GSyC/Libresoft (URJC)

Curso básico de estadística para investigadores de software libre

José Felipe Ortega Soto

Doctor Ingeniero de Telecomunicación

Resumen

Desde hace algunos años, nuestro grupo se ha especializado en el la obtención, tratamiento y análisis de una gran cantidad de datos, procedentes en su mayoría de proyectos de software libre u otro tipo de comunidades o proyectos que comparten esta filosofía de organización.

No obstante, en muchas ocasiones los análisis que se realizan sobre la enorme cantidad de datos recolectada se ven limitados por el desconocimiento del investigador de las herramientas más apropiadas para poder explorar dichos datos y extraer conocimiento que resulte útil para su labor científica.

En esta primera entrega, abordaremos los aspectos más básicos sobre estadística descriptiva, tanto desde el punto de vista teórico (conceptos y técnicas) como de las cuestiones más prácticas (cuando es más conveniente usar una u otra técnica o trucos para que el análisis sea más eficiente).

Lo que en principio estaba planificado como una serie de apuntes completos se ha convertido, por el momento, en unos cuadernos de ejercicios guiados, en los que se ofrecen referencias a las fuentes de información más útiles que el lector debe consultar para tener una visión completa de cada aspecto tratado. En toda esta serie de entregas, se utilizará GNU R como herramienta básica para realizar análisis estadísticos. Puesto que va dirigido a investigadores de software libre, todos los datos que se usan para los ejemplos provienen de las bases de datos generadas en el proyecto FLOSSMetrics. Por ello, agradecer desde aquí a todos los compañeros que han colaborado porque este proyecto sea una realidad.

Índice

Capítulo 1

Análisis descriptivo de datos

1.1 Introducción a la estadística descriptiva

Una de las herramientas más básicas para científicos e investigadores es la **estadísitica descriptiva**, que agrupa una serie de técnicas de análisis y representación gráfica para averiguar algunas propiedades básicas de los datos obtenidos. Normalmente, son técnicas muy sencillas, pero esenciales. Constituyen casi siempre el primer páso que tenemos que dar en un **análisis empírico** (es decir, cuando partimos de los datos para para validar hipótesis o intentar extraer conclusiones).

Imaginemos que estamos analizando el número de *commits* que un grupo de desarrolladores ha efectuado en un proyecto de software libre, durante el mes de junio de 2009. Podemos cargar los datos de ese proyecto de la base de datos correspondiente en FLOSSMetrics, generada con CVSAnalY. Tras una consulta a la base de datos obtenemos un listado con todos los desarrolladores que enviaron cambios al repositorio durante ese mes, y el número total de cambios que hizo cada uno. Llegados a este punto, no podemos comenzar a analizar los datos sin más, intentando responder preguntas que podamos formularnos. Un paso previo imprescidible es conocer detalles sobre el conjunto de datos con el que vamos a trabajar:

- ¿Cuál es el mayor número de *commits* realizados por un desarrollador durante este mes? ¿Y el menor?
- ¿Todos los desarrolladores hicieron al menos un cambio durante ese mes? En caso contrario, ¿necesito incluir también a los que no lo hicieron?
- ¿Cual es la media aritmética de *commits* efectuados durante ese mes? ¿Varía mucho la cantidad de *commits* que ha hecho cada desarrollador con respecto a ese valor medio?

Estas y otras muchas preguntas son las que vamos a tratar de responder a lo largo de esta primera entrega. Pero antes de entrar en materia, tenemos que conocer un poco al que será, a partir de ahora, nuestro mayor aliado en este camino.

1.2 Introducción a GNU R

GNU R [3] es un paquete de software libre para análisis estadísticos. Su gran facilidad de manejo, potencia, y el hecho de ser precisamente software libre, hacen que hoy día esté ya

cerca de convertirse en el estandar *de facto* para estudios estadísticos de cualquier tipo. Se trata de un entorno modular, en el que tenemos a nuestra disposición una gran cantidad de librerías que cubren la mayor parte de las necesidades de un amplio espectro de investigadores.

Como veremos, solo algunas de estas librerías se incluyen en la instalación básica, de manera que se pueden ir instalando y cargando librerías nuevas conforme sea necesario. Si echamos un vistazo a la página web donde se centraliza toda la información sobre librerías en R (CRAN, *Comprehensive R Archive Network*), ubicada en la URL http://lib.stat.cmu.edu/R/CRAN/) veremos que ya hay disponibles casi 1.900 paquetes o librerías para este entorno. Esto cubre la mayor parte de análisis estadísticos conocidos en la actualidad. Otra ventaja de contar con estas librerías es que han sido comprobadas y mejoradas por algunos de los mejores especialistas y programadores, lo que supone una garantía para la fiabilidad de los análisis que vamos a realizar (además de quitar una gran cantidad de trabajo al investigador, que no necesita construir sus propios métodos).

1.2.1 Instalación y primeros pasos

Para instalar GNU R y el conjunto de librerías básicas recomendadas, simplemente instalamos el meta-paquete correspondiente en Debian o Ubuntu (el proceso es similar en otras distribuciones GNU/Linux):

```
jfelipe@maquina:~\$ sudo apt-get install r-base
jfelipe@maquina:~\$ R

R version 2.9.0 (2009-04-17)
Copyright (C) 2009 The R Foundation for Statistical Computing
ISBN 3-900051-07-0

R es un software libre y viene sin GARANTIA ALGUNA.
Usted puede redistribuirlo bajo ciertas circunstancias.
Escriba 'license()' o 'licence()' para detalles de distribucion.

R es un proyecto colaborativo con muchos contribuyentes.
Escriba 'contributors()' para obtener más información y
'citation()' para saber cómo citar R o paquetes de R en publicaciones.

Escriba 'demo()' para demostraciones, 'help()' para el sistema on-line de ayuda,
o 'help.start()' para abrir el sistema de ayuda HTML con su navegador.
Escriba 'q()' para salir de R.

>
```

Tanto GNU R como la mayoría de paquetes están licenciados bajo GPLv2. Como vemos, tras invocar al programa en línea de comandos, obtenemos el mensaje de bienvenida y el

cursor de comandos de R. Tal y como indica el mensaje, para salir basta con escribir q() (y de momento, decir que no queremos que nos guarde el espacio de trabajo actual).

Por supuesto, lo primero que tenemos que conocer es la forma de cargar librerías y obtner ayuda sobre su contenido y cómo utilizarlas. Sin embargo, puede que todavía no tengamos instalada la librería que necesitamos. para ello, tenemos que ejecutar R como superusuario (para tener los permisos de acceso a los directorios de instalación). Imagninemos que estamos interesados en cargar la librería para *survival analysis*, llamada *survial*, pero todavía no la hemos instalado. Tras ejecutar R como superusuario:

```
> intall.packages("survival", dep = T)
```

Donde le indicamos qué librería queremos instalar (argumento 1) y que instale todas las dependencias de librerías necesarias para que ésta funcione (dependencies=True). En el segundo caso, vemos un ejemplo de cómo podemos acortar los identificadores de argumentos y parámetros (en este caso booleana) en R. Basta con escribir suficientes letras del identificador como para que no haya ambigüedad en la interpretación.

La instalacción nos pedirá seleccionar un mirror de R-CRAN, bajará los paquetes necesarios y los instalará en nuestra máquina. Finalmente, cerramos la sesión, puesto que salvo para instalación de librerías, no conviene ejecutar GNU R como superusuario.

Ahora, cargamos nuestra librería recién instalada y obtenemos algo de información sobre ella:

```
> library(survival)
> library(help = survival)
```

Si por ejemplo estamos interesados obtener ayuda sobre cualquier función de R, usamos el comando *help()*:

```
> help(library)
> `?`(library)
```

Escribiendo *?library* obtendríamos el mismo resultado de forma más compacta. Este método es conveniente cuando recordamos el nombre de la función. Sin embargo, puede suceder que no lo recordemos, o simplemente queramos buscar funciones relacionadas con cierta técnica o método. Para ello disponemos de dos potentes herramientas de ayuda:

```
> apropos(mean)
> help.search(pearson)
```

La primera nos muestra un listado de todas las funciones que contengan la cadena de caracteres que pasamos como argumento. La segunda hace una búsqueda en profundidad en las páginas de manual, devolviendo referencias a aquellas en las que aparece la palabra indicada como argumento.

1.2.2 El entorno de trabajo

Cada vez que abrimos una nueva sesión de trabajo en R, el programa va almacenando todas las variables y objetos que se van creando conforme ejecutamos instrucciones. Para recuperar en cualquier momento un listado completo de los objetos disponibles usamos *ls*:

```
> a = 2
> b = 3
> c = 4
> ls()

[1] "a" "b" "c"

> rm(b, c)
> ls()

[1] "a"

> save.image()
```

Es importante resaltar que *ls()* no va a listar los objetos que ya existan dentro de librerías que tengamos cargadas (como tablas de datos que suelen acompañarlas para los ejemplos). Solo aparecerán objetos que hemos creado nosotros. En este caso, hemos creado dos variables numéricas y les hemos asignado un valor. Si en algún momento vemos que tenemos demasiadas variables en nuestro espacio de trabajo, podemos eliminarlas con el comando *rm*.

