RStudio proporciona, en lugar de uno vacío, una plantilla que muestra algunas de las opciones disponibles en este formato. Eso facilita el siguiente ejercicio:

Ejercicio 3.2 Modificar el fichero de ejemplo creado en el ejercicio anterior añadiéndole títulos de varios niveles, párrafos de texto, cursivas, negritas, enlaces, listas (numeradas y sin numerar), etc. usando como guía el Cheat Sheet del paquete. *Compilar* el documento (p.e., pulsando el botón con la etiqueta *Knit HTML* situado encima del panel de edición de RStudio) para inspeccionar el resultado final.

Se puede también generar documentos en formato Word y PDF. Para estos formatos, es necesario tener instalados los programas : MS Word, LibreOffice o similar para el primero y LaTeX para el segundo.

El segundo de los componentes de Rmarkdown (y lo que lo diferencia de Markdown *a secas*) es la posibilidad de incorporar bloques de código en el hilo del documento. Estos bloques de código se procesan durante la compilación del documento y los resultados que generan (tablas, gráficos, etc.) se integran en la salida. La plantilla de fichero Rmarkdown que genera RStudio incluye unos cuantos ejemplos de bloques de código.

Ejercicio 3.3 Insertar sobre el documento (o sobre una nueva plantilla) bloques de código de R que hagan alguna cosa. Los bloques de código incluyen opciones en su encabezamiento (p.e., para que un bloque se ejecute o no; para que el código se muestre o se oculte en el documento final, etc.). Las opciones disponibles se pueden consultar en el *Cheat Sheet* del paquete.

3.1.2. Elementos de un documento Rmarkdown

3.1.2.1. Encabezado (préambulo)

La primera parte del documento es su *encabezado*. Se encuentra al principio del documento y está delimitado por tres guiones (---) antes y después:

```
---
```

```
title: "Titulo"
author: "Nombre Apellido"
date: "2 de mayo de 2018"
output: html_document
---
```

Este encabezado contiene los metadatos del documento, como su título, autor, fecha, más una serie de opciones posibles que permiten configurar o personalizar todo el documento y su representación. Aquí, por ejemplo, la línea `output: html_document` indica que el documento generado tendrá un formato HTML.

3.1.2.2. Texto del documento

El cuerpo del documento consiste en texto con la sintaxis de Markdown: un marcado ligero que permite establecer niveles de títulos o formatear texto. Por ejemplo, el siguiente texto:

```
Este es un texto *en cursiva* y **en negrita**.
```

Se puede definir una lista así:

- primer elemento
- segundo elemento

Que dará la siguiente salida

Este es un texto *en cursiva* y **en negrita**.

Se puede definir una lista así:

- primer elemento
- segundo elemento

Los títulos de diferentes niveles se pueden definir comenzando una línea con uno o más caracteres #:

```
# Titulo de nivel 1
```

```
## Titulo de nivel 2
```

```
### Titulo de nivel 3
```

Cuando se han definido los títulos, al hacer clic en el ícono *Show document outline* en el extremo derecho de la barra de herramientas asociada al archivo, se muestra una tabla dinámica de contenidos generada automáticamente a partir de los títulos que permite navegar fácilmente en el documento.

La sintaxis de Markdown permite también insertar enlaces o imágenes. Por ejemplo, la siguiente sintaxis:

[ISCIII] (<http://www.isciii.es/>)

Dará el siguiente vínculo:

ISCIII

En RStudio, el menú *Help* y luego *Markdown quick reference* proporciona una descripción más completa de la sintaxis.

3.1.2.3. Bloques de código

Además del texto libre en formato Markdown, un documento R Markdown contiene, como su nombre indica, código R. Este código se incluye en fragmentos definidos por la siguiente sintaxis:

Como esta cadena de caracteres no es muy fácil de escribir, se puede usar *R* en el menú Insertar de RStudio, o teclear el atajo **Ctrl+Alt+i**.

Se puede dar un nombre al bloque y se indica directamente después de *r*:

```
{r nombre_del_bloque}
```

No es obligatorio, pero puede ser útil en caso de error de compilación, para identificar el bloque que causó el problema. Atención, no podemos tener dos bloques con el mismo nombre.

