

# Guia para el curso interno de R

José Julián Mendoza

## Introducción a R para manejo de bases de datos y operaciones geográficas

R es un lenguaje distribuído gratuitamente para realizar estadística y análisis computacional. Gracias a su penetración y su componente de código abierto, hoy podemos utilizarlo para realizar análisis geoespaciales utilizados en el estudio del transporte.

### Objetivos

Adquirir nociones básicas de manejo de software de programación con lenguaje R, para el manejo de bases de datos, análisis estadísticos y visualizaciones, para entender y formular modelos de movilidad y transporte.

Complementar el conocimiento y fomentar el uso de herramientas especializadas que permitan a los colaboradores desarrollar e implementar procesos y metodologías eficientes, reproducibles y mantener un mejor control de los mismos.

### Contenido del curso

Sesión	Temas	Contenido
1	1, 2 y 3	Instalación, introducción y objetos
2	4	Funciones intermedias
3	5	Tidyverse
4	6	Importación y Union de datos
5	7	Visualización de datos
8	8	Limpieza de datos
8	9	Análisis Exploratorio de datos (EDA)
9	10	Estadística en R
10	11.1	Modelos de regresion
11	11.2	Modelos de regresion
12	12.1	Análisis espacial: visualización y analisis
13	12.2	Análisis espacial: estadística y mapas interactivos
14	13.1	Geocomputacion: introducción y zonificacion
15	13.2	Geocomputacion: lineas de deseo y operaciones de ruteo

## ¿Cómo usar este documento?

Esta guía está diseñada para acompañar el curso y servir de referencia para futuras consultas de los temas, conceptos, ejemplos y ejercicios contenidos en el mismo.

Se encuentra dividida en tres partes, que corresponden a cada uno de los niveles impartidos. Durante las semanas de importación del curso, es recomendable seguir la guía y de ser posible, revisar los temas de las próximas sesiones. La guía se complementa con las presentaciones que serán compartidas al inicio del curso, así como las sesiones impartidas y sus respectivas grabaciones.

El curso se encuentra dividido en tres niveles: Básico, Intermedio y Avanzado, que contienen los temas principales para avanzar de manera progresiva en el manejo y uso del lenguaje R. Los ejemplos y ejercicios están compuestos por actividades comunes que utilizan las bases de datos incluidas en la paquetería de R, así como actividades con enfoque en el procesamiento, análisis y presentación de información relativa al transporte.

El formato del documento incluye las librerías resaltadas en fuente cursiva, las funciones en negritas, mientras que el código se encuentra resaltado en el formato que utiliza R (los colores pueden cambiar de acuerdo a cada sistema).

Al ser un lenguaje de código abierto, R es totalmente dependiente de la comunidad de usuarios y desarrolladores para mantenerse actualizado y seguir evolucionando, por lo que es importante conocer algunos de los principales recursos utilizados en el desarrollo del trabajo con R.

CRAN (Comprehensive R Archive Network) es el repositorio principal para los paquetes de R. La paquetería hospedada en CRAN cumple con ciertas políticas técnicas y de calidad, incluyendo que se mantengan actualizados y que la documentación que describe sus funciones sea adecuada y suficiente. Lo anterior nos sirve por que los paquetes albergados en CRAN normalmente tienen buen soporte y participación de la comunidad.

<https://cran.r-project.org>

Algunos paquetes en desarrollo no han sido ingresados a CRAN, sin embargo, pueden estar disponibles en versiones beta directamente con el desarrollador o en otras plataformas.

La plataforma más conocida para el desarrollo de software abierto es Github. En ella se albergan la mayoría de paquetes en desarrollo y también mantiene una participación colaborativa importante, por lo que en muchos casos no existen riesgos de instalar alguno de ellos. Esto es importante por que, aunque en este curso no se utiliza paquetería fuera de CRAN, muchas de las librerías importantes para el estudio de redes de transporte si lo están.

Por su parte, en Github es posible crear una cuenta personal y mantener los proyectos de trabajo en repositorios privados, que permiten un mejor control de versiones y de la evolución del flujo de trabajo.

<https://github.com>

La plataforma más importante para resolver dudas y preguntas, es Stackoverflow. Esta plataforma contiene respuestas tanto comunes como avanzadas, que pueden ayudar para resolver problemas de manera eficiente. Generalmente, una búsqueda en Google de la tarea que se quiere realizar o del error arrojado en R.

<https://stackoverflow.com>

Finalmente, cualquier pregunta relacionada con los temas vistos en el curso puede ser consultada conmigo (Julián).