Si antes de terminar con la sesión queremos guardar nuestro entorno de trabajo para volver a usarlo la próxima vez, podemos hacerlo con el comando *save.image()*. Este comando crea por defecto un archivo oculto **.RData**, con todos los datos del espacio de trabajo. El archivo se cargará automáticamente la próxima vez que R se ejecute desde ese directorio del sistema. También podemos dar un nombre al archivo para guardar el espacio de trabajo en un punto determinado, por ejemplo:

```
> save.image("example_workspace.data")
```

La extensión que demos al nombre del archivo es indiferente. Más adelante, podemos recuperar un archivo de datos guardado previamente con el comando *load()*.

1.2.3 Introducción rápida al lenguaje de R

El lenguaje R fue creado como un sucesor libre del lenguaje de programación estadístico S (de ahí también el nombre que se le dió, siguiendo la tradición de fina ironía de los entornos UNIX tradicionales, como *more* y *less*). Es un lenguaje extremadamente sencillo de aprender y muy potente (a veces, quizá, demasiado). Con frecuencia, veremos que escribir un comando para lanzar un complejo análisis estadístico no nos ocupa más allá de una línea, y que la dificultad radicará, más bien, en entender para que me sirve ese análisis e interpretar los resultados obtenidos.

Obviamente, es de esperar que R se pueda comportar como una calculadora:

```
> 2 + 3 * (5 + 2 + 7)/4

[1] 12.5

> atan(3)

[1] 1.249046

> sqrt(2)

[1] 1.414214

> pi^2

[1] 9.869604
```

Y observamos que se siguen las clásicas normas de precedencia de operadores de otros lenguajes de programación (R se basa en Fortran y C para implementar muchas de sus librerías). La primera novedad es la forma de definir un **vector** (o *array*) de valores en R, pues necesitamos llamar a la función de concatenación de valores *c()*:

```
> v = c(1, 2, 3, 4, 5)
> v

[1] 1 2 3 4 5

> class(v)

[1] "numeric"
```

Observamos también que simplmenete con llamar al nombre del objeto R nos imprimirá el contenido del mismo. En este caso, el vector únicamente contiene valores numéricos, por lo que la función *class*, que nos devuelve la clase de un objeto que le pasamos como parámetro, nos indica que se trata de un vector de tipo *numeric*. En R, también podemos hacer objetos o vectores que almacenen cadenas de caracteres, que se representan entre comillas simples o dobles. En este caso, el tipo de objeto creado es *character*:

```
> k = c("uno", "dos", "tres")
> k

[1] "uno"  "dos"  "tres"
```

Por supuesto podemos concatenar a su vez vectores entre sí o con otros valores adicionales, mezclando incluso los tipos. Sin embargo, hay que tener cuidado, porque el resultado de mezclar un vector de valores numéricos con un vector de caracteres (o incluso con un solo valor de tipo *character*) provoca automáticamente que todos los valores se transformen en cadenas de caracteres:

```
> z = c(v, k)
> z

[1] "1"    "2"    "3"    "4"    "5"    "uno"  "dos"  "tres"

> class(z)

[1] "character"
```

Para acceder a los elementos de un vector tenemos diferentes tipos de indexación. Conviene advertir que los indices de vectores y cualquier otro elemento indexable comienzan por 1 y no por 0, como en muchos lenguajes de programación:

```
> z[1]

[1] "1"

> z[c(1, 4, 6)]

[1] "1"   "4"   "uno"

> z[1:4]

[1] "1" "2" "3" "4"

> z[-1]

[1] "2"    "3"    "4"    "5"    "uno"  "dos"  "tres"

> z[-c(2, 4)]

[1] "1"    "3"    "5"    "uno"  "dos"  "tres"

> v[v > 1]

[1] 2 3 4 5

> length(v)

[1] 5
```

En el último caso, la expresión *v>1* nos devuelve un vector de valores booleanos, que se usa como argumento para indexar v. En esencia, esto hace que los valores que nos devuelva la indexación sean sólo aquellos para los que se cumple la condición indicada:

```
> v > 1
```

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

Finalmente, R también trabaja fácilmente con aritmética vectorial. Por defecto, toda operación realizada sobre un vector se aplica elemento a elemento a todos los componentes del vector:

```
> v + 1

[1] 2 3 4 5 6

> v^2

[1]  1  4  9 16 25
```

Ejercicio 1

Genere un vector de 10 números, no consecutivos. Usando la ayuda en línea, estudie qué utilidad tiene la función sort(), y trate de usarla con el vector que acaba de crear. A continuación, compruebe el funcionamiento de la función length() sobre el mismo vector.

1.2.4 Creación de *scripts* para R

Para hacer algunas pequeñas pruebas, la línea de comandos de R puede ser muy cómoda (algo así como el intérprete de comandos que encontramos en Python). Sin embargo, funcionalmente es poco práctico que tengamos que repetir continuamente las instrucciones una y otra vez, por lo que rápidamente conviene aprender los rudimentos para construir nuestros *scripts* en R.

Un *script* de R es un archivo que contiene una serie de comandos que serán ejecutados de forma secuencial (R es un lenguaje de programación procedural). El fichero es un archivo de texto corriente que, a pesar de que puede tener cualquier extensión, se suele almacenar como un archivo *.R* para ser fácilmente reconocible. Dentro del archivo podemos ir escribiendo los comandos que se ejecutarán más adelante como si estuviesemos introduciéndolos en la consola de R. También podemos definir nuestras propias funciones (lo que veremos en próximas entregas) para llamarlas en nuestro propio codigo.

Conviene, sin embargo, conocer algunas peculiaridades. Cuando ejecutamos un *script* de R, toda la salida que va generando el programa se va presentando por pantalla. Sin embargo, en ocasiones queremos guardar esa salida en un fichero de texto para inspeccionarla posteriormente, o porque queremos almacenar los resultados obtenidos. Para esto podemos usar la función *sink()*:

```
> sink("foo")
> ls()
> sink()
```

La primera instrucción redirige toda salida producida por R al fichero "foo" en el directorio actual del sistema. Hay se almacena, por ejemplo, la salida de *ls()* y no veremos nada por pantalla. Para restablecer el comportamiento estándar de R, simplemente llamamos a *sink()* sin argumentos.

Finalmente, cuando tenemos preparado y almacenado nuestro *script* en un fichero, por ejemplo "myscript.R", podemos ejecutarlo de dos maneras distintas:

- Desde la consola de R, utilizando la función *source()*.

```
> source("myscript.R")
```

- Desde una línea de comandos del sistema, con la instrucción:

```
jfelipe@machine:~$ R --vanilla < myscript.R
```

1.2.5 Lectura de archivos de datos

Hemos visto antes cómo podemos crear vecotres de valores con GNU R, usando la función *c()*. Sin embargo, este método sería tedioso si tuviésemos que introducir el número de *commits* realizados por 3000 desarrolladores durante el mes de junio. Para agilizar la adquisición de datos en R para su posterior análisis tenemos dos métodos:

- Almacenamos la información en ficheros, que posteriormente leemos en R para cargar los datos y empezar a trabajar con ellos.
- Recuperamos la información directamente de la base de datos

En este apartado vamos a estudiar la primera opción. Lo más frecuente es que los valores que pretendemos leer estén almacenados en un fichero de texto con un formato determinado. Aqui vamos a explicar los dos más frecuentes: el formato tabla y el formato CSV.

El formato tabla almacena los datos de una forma parecida a una base de datos, o al resultado de una consulta a dicha base de datos. Tenemos una muestra por cada fila (siguiendo con nuestro ejemplo, un *committer* por fila) y una serie de columnas que indican los valores de diferentes variables para ese *committer*. Por ejemplo:

```
Committer    num_commits     age     sex
Alice        34              2       F
Bob          235             5       M
Carl         50              3       M
KaL          8525            7       M
....
```

Conviene recalcar la importancia de incluir, en la primera fila, un título para cada una de las columnas, puesto que como veremos inmediatamente, eso nos permite también leerlo al importar los datos y facilita enormemente el acceso a posteriori, porque podemos usar ese nombre para referirnos a una columna, haciendo el código más legible.

El formato CSV *(Comma Sepparated Values)* es de sobra conocido y tomaría el siguiente aspecto:

```
Committer,num_commits,age,sex
Alice,34,2,F
```

```
Bob,235,5,M
Carl,50,3,M
KaL,8525,7,M
....
```

Muchas bases de datos, programas de hoja de cálculo, etc. son capaces de exportar tablas o resultados de consultas en este formato, lo que lo hace muy popular. El caracter que separa los valores de cada fila es completamente configurable (no tiene por qué ser una coma).

Ejercicio 2

Genere dos ficheros diferentes con los nombres "bar.tab" y "bar.csv", cuyo contenido sea el indicado anteriormente para cada tipo de fichero. Arranque una sesión con R y utilice la ayuda en línea para manejar las funciones read.table() y read.csv() para cargar los datos de cada fichero en dos variables llamadas x e y, respectivamente. Como sugerencia, no olvide la importancia de usar el argumento header en ambos casos. ¿De qué tipo son las variables x e y? Inspeccione el contenido de las variables generadas.

