Introducción a R, estadística descriptiva y visualización exploratoria de datos

Federico Alvarez

20 de septiembre de 2018

Introducción

Para poder analizar cualquier conjunto de datos es necesario representarlos de una manera que permita comprender la información que contienen de forma sintética. Para ese propósito, dos procedimientos comunes a todo análisis estadístico son el empleo de estadísticos descriptivos y la representación visual de las muestras. Mientras que la estadística descriptiva condensa los detalles de las observaciones puntuales en una cantidad finita, manejable, de valores informativos, la visualización de los datos permite contemplar la distribución de las variables de estudio en la totalidad de la muestra, permitiendo observar patrones de manera intuitiva. Estas dos herramientas se relacionan entre sí en función de los tipos de variables a representar, y se complementan a la hora de interpretarse: hay parámetros de las representaciones gráficas que se definen en función de estadísticos descrptivos, y a su vez la visualización frecuentemente facilita la interpretación de los valores descriptivos. En el marco de las reuniones de este grupo, este texto pretende complementar la exposición teórica sobre la estadística descriptiva con un andamiaje básico de programación y con los pasos a seguir para computar los estadísticos relevantes, así como con los conceptos básicos para permitir la exploración visual de conjuntos de datos.

Empezando a programar

Básicamente, un lenguaje de programación es un conjunto de reglas sintácticas que permiten proporcionarle instrucciones a una computadora, junto con un vocabulario básico para este fin. En este sentido, toda tarea de programación supone encontrar una secuencia de pasos que permita llevar a cabo la tarea deseada y su subsecuente expresión en la sintaxis de algún lenguaje. Tanto la visualización como el análisis numérico de los datos se efectúan de manera programática. Para eso resulta fundamental el manejo de herramientas computacionales que permitan llevar a cabo los cálculos y las representaciones visuales. Hay varias herramientas disponibles para el análisis de datos que suponen distintos niveles de comprensión de los mecanismos subyacentes al análisis. Frente a los programas de estilo 'point-and-click' como SAS, SPSS o Stata (o Microsoft Excel, también), los lenguajes de programación ofrecen un grado mayor de control sobre la lógica del análisis, además de permitir la realización de una gama de tareas mucho más amplia. Si bien aprender la lógica de un lenguaje de programación requiere de una práctica específica para ese fin, el control sobre el proceso de análisis es mucho mayor, además de abrir la puerta a la realización de una enorme variedad de tareas.

Algunas consideraciones sobre R

Cada lenguaje de programación tiene formas de representación de información que le son propias, llamadas estructuras de datos. Cada estructura de datos conforma una categoría sobre la cual se pueden realizar diferentes operaciones. Las estructuras propias de R que nos importan en este momento son números, caracteres, vectores, data frames y factores. La idea es sencilla: uno puede multiplicar 3 por 4, pero no la letra 'a' por la letra 'b'. Cada tipo de dato puede ser afectado por un conjunto de operaciones que le son propias. El tipo de dato sobre el cual depende la gran mayoría de las operaciones que se realizan en R es el vector, que consiste básicamente en una secuencia de elementos de un mismo tipo, así es posible tener vectores que consisten de secuencias de números o de secuencias de caracteres.

Otro tipo de dato de importancia es el factor, que consiste en una lista finita de valores fijos mutuamente excluyentes. Esto significa que mientras que el tipo de dato 'caracter' sirve para codificar variables categóricas como el nombre de un sujeto, el factor se utiliza para clasificar grupos, por ejemplo 'ancianos', 'adultos' y niños. Esto nos lleva a un punto importante que es que los tipos de datos tienen una relación importante con los tipos de variables que se utilizan para realizar un estudio, y diferentes formas de operacionalización tienen consecuencias directas en el modo en el que el lenguaje interpreta los datos.

Para programar, sin embargo, rara vez manejamos datos en el vacío. En general se los retiene en la memoria de la computadora durante la ejecución del programa de forma tal que cada dato esté disponible para diferentes procesos. Estos sectores de memoria son lo que en programación se llama **variable**. Si bien los conceptos están relacionados, es importante no confundir las variables que forman parte del diseño de un estudio cuantitativo con las estructuras de almacenamiento en memoria que forman parte de un programa. En R las variables se crean al asignarle un nombre a un valor, de la siguiente manera:

```
var_numerica <- 100
print(var_numerica)
```

```
## [1] 100
```

```
var_caracter <- 'cien'
print(var_caracter)
```

```
## [1] "cien"
```

```
var_vector <- c(var_numerica, var_caracter)
print(var_vector)
```

```
## [1] "100"  "cien"
```

```
var_factor <- factor(var_vector)
print(var_factor)
```

```
## [1] 100  cien
## Levels: 100 cien
```

También podemos ver que hay otros componentes en el código de arriba: `<-`, `c()`, `print()` y `factor()`. En el caso de `<-` se trata de un constructo que se llama **operador**. Como su nombre lo indica, sirven para realizar operaciones. En este caso, se trata de una operación de **asignación de variable** (que vamos a usar muchísimo y puede escribirse con el atajo de teclado 'alt' + '-'). Otros operadores son, por ejemplo, `+`, `-`, `*` y `/`, pero no se limita a operaciones matemáticas, sino que hay un gran número de ellos que sirven para realizar diferentes tipos de tareas. Los nombres seguidos de paréntesis, por otro lado, indican que estamos ante una función. En términos escuetos, una función es un procedimiento que toma un conjunto de valores, llamados **argumentos** y devuelve un objeto que resulta de aplicar una serie de procesos sobre dichos valores. La función `c()` toma una secuencia de elementos y devuelve un vector compuesto por ellos. La función `factor()` toma un vector y lo convierte en un factor. La función `print()` toma variables o valores aislados y los imprime en pantalla.