# Temario

<b>1 Instalación</b>	<b>5</b>
<b>2 Introducción a R</b>	<b>5</b>
2.1 La interfaz de RStudio . . . . .	5
2.2 Librerías, instalación y carga . . . . .	6
2.3 Aritmética básica . . . . .	6
2.4 Declaración de variables . . . . .	7
2.5 Tipos de variables . . . . .	8
<b>3 Objetos</b>	<b>10</b>
3.1 Vectores . . . . .	10
3.1.1 Repaso sobre asignación . . . . .	10
3.1.2 Operaciones en vectores . . . . .	10
3.1.3 Sucesiones . . . . .	11
3.1.4 Índices con vectores . . . . .	13
3.2 Matrices . . . . .	14
3.2.1 Creación de una matriz . . . . .	14
3.2.2 Selección en matrices . . . . .	16
3.3 Factores . . . . .	17
3.3.1 ¿Qué es un factor? . . . . .	17
3.3.2 Conversión a factores . . . . .	17
3.4 Data Frames . . . . .	17
3.4.1 Concepto de Data Frames . . . . .	17
3.4.2 Caso de estudio: Censo de población urbana INEGI . . . . .	18
3.4.3 Introducción a manejo de datos . . . . .	19
3.5 Listas . . . . .	21
3.5.1 Concepto de lista . . . . .	21
3.5.2 Creación de listas . . . . .	21
<b>4 Funciones intermedias</b>	<b>24</b>
4.1 Condicionales y control de flujo . . . . .	24
4.1.1 Operadores relacionales . . . . .	24
4.1.2 Operadores lógicos . . . . .	26
4.1.3 Condicionales . . . . .	27
4.2 Bucles (loops) . . . . .	28
4.2.1 Bucle <b>while</b> . . . . .	28

4.2.2	Bucle <b>for</b>	29
4.3	Funciones	30
4.3.1	Concepto de funciones	30
4.3.2	Creación de funciones	31
4.4	Familia “apply”	32
4.4.1	Función <b>lapply()</b>	32
4.4.2	Función <b>sapply()</b>	33
4.5	Utilidades	34
4.5.1	Utilidades matemáticas	34
4.5.2	Expresiones regulares (regex)	36
4.5.3	Manejo de fechas	37
<b>5</b>	<b>Tidyverse</b>	<b>40</b>
5.1	Manejo de datos	40
5.1.1	Datos	40
5.1.2	Función <b>filter()</b>	41
5.1.3	Función <b>arrange()</b>	41
5.1.4	Función <b>mutate()</b>	42
5.2	Visualización	45
5.3	Agrupación y síntesis	47
5.4	Tipos de visualización	48
<b>6</b>	<b>Importación y Unión de datos</b>	<b>54</b>
6.1	Importación de datos	54
6.1.1	Archivos de excel	54
6.2	Unión interna	56
6.3	Uniones izquierda y derecha	58
6.4	Otras uniones	58
<b>7</b>	<b>Visualización de datos</b>	<b>59</b>
7.1	Gramática de los gráficos	59
7.2	Estéticas	60
7.3	Geometrías	64
7.4	Temas	74
<b>8</b>	<b>Ejercicios finales</b>	<b>79</b>

# 1 Instalación

R es un lenguaje distribuído gratuitamente para realizar estadística y análisis computacional. Gracias a su penetración y su componente de código abierto, hoy podemos utilizarlo para realizar análisis geoespaciales.

Para su instalación y el mejor uso, tenemos que instalar el compilador, así como la interfaz de usuario más popular RStudio.

Compilador:

<https://cran.r-project.org>

RStudio

<https://www.rstudio.com/products/rstudio/download/>

## 2 Introducción a R

En esta sección estudiamos los principios de R y su funcionamiento básico, cómo se puede utilizar la consola como una calculadora y como asignar variables. También veremos los principales tipos de variables o datos, sus usos y diversas formas de modificarlas.

### 2.1 La interfaz de RStudio

RStudio es un Entorno de Desarrollo Integrado (IDE: Integrated Development Environment), es decir, la interfaz entre R y el usuario. Está diseñado para contener varias herramientas simultáneas que un instancia de R no brinda por si sola.

La interfaz se compone principalmente de 4 paneles, el panel de SOURCE (Fuente) o editor de script, la CONSOLA, el WORKSPACE y el panel auxiliar, donde encontramos los archivos, visualizaciones, lista de librerías o paquetes, y el panel de ayuda. Durante el curso veremos más a fondo el uso del panel auxiliar.