1.2.6 Data frames en R

Si hemos sido capaces de completar el ejercicio anterior, entonces hemos creado uno de los tipos más importantes de objetos que podemos encontrar en GNU R: un *data.frame*. Se trata de un registro tabulado de datos que guarda, para cada muestra recogida, el valor obtenido en dicha muestra respecto a varias variables de interés para nuestro análisis. Tal y como hemos visto, cada muestra corresponde a una fila del *data.frame*, y las columnas nos van dando los valores obtenidos en dicha muestra para cada una de las variables de estudio.

Por ejemplo, como resultado de la lectura de los ficheros anteriores tenemos un *data.frame* con un *committer* por fila. En la primera columna tenemos los nombres de los *committers*. En la segunda columna tendríamos el número total de *commits* que hizo durante junio de 2009 en el proyecto estudiado. En la tercera, aparece la edad del desarrollador en el proyecto (tiempo que lleva participando) y en la última el sexo del desarrollador. Como podemos ver, es típico encontrarnos *data.frames* que mezclan variables cuantitativas y cualitativas.

Los *data.frames* pueden considerarse como arrays bidimensionales, a la hora de indexar su contenido. Sin embargo, el hecho de que las columnas (y a veces las filas) tengan nombres asignados, permite acceder a su contenido de forma más legible:

```
> load("y.dat")
> y[1, 2]

[1] 34

> y[1:2, ]

  Committer num_commits age sex
1     Alice          34   2   F
2       Bob         235   5   M
```

```
> y[, 1:2]

  Committer num_commits
1     Alice          34
2       Bob         235
3      Carl          50
4       KaL        8525

> y$Committer

[1] Alice Bob   Carl  KaL
Levels: Alice Bob Carl KaL

> y$num_commits[3]

[1] 50
```

En el primer ejemplo, consideramos el *data.frame* como una matriz, y obtenemos el elemento en la posición (1,1). El segundo ejemplo accede a las dos primeras filas de valores de todas las columnas, al contrario que el tercer ejemplo (acceso a todos los valores de las dos primeras columnas)

Como todo *data.frame* debe tener sus columnas nombradas, podemos acceder a todos los valores de una columna con la notación *$*, seguida del nombre de la columna. El resultado que nos devuelve la indexación mediante *$* es un vector, que podemos indexar a su vez de la forma habitual.

Para conocer los nombres de las columnas de un *data.frame* usamos *names()*:

```
> names(y)

[1] "Committer"   "num_commits" "age"         "sex"
```

Antes de continuar, prestemos un poco de atención a la primera columna, con los nombres de los committers. Vemos como, al imprimir sus valores, no nos salen las típicas marcas de comillas para las variables tipo *character*. Por defecto, cuando GNU R encuentra una columna con valores que se pueden considerar cualitativos (por defecto cualquier valor caracter) automáticamente la columna se lee como un tipo de datos especial, diferente del tipo *character*, que se llama *factor*.

```
> class(y$sex)

[1] "factor"

> y$sex

[1] F M M M
Levels: F M
```

Vemos como al imprimir el contenido de la columna nos muestra, en primer lugar, todos los valores almacenados en la misma. Seguidamente, nos dice los valores diferentes que tenemos en esa columna, llamados niveles del *factor*. Este comportamiento es útil para poder manejar variables cualitativas de forma más eficiente.

1.2.7 Recuperación de datos desde MySQL

El paquete Debian *r-mysql* instala en nuestro sistema la biblioteca *RMySQL*, que nos permite acceder directamente desde R a bases de datos almacenadas en MySQL. La librería implementa un estilo de interfaz de accesso muy similar al de Python. La ventaja es que no tendremos que recuperar la información en un archivo de datos para después cargarlo en R. Además, la recuperación desde la propia base de datos es mucho más rápida y eficiente (puesto que soporta, incluso, el uso de cursores).

Consideremos que tenemos cargada la base de datos *fm3_evince_cvs*, correspondiente al sistema de control de versiones de código del proyecto Evince. Dicha base de datos está disponible en los repostiorios de FLOSSMetrics (actualmente en http://melquiades.flossmetrics.org). Una sesión típica podría ser la siguiente (omitimos la salida para mayor brevedad):

```
> library(RMySQL)
> con2 = dbConnect(MySQL(), user = "felipe", password = "mypassword",
+     dbname = "fm3_evince_cvsanaly2_svn_scm")
> tables = dbListTables(con)
> metrics_field_list = fields_dbListFields(con, "metrics")
> resultset = dbGetQuery(con, "select count(*) from scmlog group by committer_id")
> dbDisconnect(con)
```

A continuación, se proponen algunos ejercicios para que el lector se habitúe al empleo de esta interfaz, que será básico en el resto de esta y de las siguientes entregas.

Ejercicio 3

Utilice ?RMySQL para obtener un ejemplo muy básico de una sesión estándar de conexión a MySQL desde una sesión de R. Observe con cuidado cómo se implementan los métodos para ir leyendo poco a poco el resultado de una consulta (clásicas funciones sendQuery y fetch) y para ejecutar más de una consulta a la vez.

Ejercicio 4

¿De qué clase es el objeto que obtenemos como resultado de la ejecución de una consulta de tipo dbGetQuery? Examine los nombres de las columnas para comprobar por qué tiene sentido que este sea el comportamiento estándar de R.

Ejercicio 5

Elabore un script en R que se conecte a la base de datos fm3_evince_cvsanaly2_svn_scm (que deberá tener previamente cargada). Obtenga los siguientes resultados de la base de datos, enviando la parte inicial de los resultados de las consultas a 3 ficheros llamados "query1.txt", "query2.txt" y "query3.txt". Para imprimir solo la parte incial de cada resultado al fichero correspondiente, examine la utilidad de la función head():

1. query1: id, committer_id y date de todos los commits efectuados antes del 1 de enero de 2008.

2. query2: El número total de commits que efectuó cada committer durante el mes de junio de 2007.

3. query3: El número total de commits registrados en el proyecto.

Una vez que podemos manejarnos de forma básica con la adquisición de datos desde BD, comenzamos a entrar en materia de estadística descriptiva.

NOTA: Para aquellos lectores que quieran dar un paseo más detallado por las funcionalidades básicas de R, el mejor lugar para hacerlo son los dos primeros capítulos de [1] (libro por otra parte altamente recomendable).

1.3 Medidas básicas de resumen

Comenzamos nuestro recorrido por la estadística descriptiva con algunas medidas básicas que nos sirven para resumir las características más sobresalientes de los datos obtenidos en un estudio empírico. A lo largo de esta y las siguientes secciones, se hará en ocasiones referencia al libro de estadística para ingeniería de Montgomery y Runger [2], por constituir la referencia más fiable para aquellos lectores que necesiten aclaraciones adicionales.

Las medidas básicas de resumen de un conjunto de datos están divididas en dos grandes grupos:

- Medidas de centralidad: Agrupan parámetros que nos indiquen características relevantes sobre la forma en la que están distribuidos los valores que hemos recogido. La media es un ejemplo.
- Medidas de dispersión: Comprenden una serie de parámetros que nos indican el rango que toman los valores que hemos recogido, y su posición con respecto a algunas medidas de centralidad (frecuentemente, respecto de la media). La desviación típica es un ejemplo.

En todas ellas, tendremos que distinguir entre el caso en que la medida se calcule sobre toda la **población** de valores empíricos que se han obtenido (o se podrían obtener), o bien, que la medida se calcule sobre un subconjunto más pequeño de valores, tomados de dicha población (muchas veces aleatoriamente), llamado **muestra**. En general, cuando nos encontramos con poblaciones muy grandes (imaginemos, por ejemplo, todo el historico de cambios en el proyecto Linux) utilizar absolutamente todos los valores disponibles podría ralentizar mucho los cálculos. En otros ámbitos, puede ser incluso inviable hacerlo (encuestas a nivel nacional, o estadísticas sobre las piezas fabricadas en una nave industrial), porque llevaría mucho tiempo o porque costaría demasiado dinero. Es por esta razón por la que la mayoría de textos de estadística se centran en los procesos de muestreo.

En toda esta serie de cuadernos, incluiremos apartados sobre muestreo. Sin embargo, por las prestaciones de nuestros sistemas de recopilación de datos, y la gran capacidad de almacenamiento de los servidores modernos, los investigadores de software libre se encuentran con que tienen fácil acceso a la población completa, y en muchos casos esto no supone un mayor esfuerzo, coste necesario, o un incremento desproporcionado de tiempo de cálculo. Se trata de una situación muy extraña, que puede llevar a errores si no se procede con cuidado, sobre todo porque la mayoría de libros de estadística no contemplan en absoluto esta posibilidad. Siempre que sea posible, haremos las distinciones apropiadas cuando estemos hablando de cálculos a partir de muestras, o sobre la población completa.

Cuando calculamos un parámtero sobre la población total, obtenemos siempre el **valor real exacto** de dicho parámetro (ya que estamos considerando todos los valores posibles de la población). Sin embargo, cuando calculamos un parámetro usando los valores de una muestra (más pequeña que la población), no obtenemos el valor exacto, sino lo que llamamos un *estimador del parámetro*. A veces, los estimadores pueden ser muy precisos, mientras que otras tendremos que tomarlos con muchas precauciones. Esto dependerá, sobre todo, del tamaño de la muestra que tomemos y del estimador en cuestión a calcular.