Desde luego, para hacer análisis estadísticos necesitamos un grado de complejidad mayor en nuestros datos y nuestros procesos. En general, en un análisis el tipo de dato que más nos va a importar se llama **data frame**. Un data frame es, básicamente, una tabla. Las filas son observaciones individuales mientras que las columnas son variables bajo análisis. R incluye algunos datos por defecto que podemos utilizar para probar nuestro código. Este es el dataset iris, al que puede accederse con sólo poner `iris` en la consola. Al incluir la función 'head()' estamos diciéndole a R que sólo queremos las primeras observaciones del dataset.

```
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
```

```
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

Este dataset incluye mediciones sobre flores, pero se supone que sin importar el tipo de dato, el formato de carga es similar. Volvamos un momento sobre la función `head()`. Nosotres podríamos querer ver más que sólo los primeros seis registros que la función arroja por defecto. Podemos comunicarle esto a R ingresando un valor numérico dentro de la función.

```
head(iris, 10)
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1           5.1         3.5          1.4         0.2  setosa
## 2           4.9         3.0          1.4         0.2  setosa
## 3           4.7         3.2          1.3         0.2  setosa
## 4           4.6         3.1          1.5         0.2  setosa
## 5           5.0         3.6          1.4         0.2  setosa
## 6           5.4         3.9          1.7         0.4  setosa
## 7           4.6         3.4          1.4         0.3  setosa
## 8           5.0         3.4          1.5         0.2  setosa
## 9           4.4         2.9          1.4         0.2  setosa
## 10          4.9         3.1          1.5         0.1  setosa
```

Esto nos muestra que no todos los argumentos de una función son obligatorios, sino que hay argumentos opcionales. Cada función tiene su propio conjunto de argumentos obligatorios y/o opcionales. En el caso de arriba los argumentos fueron empleados sin referirse a sus nombres porque se enunciaron en el orden prestablecido, pero también es posible invertir el orden y llamarlos de manera explícita.

```
head(n = 3, x = iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
```

Hay veces, incluso, que algunos argumentos opcionales se hallan definidos por defecto y su valor sólo puede modificarse de manera explícita al ejecutar la función.

Más allá de las funcionalidades que provee R por defecto, el lenguaje puede ampliarse. Cuando une usuarie o grupo escribe código que permite realizar de manera más sencilla un procedimiento determinado, puede organizarlo y compartirlo en forma de **librería** o paquete. Las librerías son conjuntos de funciones y tipos de datos creados por usuario que extienden las capacidades del lenguaje de programación.

Esto nos resulta de importancia por una serie de razones. La primera es que muchísimos análisis estadísticos se hallan disponibles en librerías, pero no forman parte del núcleo básico de R, con lo cual para llevarlos a cabo es necesario descargar la librería en cuesión o programar a mano la lógica del análisis. En la medida de lo posible, vamos a evitar hacer a mano lo que otras personas ya hayan hecho mejor que nosotres. En segundo lugar, ocurre que los criterios de diseño de código varían entre programadores, con lo cual hay variaciones en el modo en el que las librerías nombran a las clases y funciones, la forma en la que se definen los argumentos, y la propia manera en la que se ejectuan los análisis. En nuestras reuniones vamos a procurar emplear el conjunto de librerías llamado **tidyverse**, cuya sintaxis es levemente diferente a la de R básico y que emplea tipos de datos adicionales. En particular, nos importa la clase **tibble**, que vamos a usar en lugar del data frame que viene por defecto con R. Si bien las diferencias entre un tibble y un data frame no son aparentes, en realidad tibble es una extensión del data frame: puede hacer todo lo que se hace con data frame común, y algunas cosas más.

Para instalar una librería en R es necesario correr desde la consola la función `install.packages('NOMBRE DE`

LA LIBRERÍA), la cual se ocupa de buscar en línea el paquete, descargarlo e instalar todas sus dependencias. Luego, para incorporar la funcionalidad de la librería al análisis que estemos llevando a cabo, la llamamos explícitamente en la parte superior de nuestro código. Eso nos permitirá emplear sus recursos de manera inmediata.

Cuando nosotros miramos un data frame podemos querer obtener información sobre una columna individual. Por ejemplo, podemos observar la columna 'Petal.Length' de nuestro tbl_iris.

```
tbl_iris$Petal.Length
```

```
##   [1] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 1.5 1.6 1.4 1.1 1.2 1.5 1.3
##  [18] 1.4 1.7 1.5 1.7 1.5 1.0 1.7 1.9 1.6 1.6 1.5 1.4 1.6 1.6 1.5 1.5 1.4
##  [35] 1.5 1.2 1.3 1.4 1.3 1.5 1.3 1.3 1.3 1.6 1.9 1.4 1.6 1.4 1.5 1.4 4.7
##  [52] 4.5 4.9 4.0 4.6 4.5 4.7 3.3 4.6 3.9 3.5 4.2 4.0 4.7 3.6 4.4 4.5 4.1
##  [69] 4.5 3.9 4.8 4.0 4.9 4.7 4.3 4.4 4.8 5.0 4.5 3.5 3.8 3.7 3.9 5.1 4.5
##  [86] 4.5 4.7 4.4 4.1 4.0 4.4 4.6 4.0 3.3 4.2 4.2 4.2 4.3 3.0 4.1 6.0 5.1
## [103] 5.9 5.6 5.8 6.6 4.5 6.3 5.8 6.1 5.1 5.3 5.5 5.0 5.1 5.3 5.5 6.7 6.9
## [120] 5.0 5.7 4.9 6.7 4.9 5.7 6.0 4.8 4.9 5.6 5.8 6.1 6.4 5.6 5.1 5.6 6.1
## [137] 5.6 5.5 4.8 5.4 5.6 5.1 5.1 5.9 5.7 5.2 5.0 5.2 5.4 5.1
```

Como vemos, esto nos devuelve un vector con los valores de esa variable para cada observación. Si nosotros quisiéramos promediar el valor de esta variable entre observaciones empleando la sintaxis normal de R, haríamos lo siguiente:

```
mean(tbl_iris$Petal.Length)
```

```
## [1] 3.758
```

Si luego quisiéramos ver el promedio del largo del pétalo para cada especie de iris (setosa, versicolor y virginica), tendríamos que hacer lo siguiente:

```
mean(tbl_iris[tbl_iris$Species == 'setosa',]$Petal.Length)
```

```
## [1] 1.462
```

```
mean(tbl_iris[tbl_iris$Species == 'versicolor',]$Petal.Length)
```

```
## [1] 4.26
```

```
mean(tbl_iris[tbl_iris$Species == 'virginica',]$Petal.Length)
```

```
## [1] 5.552
```

El código de tidyverse es, en cambio, un tanto más organizado y fácil de seguir:

```
tbl_iris %>%
    group_by(Species) %>%
    summarise(promedio_PetalLength = mean(Petal.Length))
```

```
## # A tibble: 3 x 2
##   Species    promedio_PetalLength
##   <fct>                     <dbl>
## 1 setosa                     1.46
## 2 versicolor                 4.26
## 3 virginica                  5.55
```

El operador %>% ('Ctrl' + 'Shift' + 'M') permite encadenar operaciones y si agregamos un operador de asignación al código de recién, podemos tener un nuevo tibble con los datos que nos resultan de interés.

```
tbl_promedio_PetalLength <- tbl_iris %>%
    group_by(Species) %>%
```

```
    summarise(promedio_PetalLength = mean(Petal.Length))
```

NOTA: Hay maneras más sencillas de hacer las cosas en R base:

```
mean <- tapply(iris$Petal.Length, iris$Species, mean)
mean_df <- as.data.frame(mean)
print(mean_df)
```

```
##             mean
## setosa     1.462
## versicolor 4.260
## virginica  5.552
```

Las cosas se vuelven levemente más problemáticas a la hora de encadenar más operaciones:

```
mean <- tapply(iris$Petal.Length, iris$Species, mean)
sd <- tapply(iris$Petal.Length, iris$Species, sd)
mean_sd_df <- cbind(mean, sd)
print(mean_sd_df)
```

```
##             mean        sd
## setosa     1.462 0.1736640
## versicolor 4.260 0.4699110
## virginica  5.552 0.5518947
```

Esto se vuelve particularmente relevante si queremos computar varias cosas de manera organizada. ¿Cómo se les ocurre que sería el código de R base para lo siguiente?

```
tbl_resumen <- tbl_iris %>%
    group_by(Species) %>%
    summarise(promedio_PetalLength = mean(Petal.Length),
              sd_PetalLength = sd(Petal.Length),
              promedio_SepalLength = mean(Sepal.Length),
              sd_SepalLength = sd(Sepal.Length))

print(tbl_resumen)
```

```
## # A tibble: 3 x 5
##   Species  promedio_PetalL~ sd_PetalLength promedio_SepalL~ sd_SepalLength
##   <fct>               <dbl>          <dbl>            <dbl>          <dbl>
## 1 setosa               1.46          0.174             5.01          0.352
## 2 versico~             4.26          0.470             5.94          0.516
## 3 virgini~             5.55          0.552             6.59          0.636
```

Vale aclarar que es posible ejecutar código de tidyverse empleando la sintaxis normal de R en vez de la sugerida arriba, sin embargo preferimos la opción que mostramos porque muestra de manera más clara la secuencia de pasos, mientras que la variante de aquí abajo muestra la llamada a diferentes funciones en un orden inverso al de la ejecución real.

```
tbl_resumen <- summarise(group_by(tbl_iris, Species),
                         promedio_PetalLength = mean(Petal.Length),
                         sd_PetalLength = sd(Petal.Length),
                         promedio_SepalLength = mean(Sepal.Length),
                         sd_SepalLength = sd(Sepal.Length))
```

Visualización

Para los ejemplos que siguen, vamos a usar datos extraídos de la librería *languageR*, así que si no la tienen instálenla con `install.packages()`. Vamos a elegir el dataset *regularity* contenido en esta librería.

```
library(languageR)

tbl_regularity <- as_tibble(regularity)