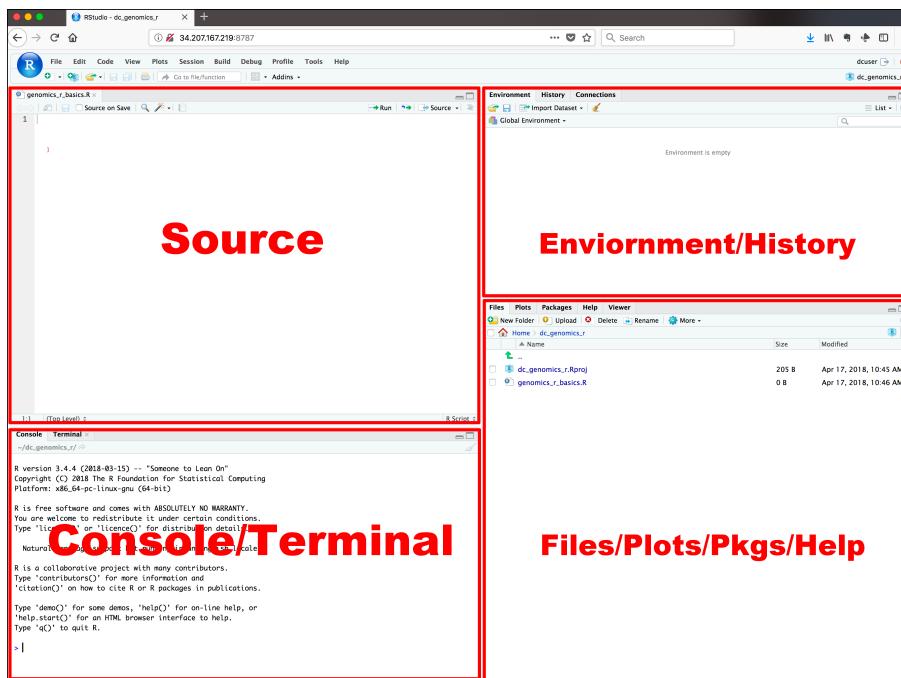


Fig. 1: Ventana típica de 4 paneles

- Source: Aquí escribimos el código
- Workspace: En el workspace tenemos dos pestañas principales, “Environment” o “Entorno” y “History” o “Historial” (pueden aparecer otras pestañas de acuerdo con su versión de R o sus sistema operativo).
  - En la pestaña Environment aparecerán las variables que declaremos y los objetos que vayamos creando
  - En la pestaña History veremos todas las instrucciones de programación que hemos corrido hasta el momento
- Consola: En la consola vemos además de las instrucciones de programación, los errores y advertencias (warnings) que resultan de correr un código. Además, podemos utilizar la consola para correr código que no necesitamos mantener en el flujo de trabajo, como la instalación de librerías, consultar los archivos de ayuda o realizar operaciones simples.

## 2.2 Librerías, instalación y carga

El primer paso para trabajar en R, es el entendimiento de la estructura del lenguaje. El lenguaje R consiste en una serie de funciones y comandos que permiten el análisis de diversos tipos de datos. Estas funciones se agrupan en librerías o paquetes (packages), que se desarrollan para resolver casos específicos, por ejemplo, importar y exportar datos de excel con *openxlsx*, manejar y transformar datos con *dplyr*, realizar gráficas y visualizaciones con *ggplot2* o trabajar con datos georeferenciados y archivos geoespaciales con *sf*.

La versión base de R no contiene estas librerías, por lo que hay que “instalarlas” dentro del sistema y cargarlas cada vez que se utilicen. Esto se puede hacer dentro del código con el comando **install.packages()** o dentro de la pestaña Packages.

```
install.packages("ggplot2")
install.packages("dplyr")
```

Las librerías instaladas se mantienen en el sistema, sin embargo, para hacer más eficiente el trabajo en cada proyecto, es necesario cargarlas en cada caso. El inicio de un script nuevo generalmente se verá como sigue:

```
install.packages(c("ggplot2", "dplyr", "tidyverse"))

library(ggplot2)
library(tidyverse)
library(dplyr)
```

Es importante notar que es posible correr la función **install.packages()** para varias librerías a la vez, lo que ahorra líneas de código y tiempo. La función **c()** significa “concatenar”, y veremos más adelante que se utiliza para crear vectores de datos. También es posible ahorrar tiempo de ejecución omitiendo el paso de instalación, recordando que las librerías solo deben instalarse una vez en cada sistema.

Finalmente, la versión base de R contiene varias funciones útiles con las que es posible hacer diversos cálculos y análisis, incluso importar y exportar archivos y generar gráficas. Como es de esperar, estas funciones base tienen limitantes pero son útiles para primeros acercamientos a una serie de datos nueva, sin necesidad de saturar el script. Así mismo, es recomendable solamente cargar las librerías en uso, para evitar saturaciones de memoria y mantener el código lo más limpio y amigable posible para otros usuarios.

## 2.3 Aritmética básica

Las principales funciones base de R, son los operadores matemáticos. Nosotros entendemos el símbolo de **+** como la suma de dos números, sin embargo, dentro del lenguaje de programación, la computadora tiene que

asociar este símbolo a una operación, por lo que también son llamados funciones. En resumen, para R es lo mismo  $5+5$  que **sum(5, 5)**.