Por supuesto, si usamos siempre la población completa eliminamos estos problemas, pero a cambio tendremos casi seguro un tiempo de cálculo mucho mayor (aunque sea tan sólo por el elevado número de valores implicados en las operaciones). Como en el caso de muchos de nuestros análisis nos podemos permitir este lujo, es probablemente una de las pocas ocasiones en las que se podría contrastar, de forma real, si lo que dice la teoría sobre la exactitud de un estimador muestral concuerda realmente con lo que observamos en la realidad. Hoy día, esto no está al alcance de casi ningún investigador en muchas otras áreas.

1.3.1 Medidas de centralidad

La media es quizá la medida de centralidad más conocida. Si consideramos toda una población de N valores empíricos, $(x_1, x_2, ..., x_N)$, obtenidos mediante un proceso de captación de datos sobre proyectos de software libre o comunidades abiertas, la **media poblacional** se denota por μ y se calcula de la siguiente forma:

$$\mu = \frac{\sum_{i=1}^{N} x_i}{N} \tag{1.1}$$

También podemos calcular la media de una muestra de valores (*sample mean*) tomados de la población total. Suponiendo que la muestra de valores $(x_1, x_2, ..., x_n)$ tiene tamaño $n < N$, entonces la **media muestral**, $\bar{x}$ se calcula así:

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n} \tag{1.2}$$

Aunque no nos detengamos demasiado a explicar las razones, el hecho de que la media muestral sea tan popular es que se puede demostrar que es el mejor estimador posible para la media de la población. En otras palabras, si se toma una muestra suficientemente grande (ya veremos que tamaño mínimo hay que garantizar en cada caso), y calculamos su media, dicho valor coincidirá (salvo un pequeño margen de error muy pequeño) con la media de la población. Obviamente, el error tiende a cero conforme el tamaño de la muestra crece, acercándose al tamaño de la población total.

Sin embargo, a pesar de que la media sea de lejos la medida de centralidad más popular (usada hasta la saciedad en muchos medios de comunicación), no es precisamente la más *robusta*. La **robustez** de un estimador (o de una medida poblacional de centralidad) se mide por lo sensible que sea su resultado con respecto a los valores incluídos en la muestra (o la población).

Esto se puede entender muy bien con un ejemplo. Supongamos que en un proyecto de software libre, el número total de commits efectuado por sus 5 únicos desarrolladores es la tupla (15,26,17,14,23). La media poblacional en este caso es de 19. Sin embargo, si en lugar de 5 hubiera 6 desarrolladores, y el sexto tuviera un número muy alto de *commits* (p.ej. 250), la media entonces pasa a ser 57.5. Vemos como un pequeño cambio en la población puede variar mucho el valor de su media, y precisamente esta es una situación que encontramos frecuentemente en muchos proyectos (debido al modelo de cebolla que explica el reparto de esfuerzo y el grado de implicación de los participantes).

En [2] (pág. 199) se explica muy bien, haciendo un símil con el centro de gravedad de un sistema de pesos. Si de repente aparece un valor muy extremo, el centro de gravedad se mueve con rapidez hacia esa dirección para compensar. Así ocurre con la media.

Estos valores muy extremos, que pueden alterar la composición de un población, se suelen denominar en inglés **outliers**. Tradicionalmente, han sido muy denostados en todos los análisis, donde se llegaba a recomendar su eliminación (pues a veces surgían de errores de medida). Así por ejemplo, la *media recortada* es una variante de la media en la que se eliminan m valores a ambos extremos de la población, para intentar obtener un resultado (o estimador) menos influenciado por dichos valores. Sin embargo, hoy día la tendencia es la opuesta. Los **outliers** muchas veces aportan información crucial, y no es admisible eliminarlos sin más. Por ejemplo, en un proyecto de software libre eliminaríamos a algunos de los "top-committers" del núcleo de un proyecto. La misma consideración valdrá también cuando veamos modelos lineales.

Una estrategia más correcta (y rigurosa) es la de conocer las limitaciones de cada medida de resumen, y complementar por ejemplo la media con otras medidas adicionales. La segunda más utilizada es la mediana. Supongamos que ordenamos todos los valores de la población, o de la muestra, de menor a mayor. Se define la **mediana** como el valor que cae exactamente en la mitad de dicha lista ordenada. Si el numero de valores de la listat es par, se toma siplemente la media aritmética de los dos valores intermedios.

La forma más fácil de interpretar la mediana es pensar que es el valor que deja el 50% del número total de valores de la población (o muestra) por encima, y el otro 50% de valores por debajo del mismo. Es sencillo ver que esta medida es mucho más robusta frente a *outliers* que la media. Incluir un nuevo valor (por muy alto o bajo que éste sea) influirá muy poco en la posición de la mediana. Para poblaciones o muestras que icluyen una gran cantidad de valores extremos, la media y la mediana pueden ser muy distintas. Esto suele ser un claro síntoma de que nos encontramos ante esa situación.

Así pues, la mediana divide a la población o muestra ordenada en dos mitades. Si extendemos este concepto aún más, podemos generar más medidas de centralidad. Los *cuartiles* dividen a la población o muestra ordenada en 4 partes iguales. A su vez, los *deciles* la dividen en 10 partes iguales, y los *percentiles* en 100 partes iguales. Por tanto, si ordenamos los valores de la muestra o población, el primer cuartil deja el 25% del numero total de valores por debajo y el 75% de valores por encima. El percentil 95 dejará 95% del número total de valores por debajo y 5% por encima.

Ejercicio 6 *¿A qué cuartil corresponde el valor de la mediana de una muestra o población? ¿Y a qué percentil?*

Como vemos, las medidas de centralidad nos ofrecen información complementaria. La función de R *summary()* nos da una lista de medidas de centralidad sobre un conjunto de

valores. Estas medidas son: el valor mínimo, el primer cuartil, la mediana, la media, el tercer cuartil y el valor máximo. Con esto nos podemos hacer una idea rápida de cómo se distribuyen los valores.

```
> f = c(15, 26, 17, 14, 23, 250)
> summary(f)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  14.00   15.50   20.00   57.50   25.25  250.00
```

La función *summary()* no sólo se puede aplicar a vectores numéricos. En particular, se comporta de forma muy inteligente con *data.frames*:

```
> summary(y)

 Committer  num_commits          age       sex
 Alice:1   Min.   :  34.0   Min.   :2.00   F:1
 Bob  :1   1st Qu.:  46.0   1st Qu.:2.75   M:3
 Carl :1   Median : 142.5   Median :4.00
 KaL  :1   Mean   :2211.0   Mean   :4.25
           3rd Qu.:2307.5   3rd Qu.:5.50
           Max.   :8525.0   Max.   :7.00
```

Como vemos, automáticamente se ejecuta la función de obtención de medidas de centralidad sobre cada columna. Para las columnas con valores numéricos, el resultado es el indicado antes. Si la columna contiene un factor, en este caso lo único que se ofrece es un conteo de las veces se repite cada uno de los niveles encontrados (como vemos por ejemplo en la columna *sex*.

1.3.2 Medidas de dispersión

Las medidas anteriores intentar dar una idea de cómo se distribuyen los valores de una población o muestra. Sin embargo, esto no basta para caracterizarlas adecuadamente. Necesitamos más información, en especial, sobre la dispersión o situación de los datos respecto a las medidas de centralidad. ¿Hay muchos valores bajos en la población o muestra? ¿Hay por el contrario muchos valores altos? ¿Los valores tienden a estar agrupados en torno a la media, o están por contra muy alejados?

La primera de estas medidas de dispersión es la *varianza*. Se puede entender la varianza de una población como una medida de la variabilidad de los valores recogidos en dicha población respecto a la media. La *varianza poblacional*, para una población de N valores, se calcula como:

$$\sigma^2 = \frac{\sum_{i=1}^{N}(x_i - \mu)^2}{N} \tag{1.3}$$

Sin embargo, la *varianza muestral*, para una muestra de n valores tomados de una población, se calcula así:

$$s^2 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1} \tag{1.4}$$

Mucho cuidado, porque como vemos las expresiones son diferentes. Además, casi siempre los programas de cálculo estadístico (y R no es una excepción) calculan la varianza usando la segunda fórmula, puesto que como ya hemos dicho, la norma general es trabajar con muestras tomadas de la población en lugar de con la población completa.

Es importante ver la diferencia entre el cálculo de la varianza poblacional y muestral. Esta diferencia es una fuente común de dudas y errores. ¿Por qué dividir la segunda por $n-1$ en lugar de por N? ¿No queremos, en ambos casos, calcular la media del cuadrado de distancias de los valores con respecto al valor medio? En efecto, así es, pero hay matices importantes. Para empezar, la varianza poblacional es un valor exacto, mientras que la varianza muestral es una *estimación* de la varianza poblacional real. En el primer caso, el valor medio μ en el numerador es un valor exacto, obtenido a partir de toda la población. En el segundo caso, primero hemos tenido que calcular una estimación de la media (la media muestral $\bar{x}$).

Por ello, la varianza poblacional se tiene que dividir por $n-1$. Según la estupenda explicación que encontramos en [2] (pág. 202), lo podemos ver de dos formas. La primera es que las observaciones incluídas en una muestra (x_i) tienden a estar más cerca de su media ($\bar{x}$) que de la media poblacional real (μ). Por eso, si dividimos por $n-1$ en lugar de N obtenemos una estimación más precisa de la varianza (si no lo hiciéramos así, el estimador s^2 obtenido sería siempre más pequeño que σ).