tbl_regularity
```

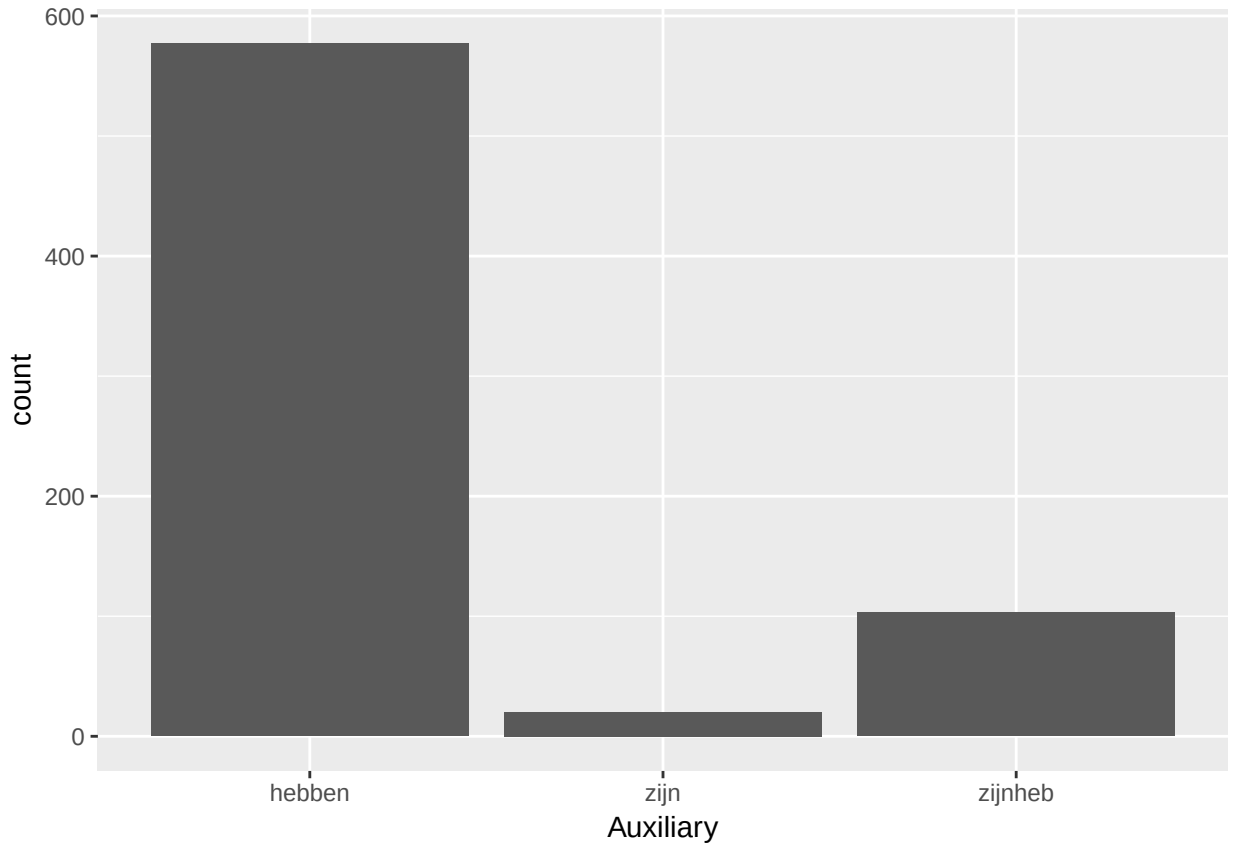
```
## # A tibble: 700 x 13
##    Verb  WrittenFrequency FamilySize LengthInLetters VerbalSynsets
##  * <fct>            <dbl>      <dbl>           <int>         <int>
##  1 stel~             1.61      2.56                5             1
##  2 toll~             5.41      2.40                3             1
##  3 blij~             9.88      1.79                5             1
##  4 gloe~             6.91      2.08                5             3
##  5 kakk~             3.78      2.08                3             1
##  6 morr~             5.44      0.693               3             1
##  7 glim~             7.03      1.79                4             2
##  8 rijz~             7.37      2.20                4             4
##  9 tier~             5.78      1.39                4             3
## 10 werp~             8.49      3.95                4             3
## # ... with 690 more rows, and 8 more variables: MeanBigramFrequency <dbl>,
## #   NcountStem <int>, Regularity <fct>, InflectionalEntropy <dbl>,
## #   Auxiliary <fct>, Valency <int>, NVratio <dbl>,
## #   WrittenSpokenRatio <dbl>
```

Como podemos ver, este dataset contiene información sobre verbos del neerlandés. Los campos son: * Verb: contiene el verbo en infinitivo. * WrittenFrequency: contiene el logaritmo de la frecuencia del verbo. * FamilySize: se supone que contiene la cantidad de types en la familia morfológica del verbo, pero no parece ser eso porque contiene un número decimal. * LengthInLetters: la cantidad de letras de la forma ortográfica de la raíz del verbo. * VerbalSynsets: la cantidad de synsets verbales a los que pertenece en