Con esto, en su forma más básica, R es capaz de funcionar como una calculadora, para esto, podemos utilizar la consola, si requerimos hacer cálculos rápidos, o agregar estas operaciones dentro del código. Por ejemplo, podemos calcular sumas, restas, multiplicaciones, divisiones y potencias con los operadores conocidos:

```
# Suma  
5 + 5  
  
## [1] 10  
  
# Resta  
5 - 5  
  
## [1] 0  
  
# Multiplicación  
3 * 5  
  
## [1] 15  
  
# División  
(5 + 5) / 2  
  
## [1] 5  
  
# Potencia  
2^5  
  
## [1] 32
```

## 2.4 Declaración de variables

Las variables son un concepto básico en programación, nos permiten guardar valores (ej. 8) u objetos (ej. el vector `c("ggplot2", "dplyr", "tidyverse")`) en R, para después utilizar el nombre de la variable, acceder a su contenido y realizar operaciones sobre ella.

Para asignar una variable, se utiliza la función `=` o la función `<-`. Por convención (y limpieza del código) la segunda se utiliza de manera generalizada y en solo en ciertos casos se utiliza el signo de `=` (por ejemplo, al asignar variables dentro de una función).

Se pueden asignar variables de diferentes tipos y nombrarlas de casi cualquier forma, de nuevo, la convención dice que es mejor colocar nombres descriptivos y separar las palabras con un punto o un guión bajo.

```
# Buena práctica  
paquetes_lista <- c("ggplot2", "dplyr", "tidyverse")  
calificaciones_grupo_A <- c(8, 6, 7, 5, 10)  
  
# Mala práctica  
pkt = c("ggplot2", "dplyr", "tidyverse")  
calA = c(8, 6, 7, 5, 10)
```

Como se puede observar, las variables declaradas se añaden al entorno, y podemos acceder a ellas dando click, escribiendo su nombre en la consola o con la función **print()**.

```
print(paquetes_lista)

## [1] "ggplot2"    "dplyr"      "tidyverse"
```

Una vez que tenemos la definición de variables, se pueden hacer operaciones con ellas, un ejemplo simple es si asignamos valores únicos a dos variables diferentes.

```
manzanas <- 5
peras <- 4

manzanas + peras

## [1] 9
```

El resultado también puede asignarse a otra variable, como sigue:

```
frutas <- manzanas + peras
print(frutas)
```

```
## [1] 9
```

Pero no siempre podemos sumar peras con manzanas.

## 2.5 Tipos de variables

En R, se trabaja con diversos tipos de variables, algunos de los tipos fundamentales son:

- Valores decimales como 8.8 se llaman **numerics** (numéricos)
- Valores enteros como 8 se llaman **integers** (enteros)
- Valores de verdadero o falso se llaman **logicals** (lógicos)
- Registros de texto se llaman **characters** (caractéres)

Como dijimos, no podemos sumar peras con manzanas

```
manzanas <- 5
peras <- "cuatro"
manzanas + peras

## Error in manzanas + peras: non-numeric argument to binary operator
```

El error indica que R no puede operar un valor numérico con un valor de texto, lo cual tiene sentido y nos ayuda a buscar alternativas dentro de nuestro código, por ejemplo, en una base de datos, los valores numéricos pueden ser interpretados como texto, y podríamos estar cometiendo errores al no convertirlos primero.

```
manzanas <- 5
peras <- "4"
manzanas + peras
```

```
## Error in manzanas + peras: non-numeric argument to binary operator
```

Para conocer con qué tipo de variable estamos trabajando, la versión base de R cuenta con la función `class()`.

```
class(manzanas)
```

```
## [1] "numeric"
```

```
class(peras)
```

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

## 3 Objetos

En R trabajamos con valores y estructuras de datos que se almacenan como objetos. Un objeto es un valor o una serie de valores almacenados en diferentes composiciones que sirven para ejecutar funciones sobre ellos.

En esta sección trabajaremos con las estructuras de datos más versátiles y con mayor funcionalidad de R, veremos las mejores prácticas para utilizarlas, así como los usos más comunes de cada una.

### 3.1 Vectores

Una de las estructuras de datos más simples, y la más común en ser utilizada por su versatilidad es el vector. Los vectores en R son colecciones ordenadas de números. Regresando a la definición de los objetos, un valor único se considera un vector de longitud uno.

#### 3.1.1 Repaso sobre asignación

Para entender mejor la asignación de objetos, dedicamos unas líneas para extender los conceptos detrás de ella. Podemos observar que existen diversos métodos para asignar un objeto a una variable, como se mencionó anteriormente, el estándar más aceptado por convención es el operador `<-` (signo de menor que seguido por un guión medio). El signo de igual (`=`) funciona pero es utilizado en casos específicos.

La función `assign()` también sirve para asignar objetos a variables, no es tan comúnmente utilizada en vectores pero es útil conocerla, ya que para otro tipo de objetos es más común, e incluso cuando se asigna un vector dentro de un bucle o una función.