Otra forma de verlo es con el concepto, un tanto abstracto, de ***grados de libertad***. Como hemos tenido que usar los valores de la muestra (x_i) para calcular la media muestral $\bar{x}$, en realidad ya no tenemos n grados de libertad para calcular la varianza muestral, porque dados $n-1$ valores y la media muestral $\bar{x}$ siempre obtendremos automáticamente el valor que falta (en este caso, porque la suma de las distancias de todos los valores a la media muestral ha de ser 0). Del mismo modo, si en otra fórmula usasemos dos estimadores, obtenidos a partir de las muestras x_i extraídas de una población, y necesitamos calcular una media aritmética, tendríamos que dividir por $n-2$. Al haber calculado previamente 2 estimadores usando los datos, nos quedan sólo $n-2$ grados de libertad de los n iniciales.

La **desviación típica**, como su propio nombre indica, nos da una idea de lo alejados que suelen estar los datos de esa población o muestra respecto al valor medio. Siempre se calcula como la raíz cuadrada de la varianza (ya sea muestral o poblacional):

$$s = \sqrt{s^2}\ \sigma = \sqrt{\sigma^2} \tag{1.5}$$

En R, la varianza muestral se obtiene fácilmente con la función *var()*, mientras que la función *sd()* nos da la desviación estándar de la muestra. R no incluye funciones para la fórmula de la varianza o desviación estándar poblacional (como hemos dicho, en casi todas las demás áreas es imposible acceder a todos los datos de la población).

Ejercicio 7 *Utilice la función sum() de R para calcular la varianza poblaciónal del número total de commits que ha efectuado cada desarrollador de evince a lo largo de toda la vida del*

proyecto. Use la función sqrt() para obtener la desviación típica de dicha población de valores. Calcule también la media poblacional (en este caso, sí que puede usar la función mean).

Ejercicio 8 *La función sample() permite obtener muestras aleatorias de una población. Tan solo tenemos que pasarle como parámetro size el tamaño de la muestra deseado (véase ?sample). Utilice esta función para tomar una muestra aleatoria del número total de commits de 10 desarrolladores de evince. Calcule la media, la varianza y la desviación típica de la muestra, usando las funciones proporcionadas por R. Repita el proceso dos veces más, obteniendo una nueva muestra aleatoria de la población en cada ocasión, (se recomienda guardar cada muestra nueva en una variable diferente, para poder recuperarlas luego). Calcule ahora la varianza y la desviación típica muestral operando con R, dividiendo por n en lugar de $n-1$. Compare los resultados de los estimadores obtenidos a partir de las muestras con los valores poblacionales reales que obtuvo en el Ejercicio 7.*

Aunque la media muestral y la varianza muestral (tal y como se ha definido) sean buenos estimadores de los valores poblacionales reales, la desviación típica es un caso aparte. La raíz cuadrada malogra todas las propiedades del estimador s^2, y por tanto, s debe considerarse con precaución.

1.4 Representaciones gráficas

Los valores numéricos son interesantes para resumir las características de una muestra o población. Sin embargo, gráficamente somos capaces de obtener valiosa información adicional, que se nos podría escapar de otra forma. Los siguientes apartados presentan algunas de estas herramientas, fundamentales en el análisis estadístico de exploración de datos.

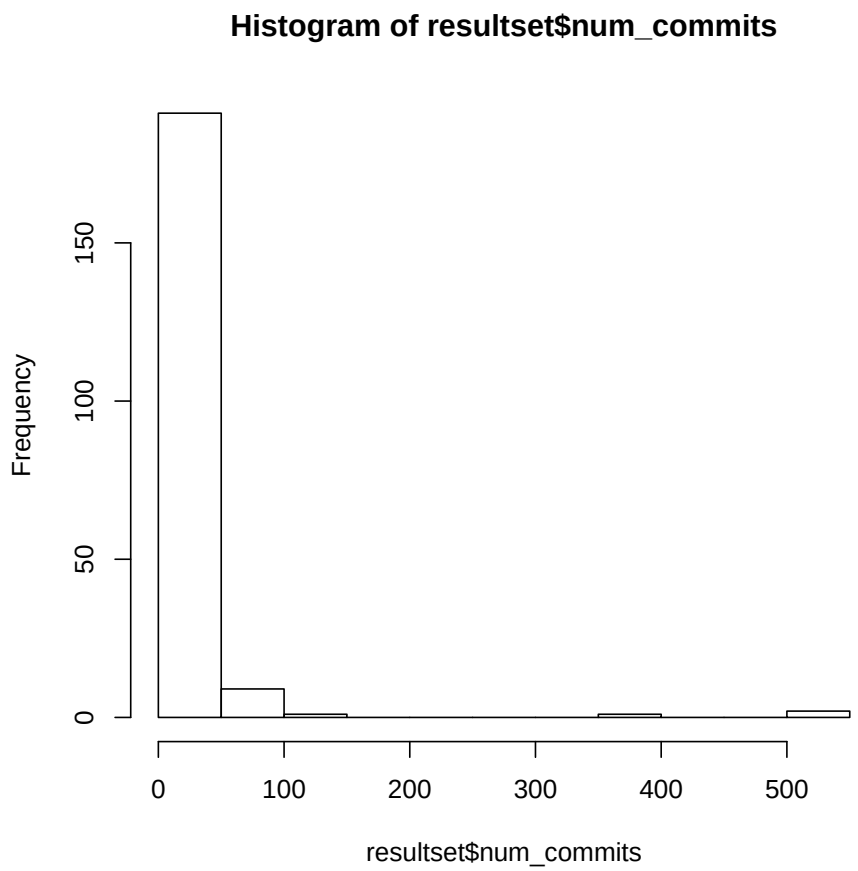
1.4.1 Histogramas y funciones de densidad

El **histograma** es una herramienta muy popular para observar gráficamente la distribución de valores de una población. En R se obtiene mediante el comando *hist()* (se recomienda encarecidamente consultar la página de ayuda de esta función). El histograma dibuja una serie de barras que muestran en el eje vertical el número de valores en la muestra o población que están comprendidos entre los valores mostrados en el eje horizontal.

Por ejemplo, veamos el aspecto que tiene el histograma del número total de *commits* realizados por los desarrolladores de *evince* a lo largo de toda la historia del proyecto:

```
> library(RMySQL)
> con = dbConnect(MySQL(), user = "felipe", password = "mypassword",
+     dbname = "fm3_evince_cvsanaly2_svn_scm")
> resultset = dbGetQuery(con, "select count(*) as num_commits from scmlog group by committ
> dbDisconnect(con)

[1] TRUE

> head(resultset$num_commits)

[1] 102  13  68   2   1   1
```

```
> hist(resultset$num_commits)
```

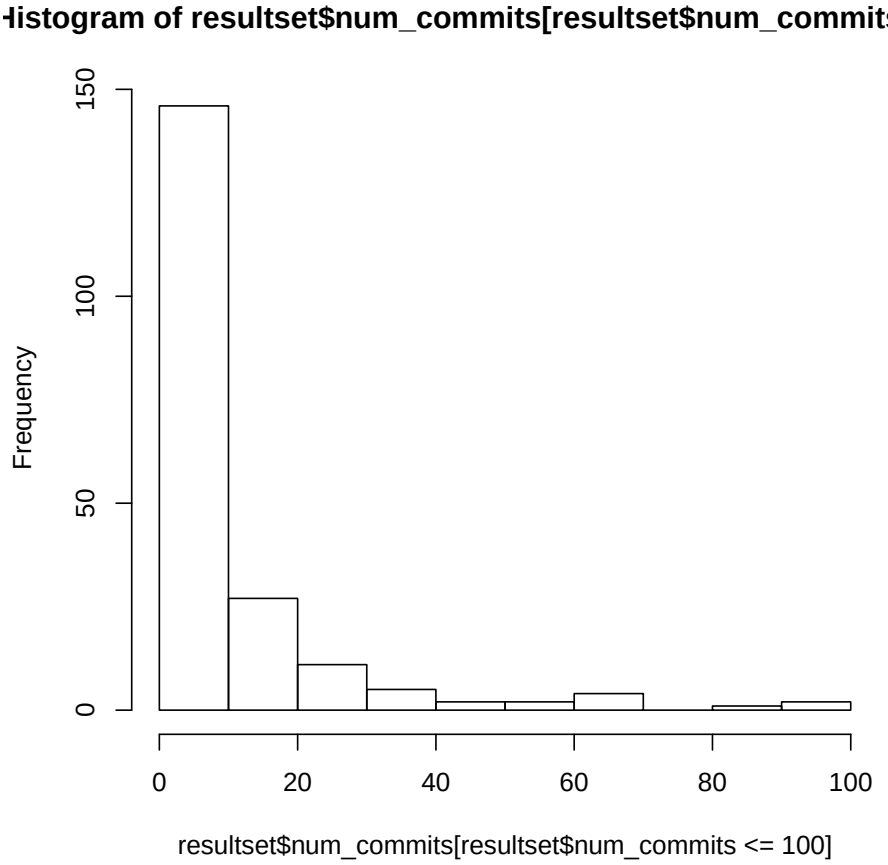


En este caso, vemos un claro ejemplo de una distribución extremadamente asimétrica. Muchísimos valores (casi 200) están comprendidos en el intervalo 0-50. por otro lado, en la parte derecha del gráfico apenas si podemos apreciar las barras que corresponden a los 3 valores registrados por los desarrolladores principales del proyecto:

```
> summary(resultset$num_commits)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.00    2.00    4.00   17.71   12.00  535.00

> head(rev(sort(resultset$num_commits)))

[1] 535 531 375 102 100  97
```