WordNet. * MeanBigramFrequency: promedio del logaritmo de la frecuencia en bigramas. * NcountStem: cantidad de vecinos ortográficos. * Regularity: si el verbo es regular o irregular. * InflectionalEntropy: entropía de Shannon calculada para las variantes flexivas del verbo. * Auxiliary: el auxiliar que toma para los tiempos perfectos. * Valency: la cantidad de argumentos que toma. * NVratio: el logaritmo de la proporción entre los usos nominales y los verbales. * WrittenSpokenRatio: logaritmo de la proporción de la frecuencia de uso entre el neerlandés escrito y hablado.

La información más sencilla para observar en un dataset es la frecuencia de los valores de una variable. Por ejemplo, podríamos tratar de fijarnos cuántos verbos emplean cada auxiliar. Para eso vamos a empezar a ver cómo emplear la librería de visualización de tidyverse, **ggplot2**. Como se dijo antes, tidyverse no es una librería única, sino un conjunto de las mismas con una serie de criterios de diseño en común. Ggplot2 es el núcleo de todas las librerías de tidyverse dirigidas a la representación visual de información, y tiene como peculiaridad que su sintaxis es levemente diferente al resto, fundamentalmente por usar el operador `+` para superponer capas en vez de `%>%` para concatenar operaciones.

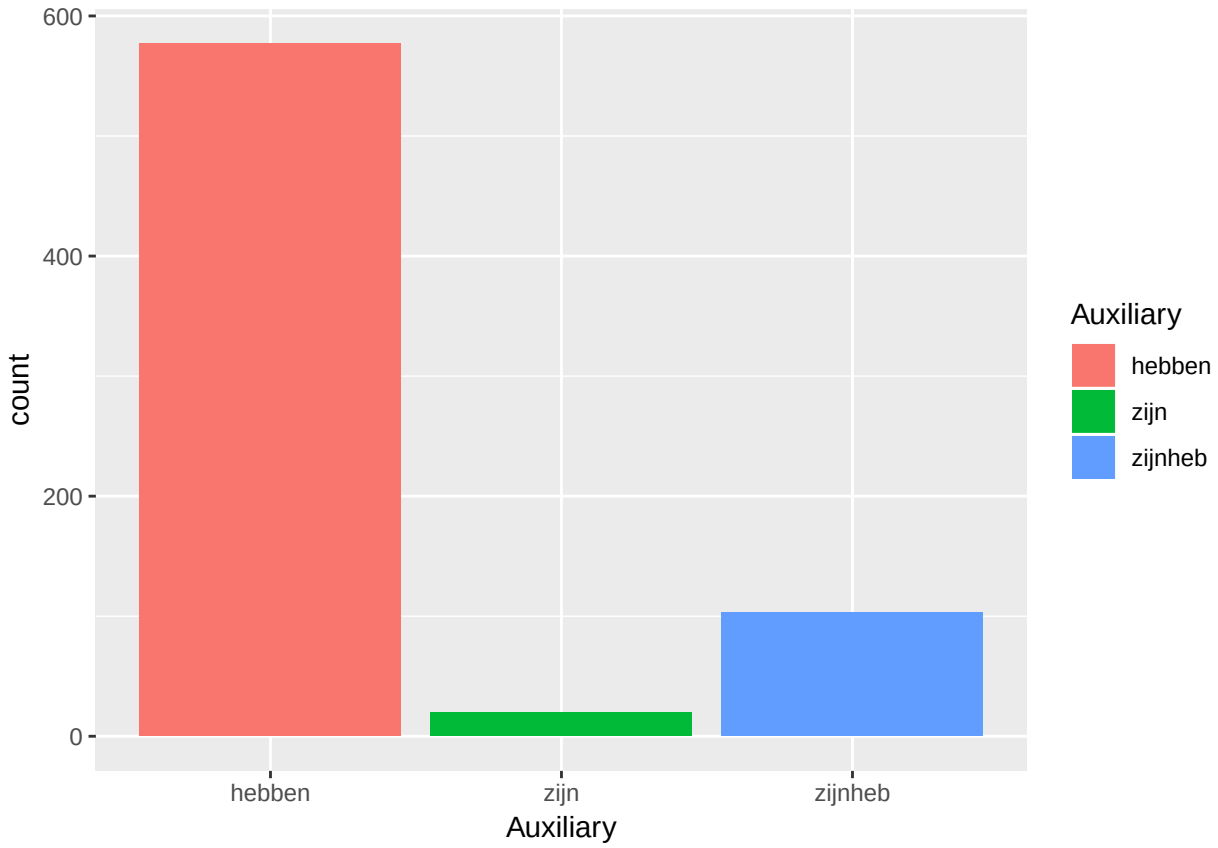
Para crear nuestro gráfico, lo primero que hacemos es invocar a la función `ggplot()` que se encarga de crear el sistema de coordenadas apropiado para el dataset que recibe como argumento. Después añadimos una **capa** consistente en el tipo de representación elegida para nuestros datos. Veamos cómo lucen los gráficos al emplear barras para visualizar las frecuencias:

```
ggplot(tbl_regularity) +
    geom_bar(aes(x = Auxiliary))
```



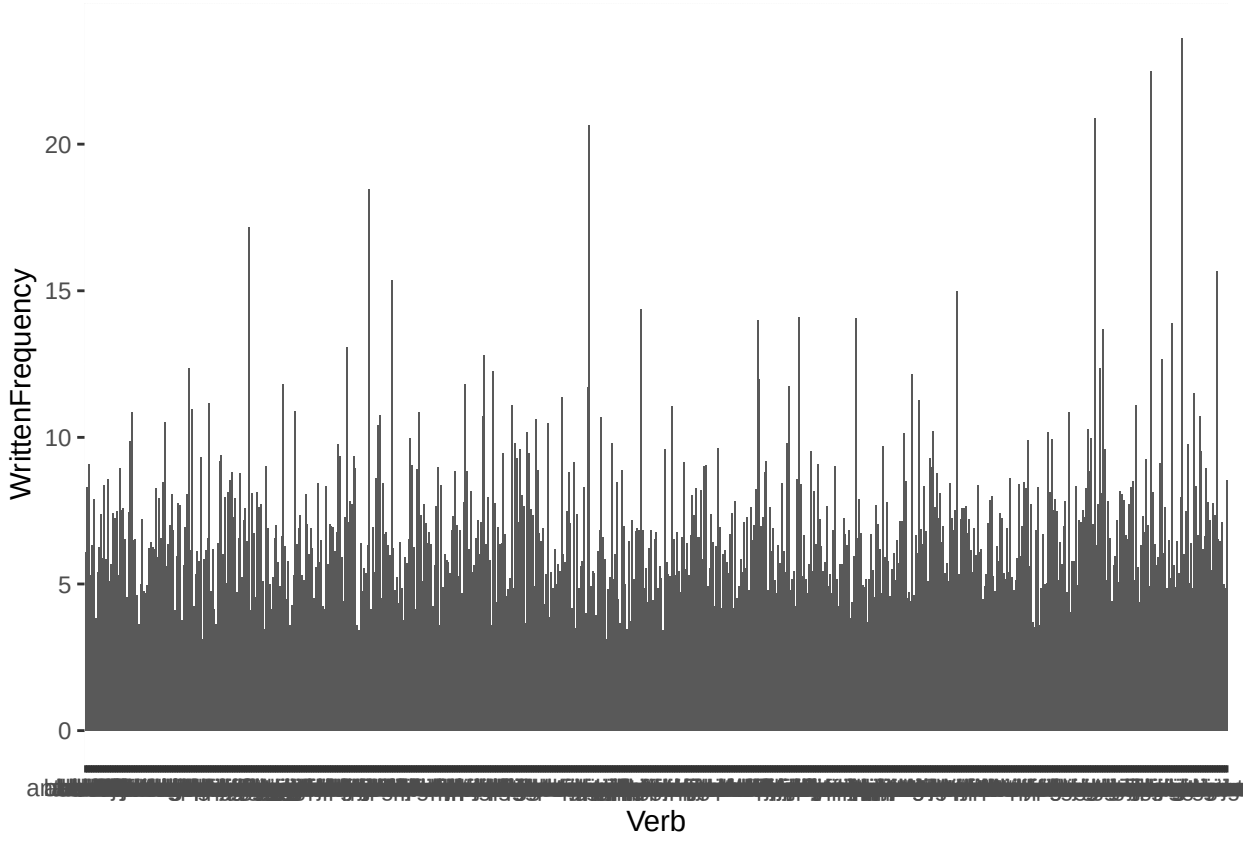
Lo siguiente a observar en el código invocado es el argumento `mapping`. Cada función *geom* toma un argumento `mapping` que define la manera en la que nuestras variables son representadas por diferentes parámetros visuales. Los *geoms* siempre toman un *aesthetic mapping* creado por `aes()` como el valor para `mapping`. `aes()` especifica las propiedades estéticas del gráfico, en este caso proyectamos la variable de nos interesaba, el auxiliar, sobre el argumento `x` de `aes()`. Hay más parámetros visuales que pueden representar la misma información, sin embargo, y en general vamos a tratar de ser visualmente redundantes en la generación de nuestros gráficos, lo cual implica que vamos a usar más de un único parámetro para representar una misma variable.