Es importante recordar la sintaxis de asignación de vectores, compuesta por la variable, el operador o función de asignación y la función `c()` que concatena los valores.

```
# Vector de tiempos de recorrido en minutos

tiempos <- c(5, 10, 15, 20)
assign("tiempos", c(5, 10, 15, 20))
```

Al asignar valores a la misma variable, ésta se sobreescribe.

El operador `<-` también asigna si se utiliza el revés, manteniendo la lógica de la expresión, es decir, primero el contenido del objeto y después la variable.

```
# Vector de distancias en kilómetros

c(8, 15, 20, 40) -> distancias
```

#### 3.1.2 Operaciones en vectores

Los operadores aritméticos vistos anteriormente funcionan también en los vectores de la misma manera en como operarían un vector normal en matemáticas. Por lo que es posible realizar cualquier operación entre ellos. Incluso podemos operar sobre vectores de un elemento.

```
# Cálculo de velocidades en kilómetros por hora

minutos <- 60
velocidades <- distancias / tiempos * minutos

print(velocidades)
```

```
## [1] 96 90 80 120
```

También es posible utilizar otras funciones matemáticas en forma de texto, como por ejemplo log, exp, sin, cos, tan y sqrt. Por otra parte, muchas de las funciones dentro de la paquetería de R están diseñadas para trabajar sobre vectores, por ejemplo, las funciones **min()** y **max()** extraen los valores mínimos y máximos de un vector o una serie de vectores, la función **mean()** devuelve la media de los valores en el vector, **median()** devuelve la mediana, etc.

```
max(velocidades)
```

```
## [1] 120
```

```
min(velocidades)
```

```
## [1] 80
```

```
mean(velocidades)
```

```
## [1] 96.5
```

```
median(velocidades)
```

```
## [1] 93
```

### 3.1.3 Sucesiones

En R se pueden generar sucesiones numéricas con diferentes funciones, por ejemplo, **1:5** conforma el vector **c(1,2,3,4,5)**. R interpreta los “dos puntos” como un operador. La forma **5:1** construye una sucesión descendente. Para generar sucesiones más complejas, podemos utilizar la función **seq()** (secuencia), que cuenta con 5 argumentos.

Vale la pena detenerse en los argumentos de una función y en la función de ayuda **?**. Al escribir la la función de ayuda como sufijo de cualquier función en la consola, R regresa el archivo de ayuda de la función, que se compone de una descripción de la función y sus argumentos. En este caso, si escribimos **?seq()**, en la pestaña “Help” veremos el documento descriptivo.

```
?seq()
```

seq {base} R Documentation

## Sequence Generation

### Description

Generate regular sequences. `seq` is a standard generic with a default method. `seq.int` is a primitive which can be much faster but has a few restrictions. `seq_along` and `seq_len` are very fast primitives for two common cases.

### Usage

```
seq(...)

## Default S3 method:
seq(from = 1, to = 1, by = ((to - from)/(length.out - 1)),
    length.out = NULL, along.with = NULL, ...)

seq.int(from, to, by, length.out, along.with, ...)

seq_along(along.with)
seq_len(length.out)
```

### Arguments

...	arguments passed to or from methods.
from, to	the starting and (maximal) end values of the sequence. Of length 1 unless just <code>from</code> is supplied as an unnamed argument.
by	number: increment of the sequence.
length.out	desired length of the sequence. A non-negative number, which for <code>seq</code> and <code>seq.int</code> will be rounded up if fractional.
along.with	take the length from the length of this argument.

### Details

Numerical inputs should all be `finite` (that is, not infinite, `NaN` or `NA`).

The interpretation of the unnamed arguments of `seq` and `seq.int` is *not* standard, and it is recommended always to name the arguments when programming.

`seq` is generic, and only the default method is described here. Note that it dispatches on the class of the `first` argument irrespective of argument names. This can have unintended consequences if it is called with just one argument intending this to be taken as `along.with`: it is much better to use `seq_along` in that case.

`seq.int` is an [internal generic](#) which dispatches on methods for "seq" based on the class of the first supplied argument (before argument matching).

Fig. 2: Documento de ayuda de la función `seq()`

Los argumentos de la función `seq()` son “from”, “to”, “by”, “length.out” y “along.with”. Una función se compone de argumentos obligatorios y argumentos opcionales, además de que los argumentos se dan por posición, es decir, R entiende y asigna los argumentos según sean dados, por lo que `seq(1,5)`, `seq(from = 1, to = 5)`, son equivalentes a `1:5`.

```
seq(1, 5)
seq(from = 1, to = 5)
1:5
```

La función `seq()` puede operar con argumentos incompletos porque incluye valores predeterminados para los argumentos no obligatorios, por ejemplo, `by = 1`. El argumento `by` determina el “paso” de la sucesión, que por defecto es “de uno en uno”, pero se puede modificar para hacerlo “de dos en dos” o en cualquier escala deseada.

```
seq(from = 1, to = 5, by = 0.2)
```

```
## [1] 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0 3.2 3.4 3.6 3.8 4.0 4.2 4.4 4.6
## [20] 4.8 5.0
```

```
tiempos <- seq(5, 20, 5)
print(tiempos)
```

```
## [1] 5 10 15 20
```

### 3.1.4 Índices con vectores

Como vimos, los vectores pueden ser de diferentes clases, a su vez, pueden ser utilizados para guardar información, operar sobre ellos e incluso operar con ellos, es decir, se puede utilizar un vector lógico o de índices para realizar selecciones sobre otro vector.