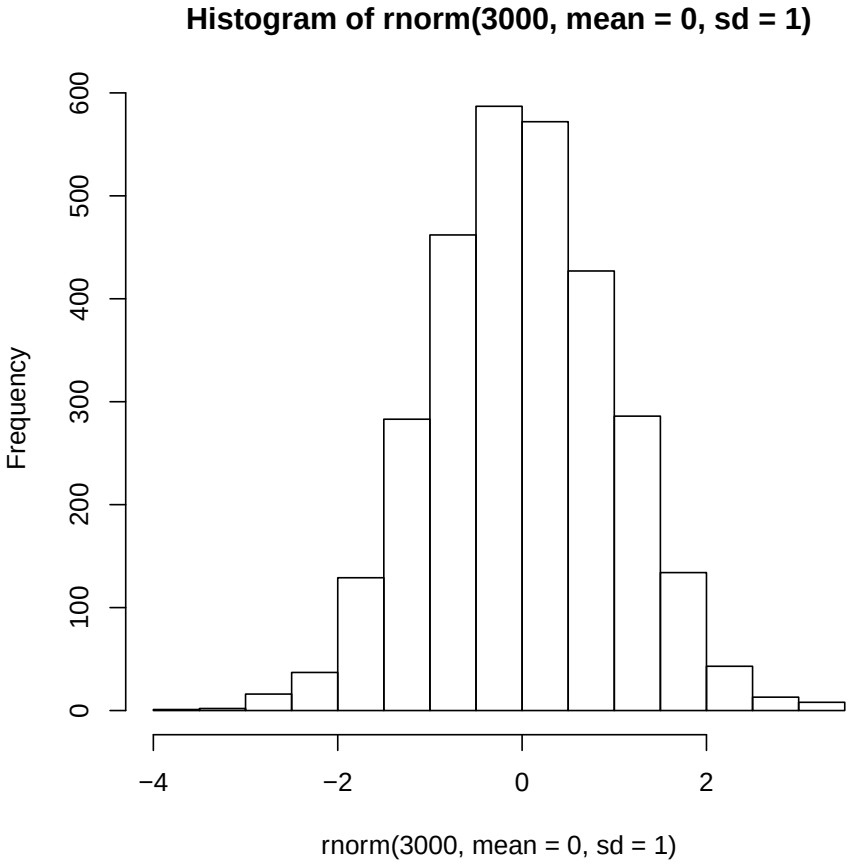
Así pues, en distribuciones como esta (por otro lado muy comunes en proyectos de software libre) el histograma pierde un poco de eficacia. En otro tipo de situaciones, sin embargo su utilidad es más clara. Si queremos ver mejor cómo se distribuyen los valores de la población para los desarrolladores que hicieron 100 *commits* o menos:

```
> hist(resultset$num_commits[resultset$num_commits <= 100])
```



Mucho más habitual resulta que la ditribución de valores de una población o muestra siga una distribución gaussiana o normal. Podemos ver una muestra usando la función de R *rnorm()*, que extrae una muestra del tamaño que queramos, procedente de una población normal:

```
> hist(rnorm(3000, mean = 0, sd = 1))
```



En este caso, la mayoría de valores están concentrados respecto al valor medio (0 para el ejemplo que nos ocupa).

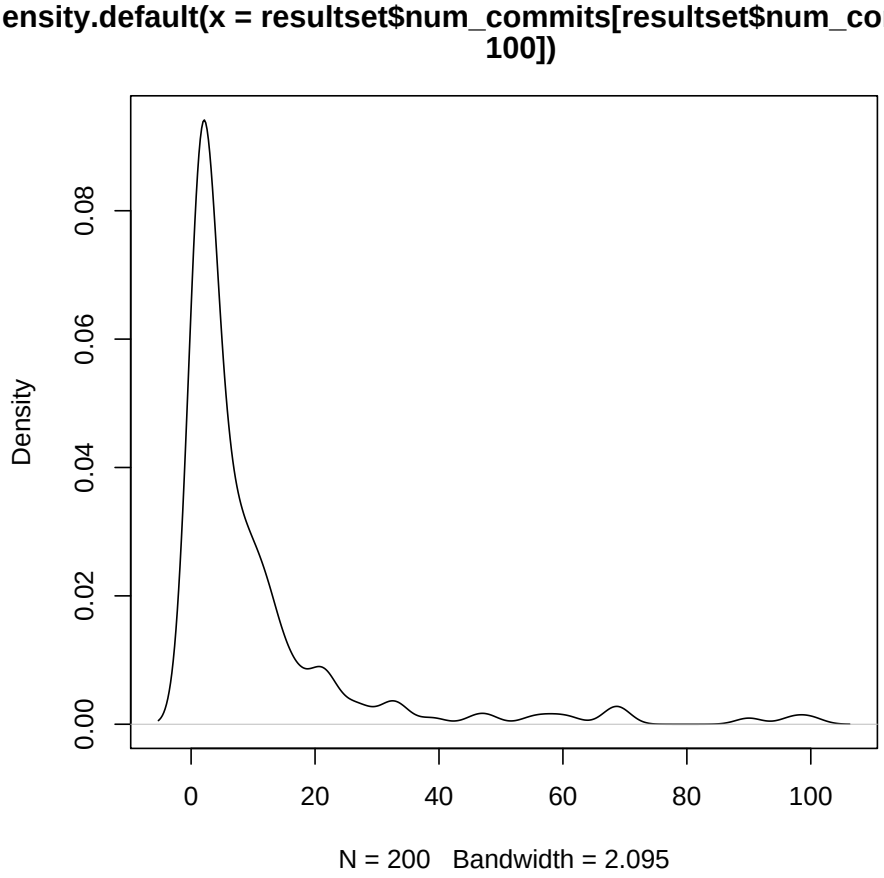
El histograma que hemos construído hasta ahora presenta en el eje vertical el número de valores que caen en cada intervalo. Sin embargo, podemos construir también el histograma con los valores del eje vertical normalizados, de forma que el área total de la gráfica sea 1. En este caso, pasamos el argumento *freq=FALSE* a la función *hist()*. El valor que presenta en ese caso el histograma se denomina **densidad de probabilidad**, ya que representa la probabilidad de que un valor tomado al azar de la muestra o la población caiga en ese intervalo de valores.

Sin embargo, los histogramas tienen un importante problema. Si no escogemos bien la anchura de las barras (es decir, el tamaño de los intervalos) el aspecto del histograma puede variar muchísimo, a pesar de que la distribución de valores analizada sea la misma. Por esta razón, convendría tener un diagrama que, representase en el eje vertical la densidad de probabilidad (siempre es conveniente hacer gráficas normalizadas), pero lo hiciese para intervalos horizontales muy pequeños, idealmente infinitesimales. Con esta idea se creo el diagrama de densidad de probabilidad.

El **diagrama de densidad de probabilidad** utiliza un método matemático para calcular la densidad de probabilidad de una distribución de valores cualquiera. Una de sus supuestas ventajas es que es capaz de extrapolar la forma de la función de densidad para valores fuera del rango de la población o muestra. Veamos el ejemplo que hemos considerado del número total de *commits* por cada desarrollador que ha hecho 100 o menos. La función que calcula

el diagrama de densidad de probabilidad es *density()*. Al contrario que *hist()*, no genera automáticamente un diagrama, sino que tenemos que usar la función genérica de R para dibujar gráficos, *plot()* para la representación (analizaremos brevemente *plot()* más adelante).

```
> plot(density(resultset$num_commits[resultset$num_commits <= 100]))
```



En general, se suele preferir hoy día los diagramas de densidad, puesto que suelen ser más precisos. Sin embargo, conviene tener en cuenta algunos matices. El más importante es recordar que la función de densidad extrapola la forma de la función fuera del rango de valores de la población o distribución. A veces esto puede producir un diagrama poco riguroso, si por ejemplo se genera un trozo de gráfica para valores imposibles (como valores negativos cuando la variable representada es número de *commits*). Para ello, lo mejor será restrigir el rango de representación de la gráfica, actuando con argumentos adicionales en la función *plot()*.