Además de un nombre, se puede pasar a un bloque una serie de opciones para modificar su comportamiento.

```
```{r echo = FALSE, warning = FALSE}
x <- 1:5
````
```

Una de las opciones más útiles es la opción *echo*. Por defecto *echo=TRUE*, y el bloque de código R se inserta en el documento generado:

```
x <- 1:5
print(x)
```

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

Pero, si la opción *echo=FALSE*, entonces el código R ya no se inserta en el documento, y solo se el resultado será visible:

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

Aquí hay una lista de algunas de las opciones más comunes:

| opción | valores | descripción |
|---------|--------------|-------------------------------------------------------|
| echo | TRUE / FALSE | Mostrar o no el código R en el documento |
| eval | TRUE / FALSE | Ejecutar o no el código R en tiempo de compilación |
| warning | TRUE / FALSE | Mostrar o no las advertencias generadas por el bloque |
| message | TRUE / FALSE | Mostrar o no los mensajes generados por el bloque |

Hay muchas otras opciones descritas en la Guía de referencia de R Markdown.

3.1.3. Tablas con Rmarkdown

3.1.3.1. Tablas cruzadas

Por defecto, las tablas generadas por la función *table* se muestran tal y como aparecen en la consola de R, es decir, en texto sin formato:

Cuadro 2: Supervivencia a la catástrofe del Titanic según la clase

| | No | Yes |
|------|-----|-----|
| 1st | 122 | 203 |
| 2nd | 167 | 118 |
| 3rd | 528 | 178 |
| Crew | 673 | 212 |

```
# titanic<-ftable(Survived~Class,data=Titanic)
# Supervivencia al Titanic según clase
titanic<-apply(Titanic,c(1,4),sum)
titanic
```

```
##           Survived
## Class     No Yes
##   1st    122 203
##   2nd    167 118
##   3rd    528 178
##   Crew   673 212
```

Su presentación se puede mejorar utilizando la función `kable` de la extensión `knitr`:

```
library(knitr)
kable(titanic,caption="Supervivencia a la catástrofe del Titanic según la clase")
```

3.1.3.2. Base de datos

Respecto a las bases de datos (`tibble` o `data.frame`), la presentación HTML por defecto es el contenido que aparece en consola. Este formato puede ser poco adecuado si la tabla excede una cierta dimensión.

Una alternativa es usar la función `paged_table`, que muestra una representación HTML paginada de la base:

```
require(gapminder)
rmarkdown::paged_table(gapminder)
```

Otra alternativa es la función `datatable` de la extensión `DT`, que ofrece aún más interactividad:

```
library(DT)
datatable(gapminder)
```

En cualquier caso, no es recomendable mostrar una tabla de datos muy grandes de esta manera porque el archivo HTML resultante contendría todos los datos y, por lo tanto, sería muy grande.

3.2. Shiny

`shiny` es un paquete de R para la construcción de cuadros de mando *web* interactivos. Permite, por ejemplo, crear interfaces para algoritmos o acceder y manipular tablas de datos a través de controles de HTML: *sliders*, botones, etc.

El paquete proporciona varias aplicaciones de ejemplo que usaremos para aprender los rudimentos de `shiny`. Por ejemplo, se puede hacer

```
library(shiny)
runExample(example = "01_hello")
```

para desplegar la aplicación de ejemplo `01_hello`. Esta aplicación pinta en el panel central un histograma y tiene en el lateral un *slider* con el que modular su granularidad (técnicamente, para definir el número de pedazos, *breaks*, en los que partir el rango de valores del vector subyacente).

Para detener la aplicación, en RStudio, presiona sobre el icono de la señal de *stop* (en la parte superior de la ventana de la consola); en una terminal, usa Control-C para interrumpir la ejecución.

Ejercicio 3.4 Ejecuta `runExample()` (sin argumento); el mensaje de error indica qué otros ejemplos además de `01_hello` están disponibles por defecto. Échales un vistazo a algunos.