Por ejemplo, si tenemos un vector de tiempos, y queremos filtrar solamente los valores mayores a 15 minutos, podemos hacerlo de la siguiente forma:

```
tiempos <- seq(1, 30, 2)
tiempos_select <- tiempos > 5
tiempos_select
```

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

Lo que nos da un vector lógico, en donde solamente se imprimen como verdaderos los valores en donde el tiempo es mayor a 5 minutos.

El operador [ ] (corchetes) le dice a R que busque el contenido del mismo dentro de un vector (o base de datos), por lo que el siguiente código devuelve solamente los elementos que aparecen como “TRUE” en el vector índice.

```
tiempos[tiempos_select]
```

```
## [1] 7 9 11 13 15 17 19 21 23 25 27 29
```

Es posible anidar funciones dentro del vector de índices, por lo que lo anterior se puede realizar en una sola línea, tal que:

```
tiempos[seq(1,30,2)>5]
```

```
## [1] 7 9 11 13 15 17 19 21 23 25 27 29
```

Además de vectores lógicos, los índices pueden ser números naturales positivos, en donde estos números corresponden a la posición dentro del vector a filtrar. Así, **tiempos[3]** nos devolverá el tercer elemento del vector **tiempos**, **tiempos[1:5]** nos devolverá los primeros 5 elementos y **tiempos[c(1,5)]** imprime el primer y el quinto elemento. Cabe destacar la diferencia en este último caso, en donde el vector de índices debe proporcionarse con la función **c()**.

```
tiempos[3]
```

```
## [1] 5
```

```
tiempos[1:5]
```

```
## [1] 1 3 5 7 9
```

```
tiempos[c(1,5)]
```

```
## [1] 1 9
```

Por su parte, un vector de números naturales negativos devuelve todos los elementos **excepto** los marcados en el índice.

```
tiempos <- seq(5,20,5)
tiempos[-3]
```

```
## [1] 5 10 20
```

```
tiempos[-(1:3)]
```

```
## [1] 20
```

```
tiempos[-c(2,4)]
```

```
## [1] 5 15
```

Es posible utilizar un vector de caractéres o cadena de caracteres como índice, siempre que el vector original contenga el atributo **names** que sirve para identificar sus componentes.

```
names(tiempos) <- c("auto", "tp", "bici", "peatón")
tiempos[c("auto", "bici")]
```

```
## auto bici
##      5    15
```

Lo anterior puede ayudar a realizar la selección ya que es más fácil recordar los nombres que los índices numéricos, que es una estrategia que se utiliza mucho más para realizar selecciones y filtros en data frames (hojas o bases de datos).

## 3.2 Matrices

En R, las matrices son colecciones de elementos que tienen el mismo tipo o clase de datos (numérico, carácter o lógico) que tienen un arreglo de filas y columnas fijo. Una matriz puede construirse utilizando la función **matrix()**.

En los siguientes ejemplos, utilizaremos lo visto hasta el momento para analizar los elementos como una matriz. Podemos empezar por retomar los dos vectores creados de tiempos y distancias, pero esta vez de manera horizontal. Es decir, en lugar de definir vectores de tiempos y distancias, definiremos el vector de tiempo y distancia para cada modo. Cabe mencionar que se puede trabajar de las dos formas.

### 3.2.1 Creación de una matriz

Utilizaremos un ejemplo simple, un recorrido de la oficina de Cal y Mayor al Parque de los Venados.

```
# Vectores de modo de transporte con la forma (tiempo, distancia)
auto <- c(9, 3.2)
peaton <- c(49, 3.9)
bici <- c(12, 3.2)
```