Ejercicio 10 *Calcule el diagrama de densidad de probabilidad del número de committers activos por cada mes de proyecto.*

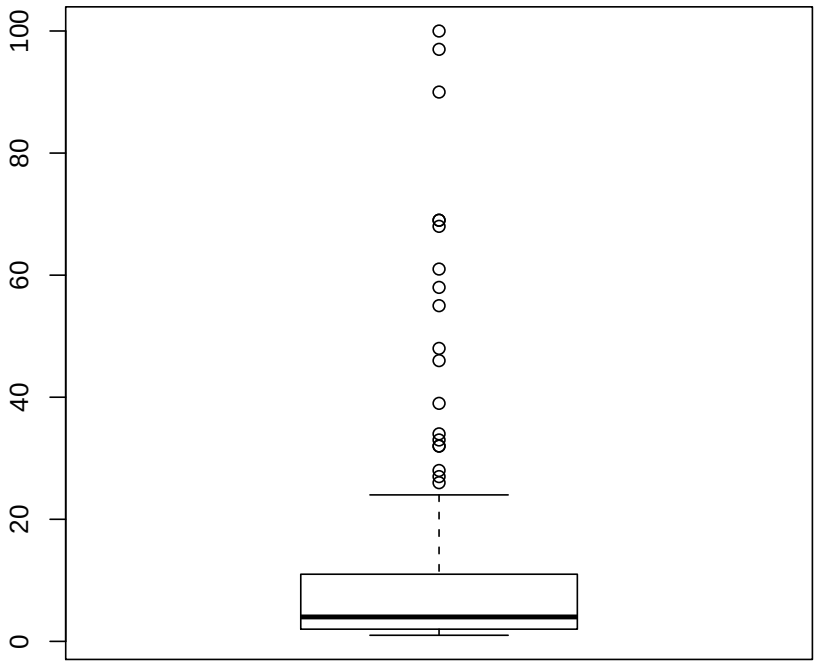
1.4.2 El diagrama box-plot

Otro resumen gráfico muy interesante es el gráfico "box-and-whisker", comúnmente llamado **box-plot**, propuesto por John W. Tukey. De una sola tacada, nos muestra:

- Donde está la mediana de los valores.
- El rango de valores entre el primer y el tercer cuartil, marcado como una caja. A este rango se le conoce como *rango intercuartílico* (IQR) de la distribución de valores.
- Los *outliers* de la distribución. Se consideran como tales para dibujar el boxplot aquellos valores situados a una distancia 1,5*IQR por debajo del primer cuartil o por encima del tercer cuartil.
- Se marca cuál es el último valor de la muestra o población que cae antes del límite marcado para los *outliers* con un par de líneas, llamadas "whiskers" ("bigotes").

Veamos que aspecto tiene este gráfico con el ejemplo del número total de *commits* realizados por los desarrolladores de evince con 100 *commits* o menos:

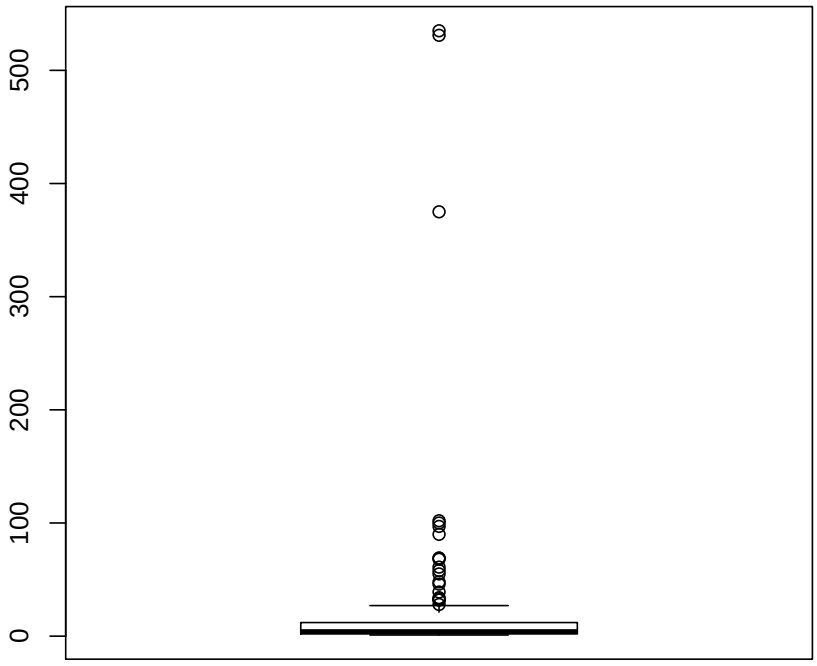
```
> boxplot(resultset$num_commits[resultset$num_commits <= 100])
> summary(resultset$num_commits[resultset$num_commits <= 100])

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.00    2.00    4.00   10.35   11.00  100.00
```



Aquí podemos ver la caja que comprende el IQR, la línea negra gruesa que marca el valor de la mediana y los whiskers, que marcan el límite de los *outliers*. Por la parte inferior, vemos que no hay *outliers*, al contrario que en la parte superior, donde muchos puntos se consideran como valores extremos y se representan individualmente.

Si lo comparamos con el diagrama para la población total:

```
> boxplot(resultset$num_commits)
> summary(resultset$num_commits)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.00    2.00    4.00   17.71   12.00  535.00
```



Observamos que de nuevo, el gráfico sufre las consecuencias de una distribución muy asimétrica, con demasiados valores en la parte más baja del rango. No obstante, también nos sirve para ver cómo los 3 desarrolladores principales del proyecto destacan claramente sobre el resto.

Ejercicio 10 *Obtenga el diagrama box-plot de una muestra de 2000 valores tomados aleatoriamente de una población normal.*

1.4.3 Gráficos en R

Antes de finalizar con esta primera entrega, nuestra introducción a R y al análisis de datos no estaría completa si no presentamos la función genérica de creación de gráficas del entorno: la función *plot()*. Decimos que es una función genérica porque, si bien la podemos utilizar de forma autónoma, veremos que muchos objetos de R (modelos lineales, series temporales, etc.) poseen su propia versión de *plot()* que podemos usar para obtener una representación gráfica completa de muchos de sus parámetros descriptivos habituales.

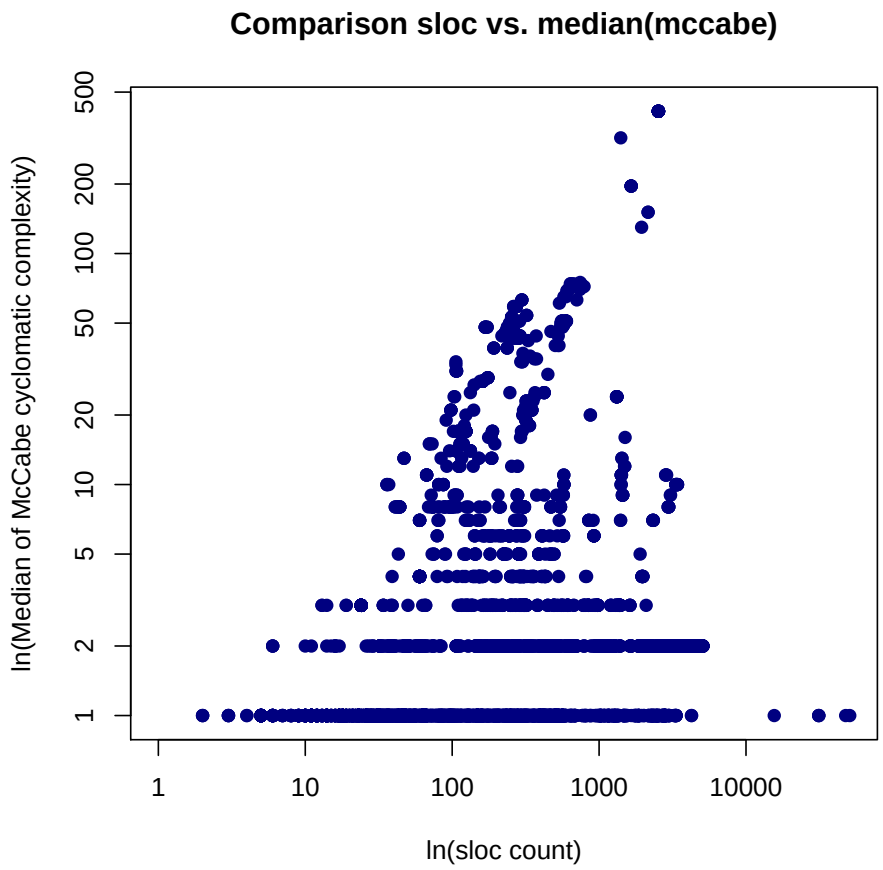
Aquí sólo vamos a presentar algunas nociones básicas sobre el manejo de plot, recomendando al lector como siempre que bucee en la página de ayuda de la función para obtener una idea más completa de sus amplias posibilidades.

El comportamiento estándar de plot es el de dibujar el llamado *scatterplot*, o diagrama de puntos, para observar la posible relación entre dos variables. Por ejemplo, para representar la relación entre la métrica de número de líneas de código fuente (*sloc*) y la mediana de la complejidad ciclomática de McCabe (para valores positivos de ambos), tendríamos:

```
> con = dbConnect(MySQL(), user = "felipe", password = "mypassword",
+     dbname = "fm3_evince_cvsanaly2_svn_scm")
> resultset2 = dbGetQuery(con, "select sloc, mccabe_median from metrics")
> dbDisconnect(con)
```

```
[1] TRUE
```

```
> plot(resultset2$sloc, resultset2$mccabe_median, xlab = "ln(sloc count)",
+     ylab = "ln(Median of McCabe cyclomatic complexity)", main = "Comparison sloc vs. med
+     pch = 19, type = "p", col = "navy", log = "xy")
```



Aquí vemos algunos de los muchos parámetros configurables de plot:

- El primer argumento es el vector de valores en el eje de abscisas, el segundo para el eje de ordenadas.
- *xlab* e *ylab* son las etiquetas para el eje X e Y, respectivamente.
- *main* toma la etiqueta del título principal del gráfico.
- *pch* controla el símbolo (*point character*) usado para la representación. Suele tomar un valor numérico, y los símbolos más comunes están entre el 19 y el 25.
- *type* controla el tipo de gráfico dibujado (por defecto es "p", para un gráfico de puntos, pero en la ayuda podemos ver que hay muchas más opciones.
- *col* controla el color (véase *?colors* y *?colours* para detalles sobre la gigantesca gama de colores de R, que haría empalidecer a los dueños de Titanlux y Bruguer juntos).
- Argumentos adicionales como *log*, que puede tomar los valores "x", "y", o "xy", indicando qué ejes del gráfico queremos que se representen en escala logarítmica (la típica transformación cuando, como en este caso tenemos muchos valores pequeños y algunos pocos demasiado grandes).