```
ggplot(tbl_regularity) +
    geom_bar(aes(x = Auxiliary, fill = Auxiliary))
```



Si prestamos atención, vamos a notar que en ningún momento especificamos la altura de las barras. El propio `geom_bar()` computa la cantidad de ocurrencias de cada nivel de la variable y lo proyecta sobre el eje y del gráfico. Nosotres podríamos, sin embargo, querer visualizar algún valor asociado diferente de la cantidad de ocurrencias del valor de una variable, o podríamos encontrar que tenemos una columna que ya da cuenta de la frecuencia de una variable, como es el caso de la columna *WrittenFrequency* de nuestro dataset. Para eso necesitamos usar `geom_col()` en lugar de `geom_bar()` y especificar la variable que se proyecta sobre el eje y.

```
ggplot(tbl_regularity) +
    geom_col(aes(x = Verb, y = WrittenFrequency))
```



Dado que el dataset contiene la información de alrededor de 700 verbos, resulta ilegible en estas condiciones, así que vamos a tomar sólo los diez verbos más frecuentes para graficar.

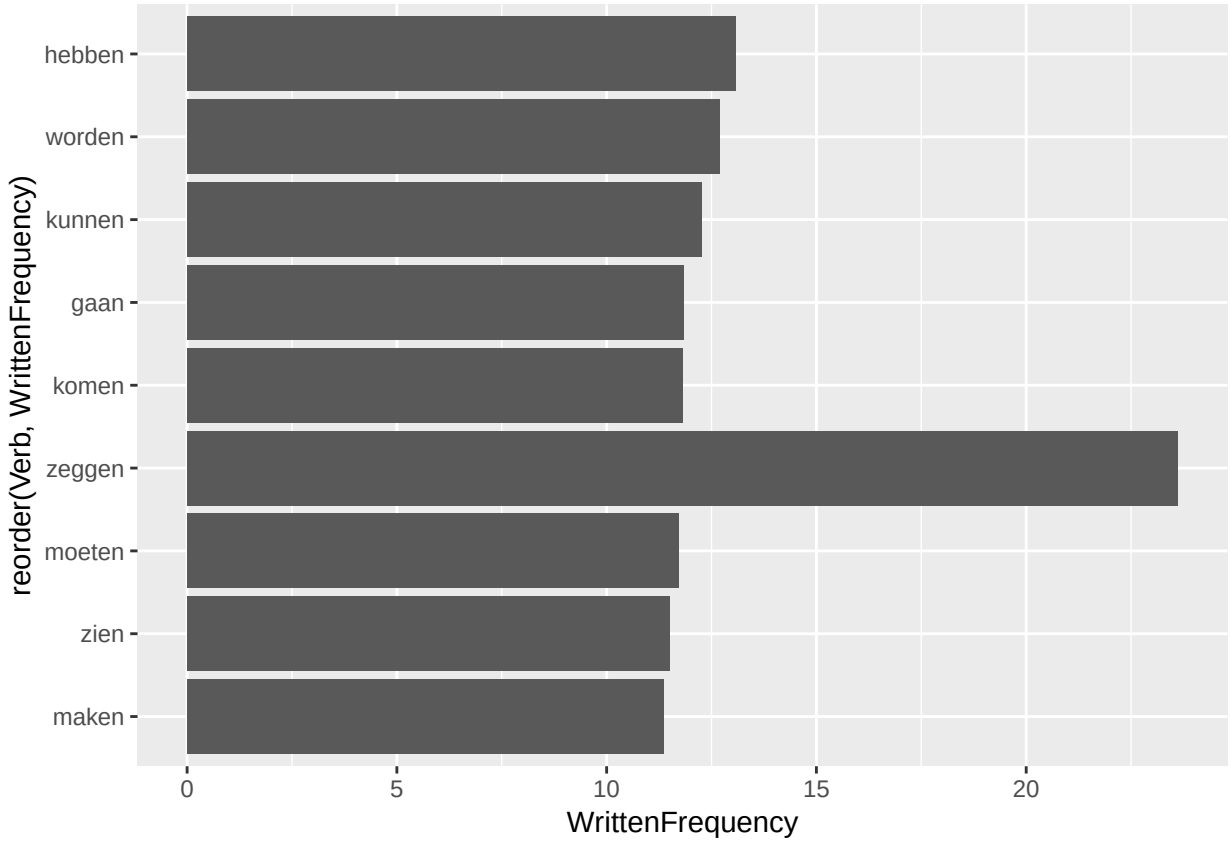
```
mas_frec <- tbl_regularity %>%
    top_n(10, WrittenFrequency)