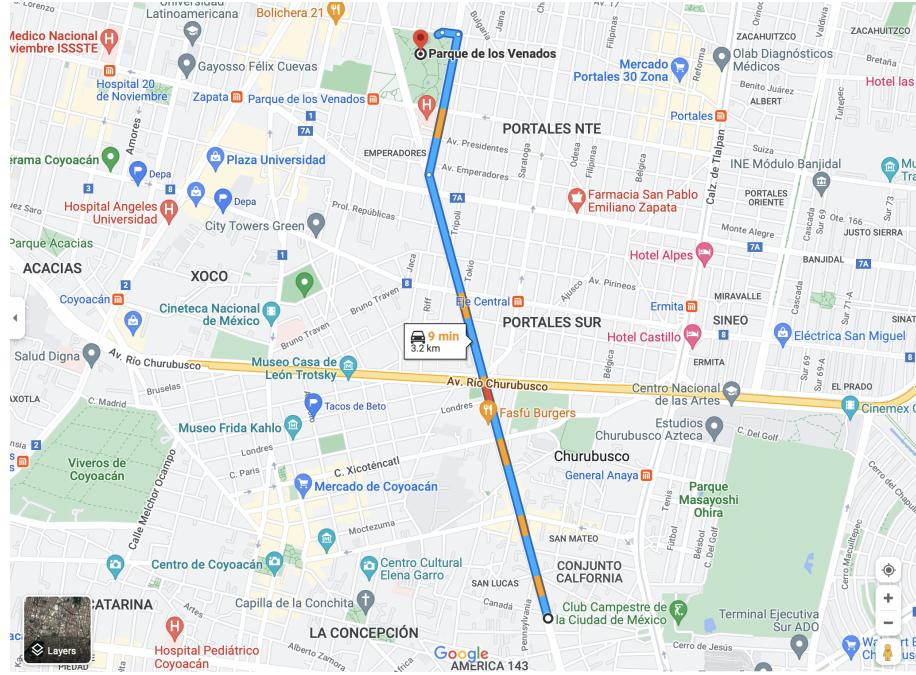


Fig. 3: Recorrido ejemplo en google maps

Después podemos concatenar los vectores, y construir la matriz. El argumento **byrow** le indica a la función que le dictaremos los datos de manera horizontal, mientras que el argumento **nrow** le indica el número de filas.

```
modos <- c(auto, peaton, bici)
modos

## [1] 9.0 3.2 49.0 3.9 12.0 3.2

modos_matrix <- matrix(modos, byrow = TRUE, nrow = 3)

modos_matrix

##      [,1] [,2]
## [1,]    9   3.2
## [2,]   49   3.9
## [3,]   12   3.2
```

Como vemos, generamos una tabla, que es una matriz y no un data frame debido a que todos sus elementos son del mismo tipo de variable (numérico en este caso). También es posible tener una matriz lógica o de caractéres. La matriz generada no contiene identificadores de filas o columnas, pero eso podemos solucionarlo fácilmente con las funciones **rownames()** y **colnames()**.

```
modos <- c("auto", "peaton", "bici")
metrica <- c("tiempo", "distancia")

rownames(modos_matrix) <- modos
colnames(modos_matrix) <- metrica
```

```
modos_matrix

##      tiempo distancia
## auto      9      3.2
## peaton   49      3.9
## bici     12      3.2
```

### 3.2.2 Selección en matrices

Podemos extraer los valores de la matriz tanto por su posición numérica como por los nombres de filas o columnas.

```
modos_matrix[, "tiempo"]
```

```
##    auto peaton  bici
##    9     49     12
```

```
modos_matrix["auto",]
```

```
##      tiempo distancia
##      9.0      3.2
```

```
modos_matrix[1:3, 2]
```

```
##    auto peaton  bici
##    3.2     3.9     3.2
```

```
modos_matrix[1, 1:2]
```

```
##      tiempo distancia
##      9.0      3.2
```

```
velocidades <- modos_matrix[, "distancia"]/modos_matrix[, "tiempo"] * 60
velocidades
```

```
##    auto  peaton    bici
## 21.33333 4.77551 16.00000
```

Podemos calcular un vector de velocidades utilizando estos índices, que después podemos añadir a la matriz por medio de la función **cbind()**. Las funciones **rbind()** y **cbind()** se utilizan para agregar datos a matrices y data frames. La primera anexa filas y la segunda columnas.

```
modos_matrix_total <- cbind(modos_matrix, velocidades)
modos_matrix_total
```

```
##      tiempo distancia velocidades
## auto      9      3.2    21.33333
## peaton   49      3.9    4.77551
## bici     12      3.2    16.00000
```

### 3.3 Factores

La información puede categorizarse en un número limitado de formas. En R, conviene guardar los datos categóricos como factores. Los factores son importantes en el análisis de datos, tanto para agrupar datos y generar estimaciones sobre ellos, o para realizar visualizaciones más amigables para diversas audiencias.

#### 3.3.1 ¿Qué es un factor?

El término “factor” se refiere a un tipo de datos estadísticos utilizados para guardar variables categóricas. Es importante que R conozca cuales variables son factores.

En nuestro ejemplo, para mantenerlo práctico, podemos agregar un vector de factores que ayuden a describir el viaje realizado. Los factores son claramente más útiles para colecciones de datos más amplias, pero en este documento nos concentraremos más en la explicación básica de las funciones del programa. Más referencias pueden encontrarse en las presentaciones y ejercicios del curso.