Se recomienda echar un vistazo a los ejemplos que acompañan a plot, para hacernos una idea más completa de sus posibilidades. Como en todas las demas funciones de R, podemos acceder a ellos mediante *example*. Así pues, tecleamos *example(plot)* y tendremos un tour guiado por sus múltiples posibilidades.

1.5 Análisis de datos longitudinales

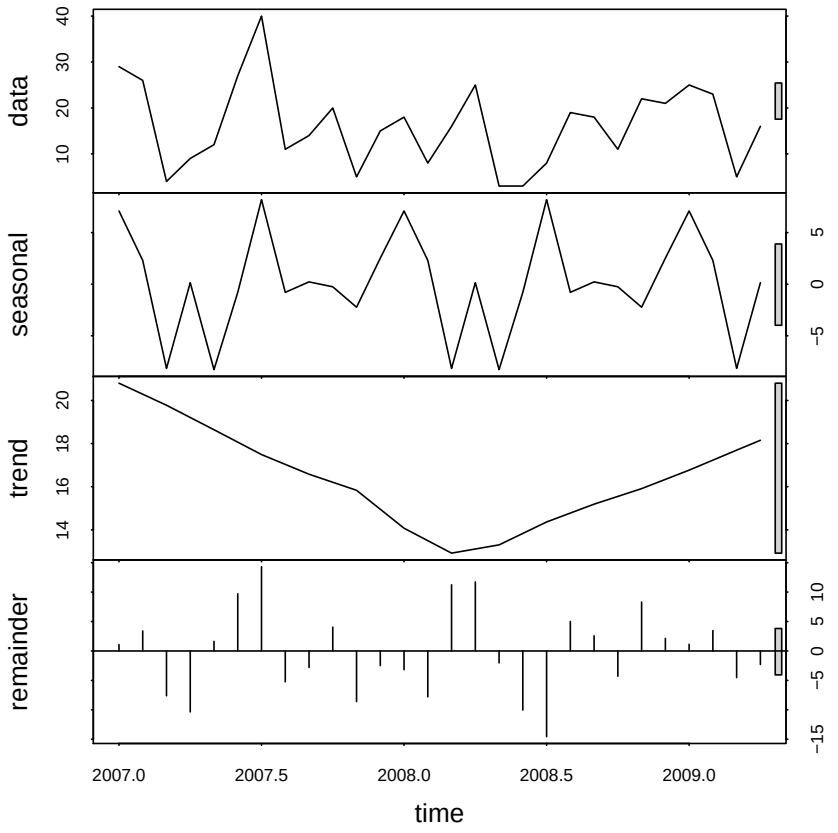
La última sección de esta entrega pretende atajar desde el comienzo otro de los errores más comunes, y desafortunados, que muchos investigadores cometen al analizar datos empíricos. Existe la creencia extendida de que los datos empíricos tienen siempre la misma naturaleza, sin importar el tipo de variables que entran en juego. Esto no es cierto. En particular, si los datos obtenidos son resultados de una o varias variables medidas a lo largo del tiempo, tenemos una situación particular que hay que tratar de forma adecuada.

En estadística descriptiva llamamos ***cross-sectional data*** a medias o datos de una o varias variables que se han obtenido en el mismo instante de tiempo (aproximadamente). Sin embargo, cuando nos encontramos ante medidas de una o varias variables que se obtienen periódicamente, estos datos reciben en inglés el nombre de ***longitudinal data***. En este último caso, para analizar con rigor estos datos tenemos que aplicar un tipo específico de análisis llamado **análisis longitudinal**.

¿Por qué es tan importante esta distinción? Muchas técnicas estadísticas fundamentan su validez en que se han de cumplir una serie de requisitos iniciales. En muchas ocasiones, los requisitos impuestos condicionan que las muestras tomadas sean independientes entre sí, es decir, los valores no tengan ninguna relación causal entre ellos. De otro modo, lo único que estaríamos haciendo es introducir más información de la necesaria en nuestro modelo para explicar el comportamiento que revelan los datos.

En datos longitudinales, casi nunca se da esta situación. Pensemos por ejemplo en el número de *commits* mensuales que hace un desarrollador. Muy probablemente, la actividad que ha venido registrando en los últimos meses va a influenciar notablemente lo que haga en el siguiente mes. Así, es común que en este tipo de datos podamos reconocer **tendencias**. Veamos por ejemplo el caso del desarrollador *carlosgc* en evince, a partir de 2007:

```
> con = dbConnect(MySQL(), user = "felipe", password = "mypassword",
+     dbname = "fm3_evince_cvsanaly2_svn_scm")
> resultset3 = dbGetQuery(con, "select count(*) as num_commits from scmlog where
+ committer_id=(select id from people where name='carlosgc')\nand year(date)>2006
+ group by year(date), month(date) order by year(date), month(date)")
> dbDisconnect(con)

[1] TRUE

> ts_commits_KaL = ts(resultset3$num_commits, start = c(2007, 1),
+     freq = 12)
> plot(stl(ts_commits_KaL, "periodic"))
```



En este caso, tras recuperar los datos utilizamos una función específica de R para trabajar con datos longitudinales (llamados también *series temporales*). La función *ts()* toma los valores numéricos adquiridos periódicamente como primer argumento. Después, le tenemos que decir el valor del primer punto temporal (en este caso, los valores empiezan en enero de 2007) y la frecuencia con la que se tomaron (12 en este caso, al ser mensual).

Para observar el efecto que comentábamos sobre las tendecias periódicas en los datos, podemos usar una sencilla pero potente función de R, llamada *stl()*. Esta función está pensada para extraer la componente estacionaria (*seasonal*) presente en los datos, depurando así la componente de tendencia (*trend*) que explica la evolución a lo largo del tiempo. En este caso, la tendencia de evolución nos muestra que las contribuciones de este desarrollador a evince tendieron a la baja en 2007, y han vuelto a repuntar en 2008. Por otro lado, la componente estacionaria nos explica la parte de hábitos repetitivos en la contribución del desarrollador al proyecto en las diferentes épocas del año.

Conviene de todos modos ser precavidos, puesto que 2 años es un historial bastante corto para confiar en el resultado de este tipo de técnicas de análisis tan simple. Finalmente, comentar también que debemos ser cuidadosos en el proceso de adquisición de datos para evitar sorpresas. En este ejemplo, los datos han sido analizados previamente para comprobar que tenemos resultados para todos los meses del año. En caso de que en algún mes no tuvieramos valor, la función *ts()* simplemente tomaría el siguiente valor disponible para el mes en curso, con lo que la serie se construiría de forma errónea. Aprenderemos más adelante

cómo evitar este tipo de problemas.

1.6 Lecturas complementarias

Hasta aquí la primera entrega de este curso de estadística para investigadores de software libre. No hemos hecho sino adentrarnos mínimamente en las posibilidades que nos ofrece GNU R para estos menesteres, y todavía quedan muchos temas interesantes que iremos tratando en sucesivas entregas futuras.

Sin embargo, para aquellos lectores interesados en profundizar un poco más en los temas recogidos en esta entrega, se ofrecen aquí algunas reseñas a bibliografía complementaria que puede resultar de utilidad:

- Como ya se ha indicado anteriormente, el libro de Montgomery y Runger [2] constituye una referencia de valor incalculable. Es uno de los pocos libros de estadística claros y a la vez rigurosos a la hora de explicar muchos temas que resultan confusos en las obras de otros autores. En particular, el capítulo 6 trata detalladamente todos los temas recogidos en esta entrega.

- El homólogo del libro anterior en el caso de GNU R es sin duda el excelente cuaderno de introducción creado por Peter Dalgaard [1]. Imprescindible para todo aquel que quiera iniciarse seriamente con R. Los capítulos 1, 2 y 4 cubren con más calma y detalle los temas de esta entrega. Abundantes ejercicios (todos ellos resueltos) suponen el perfecto complemento para el aprendiz de GNU R.

Bibliografía

[1] Peter Dalgaard. *Introductory Statistics with R (Statistics and Computing)*, chapter 14. Springer, 2nd edition, August 2008.

[2] Douglas C. Montgomery and George C. Runger. *Applied Statistics and Probability for Engineers, 4th Edition, and JustAsk! Set.* John Wiley & Sons, May 2006.

[3] R Development Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2009. ISBN 3-900051-07-0.