mas_frec
```

```
## # A tibble: 10 x 13
##    Verb  WrittenFrequency FamilySize LengthInLetters VerbalSynsets
##    <fct>             <dbl>      <dbl>           <int>         <int>
##  1 word~              12.7       2.71               4             2
##  2 zien               11.5       4.50               3             7
##  3 hebb~              13.1       3.81               3             5
##  4 zegg~              11.8       3.83               3             3
##  5 zegg~              11.8       3.83               3             3
##  6 gaan               11.8       4.43               2            15
##  7 maken              11.4       5.08               4             4
##  8 kunn~              12.3       2.64               3             4
##  9 komen              11.8       5.12               3            12
## 10 moet~              11.7       0                  4             5
## # ... with 8 more variables: MeanBigramFrequency <dbl>, NcountStem <int>,
## #   Regularity <fct>, InflectionalEntropy <dbl>, Auxiliary <fct>,
## #   Valency <int>, NVratio <dbl>, WrittenSpokenRatio <dbl>
```

Además, si ponemos los verbos de manera longitudinal sobre el eje x es probable que se superpongan y no podamos leerlos, con lo cual vamos a emplear la función `coord_flip` para rotar el gráfico una vez definidas las proyecciones. También es necesario indicar que queremos que los verbos aparezcan ordenados del más al

menos frecuente.

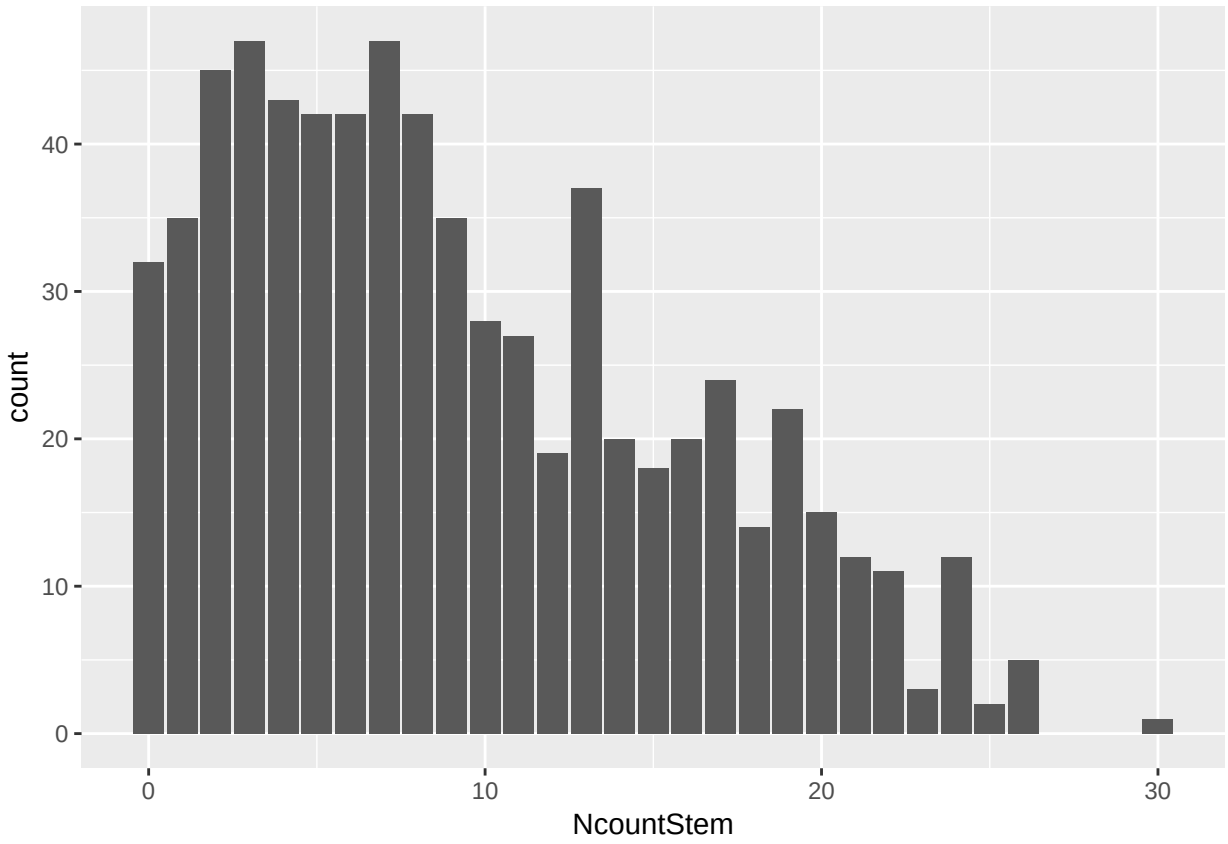
```
ggplot(mas_frec) +
    geom_col(aes(x = reorder(Verb, WrittenFrequency), y = WrittenFrequency)) +
    coord_flip()
```



¿Qué pasó? *zeggen* aparece muy por debajo de lo que debería estar y hay sólo 9 verbos en vez de 10. Si revisamos `mas_frec` vamos a ver que *zeggen* aparece dos veces, pero con distintos valores en la columna de *VerbalSynsets* (y potencialmente en alguna otra). Un plot que parece completamente anómalo puede ser útil para encontrar anomalías en los propios datos de origen.

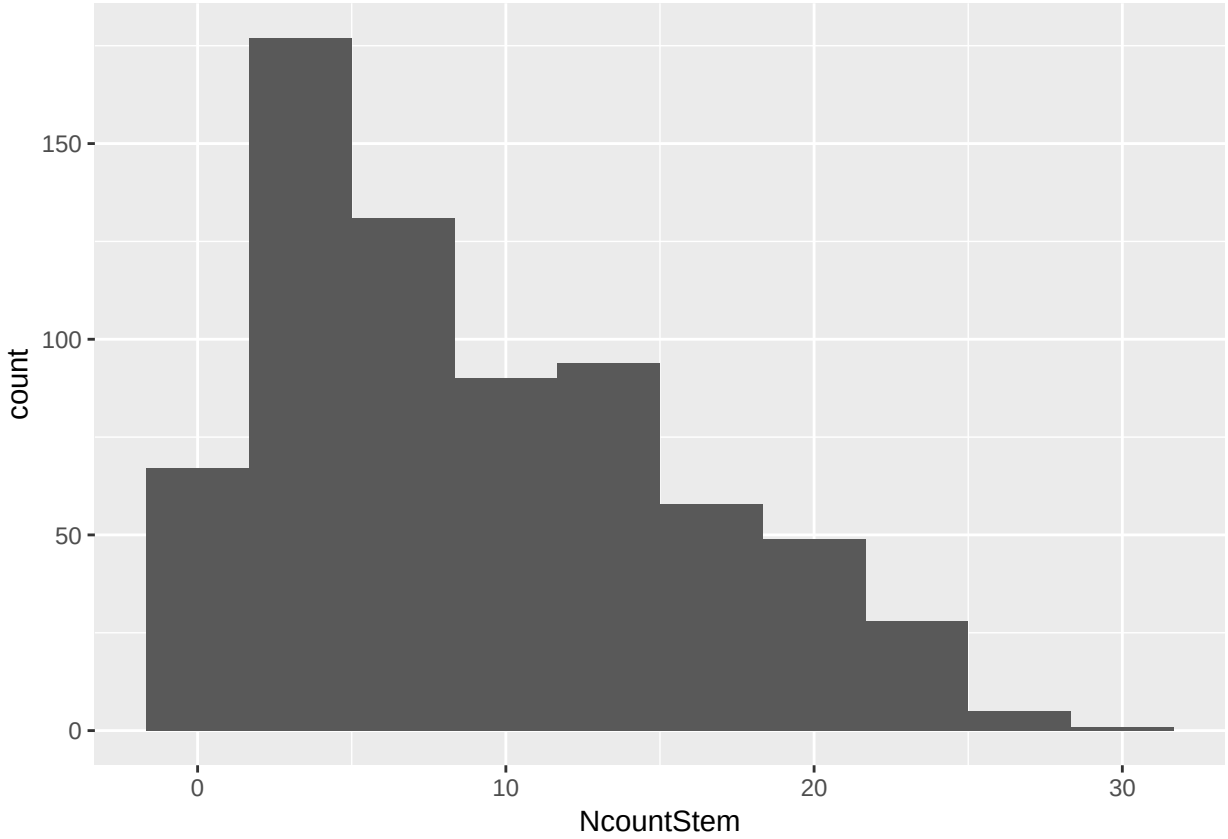
Volvamos a `geom_bar`. Tratemos de ver la cantidad de verbos con raíces del mismo largo.

```
ggplot(tbl_regularity) +
    geom_bar(aes(x = NcountStem))
```



Podemos ver que el largo de las raíces va de 0 a 30 caracteres, pero podríamos tratar de agrupar esos valores. Para eso se vuelve necesario usar otro tipo de visualización, con su correspondiente `geom`.

```
ggplot(tbl_regularity) +
    geom_histogram(aes(x = NcountStem), bins = 10)
```



Un histograma supone la división de un rango de valores en varias porciones del mismo largo. En este caso, partimos el rango 0-30 en diez porciones de 3 (.033). Mientras que en la primera visualización encontrábamos picos en los valores 3 y 7, al agregar la información en secciones encontramos que la gran mayoría de los verbos posee raíces de 3, 4, o 5 letras. Los histogramas se vuelven más útiles a medida que el rango a dividir aumenta. Cuando se poseen observaciones en el rango de 0 a 200, por ejemplo, la visualización por columnas de ancho 1 se vuelve impracticable.

¡Más en la próxima reunión!

Bibliografía

Wickham, Hadley, y Garrett Grolemund. 2017. «Data Visualization with ggplot2». En *R for Data Science*. Sebastopol: O'Reilly.