Ejercicio 3.5 Crea tu primera aplicación en `shiny`. Para ello, despliega `01_hello`. Luego, copia los ficheros `ui.r` y `server.r` en un nuevo directorio vacío. LLámalo, por ejemplo, `prueba00`. Luego ejecuta `runApp("prueba00")` para desplegarla. (Nota: el argumento de `runApp` tiene que ser la ruta, sea absoluta o relativa, del directorio en cuestión; recuerda que una aplicación en `shiny` recibe el nombre del directorio que la contiene).

El ejercicio anterior muestra cómo construir aplicaciones en `shiny`. Una aplicación en `shiny` es un directorio que da nombre a la aplicación. Dentro de él tiene que haber, como mínimo, dos ficheros: `ui.r` y `server.r`. El primero define la interfaz de la aplicación. El segundo realiza los cálculos en segundo plano cada vez que el usuario manipula los controles de la interfaz.

Además de estos ficheros, en aplicaciones más complejas, puede haber otros organizados o no en subdirectorios: datos, otros ficheros auxiliares de código, logos, CSSs, imágenes estáticas, etc. Tendrás que leer la documentación de `shiny` para averiguar cómo y dónde colocar estos recursos adicionales.

`ui.r` y `server.r` se comunican entre sí: `ui.r` tiene que pasarle parámetros a `server.r` y este resultados a aquél. Esto se hace a través de variables y estructuras de datos con una forma muy particular. El siguiente ejercicio está pensado para que descubras el mecanismo de comunicación. Se te va a pedir que traduzcas el nombre de las variables y los parámetros al español. Obviamente, al traducir las variables en uno de los ficheros se romperá la aplicación. Realinear los nombres en el otro fichero te servirá para identificar los mecanismos de comunicación.

Ejercicio 3.6 Crea `prueba01` como una copia de `prueba00`. Entonces, traduce al español el nombre de todas las variables implicadas en la aplicación.

El siguiente ejercicio te enseñará a modificar la interfaz de una aplicación en `shiny`, incorporar nuevos controles y añadir el código subyacente para que responda adecuadamente.

Ejercicio 3.7 Añade a la aplicación `prueba01` otro *slider* que mueva una linea vertical roja que dibujes sobre el histograma. Es recomendable que comiences, y en este orden, añadiendo una línea roja en algún punto (del eje x) prefijado con la función `abline`, incorporando el *slider* y finalmente, vinculando el valor proporcionado por el *slider* al punto del eje x.

Nota: cuando modifiques `ui.r`, presta atención a la estructura del programa y cómo se corresponde a la de la interfaz web: qué es lo que va en la barra lateral, qué en el panel central, etc. Ten cuidado además con los paréntesis: ¡hay muchos y es fácil desemparejarlos!

Ejercicio 3.8 Inspecciona el tutorial de `shiny` (<http://shiny.rstudio.com/tutorial/>) para descubrir qué tipo *widgets* (además de *sliders*) existen, cómo se procesan esos *inputs* en `server.r`, etc. Recuerda que ese tutorial es la principal fuente de información sobre todo lo relacionado con `shiny`.

Ejercicio 3.9 Visita la galería de aplicaciones de `shiny` (<http://shiny.rstudio.com/gallery/>) para investigar cómo implementar controles, qué tipos adicionales de paneles existen, etc.

Uno de los asuntos (avanzados) que se discuten en esas páginas es el de las reacciones: `shiny` está basado en un tipo de programación denominada *reactiva*, que es la que permite que las funciones de `server.r` simulen estar escuchando (y *reaccionen*) a los cambios que realiza el usuario en los controles de la aplicación. Puedes buscar en Internet más información sobre la *programación reactiva* si te interesa el tema.

4. Referencias

Este curso está basado en las siguientes referencias:

- *R para profesionales de los datos*, Carlos Bellosta, 2017, https://datanalytics.com/libro_r
- El libro R for data science disponible en línea, que contiene un capítulo dedicado a R Markdown .

Y también muy inspirado de estos otros libros:

- *Data Visualisation with R*, Thomas Rahlf, 2014, <http://www.datavisualisation-r.com/>
- *R Graph Cookbook* , Hrishi Mittal, 2011