#### 3.3.2 Conversión a factores

Con lo anterior, podemos anexar el vector de factores con el día en que se realizó el viaje. Generamos un vector de caractéres, que después convertimos en un vector de factores por medio de la función **factor()**. Es importante mencionar que uno de los atributos de un vector de factores son sus niveles, que definen el orden de los mismos. Si bien los factores pueden tener un orden, no tienen jerarquía, es decir, no es posible determinar si un martes y más o menos que un miércoles, pero si es posible determinar que el martes es primero que el miércoles.

```
dia <- c("martes", "martes", "miercoles")
dia_factor <- factor(dia)
```

Como vemos, podemos definir todos los niveles posibles del vector, sin necesidad de que éste contenga todos ellos. También es posible definir los niveles y su orden dentro de la propia función **factor()**.

```
dia_factor <- factor(dia, ordered = TRUE, levels = c("lunes", "martes", "miercoles",
                                                       "jueves", "viernes", "sabado",
                                                       "domingo"))
cbind(modos_matrix_total, dia_factor)
```

```
##      tiempo distancia velocidades dia_factor
## auto        9       3.2    21.33333      2
## peaton     49       3.9     4.77551      2
## bici       12       3.2    16.00000      3
```

Más adelante veremos diversas operaciones utilizando factores, tanto en el análisis de datos como para la generación de visualizaciones gráficas.

### 3.4 Data Frames

#### 3.4.1 Concepto de Data Frames

La mayor parte de las series de datos con las que se trabaja en R están guardados como **dataframes**. Del capítulo de matrices podemos recordar que en ellas, todos los datos que guardamos deben ser del mismo tipo. La serie de datos utilizada para nuestra matriz **modos\_matrix\_total** estaba compuesta solo por

elementos numéricicos. Pero para realizar análisis más profundos en transporte, normalmente tendremos que utilizar una combinación de elementos numéricos, de caractéres y lógicos. Por ejemplo, podemos guardar el modo de transporte o el género de los usuarios como una variable categórica en forma de factores, si el viaje fue realizado de día como una variable lógica, etc.

### 3.4.2 Caso de estudio: Censo de población urbana INEGI

Para este capítulo, utilizaremos el paquete *importinegi*, desarrollado por el Instituto Nacional de Estadística y Geografía (INEGI), que se encuentra en CRAN y presenta funciones muy útiles. Con la función **censo\_poblacion\_urbano()** podemos descargar el censo de población para las AGEBS urbanas, utilizando los argumentos **year** y **estado** limitamos la descarga al año 2010 en la Ciudad de México. Por su parte, la función **head()** nos muestra los primeros 10 registros de la base de datos. Importante mencionar que para R, una fila es llamada una **observación** y una columna una **variable**.

```
library(importinegi)
censo <- censo_poblacion_urbano(year = "2010", estado = "CDMX")

head(censo)
```

```
##   ENT      NOM_ENT MUN      NOM_MUN LOC      NOM_LOC AGEB MZA CONJHAB
## 1 09 Distrito Federal 002 Azcapotzalco 0001 Azcapotzalco 0542 046      1
## 2 09 Distrito Federal 002 Azcapotzalco 0001 Azcapotzalco 0542 047      1
## 3 09 Distrito Federal 002 Azcapotzalco 0001 Azcapotzalco 0542 048      1
## 4 09 Distrito Federal 002 Azcapotzalco 0001 Azcapotzalco 0542 050      1
## 5 09 Distrito Federal 002 Azcapotzalco 0001 Azcapotzalco 0542 049      1
## 6 09 Distrito Federal 002 Azcapotzalco 0001 Azcapotzalco 0330 001      3
##   RECUCALL_ BANQUETA_ GUARNICI_ ARBOLES_ RAMPAS_ ALUMPUB_ SENALIZA_ TELPUB_
## 1      5        5        5        5        5        5        5        5
## 2      5        5        5        5        5        5        5        5
## 3      5        5        5        5        5        5        5        5
## 4      5        5        5        5        5        5        5        5
## 5      5        5        5        5        5        5        5        5
## 6      2        2        2        2        2        1        2        2
##   DRENAJEP_ TRANSCOL_ ACESOPER_ ACESOAUT_ PUESSEMI_ PUESAMBU_ VIVTOT TVIVHAB
## 1      5        5        5        5        5        5       48        1
## 2      5        5        5        5        5        5      120        0
## 3      5        5        5        5        5        5      132        7
## 4      5        5        5        5        5        5      102        1
## 5      5        5        5        5        5        5      144        0
## 6      2        2        1        2        2        2      135     124
##   TVIVPARHAB VPH_DEPTO POBTOT
## 1          0        1      1
## 2          0        0      0
## 3          7        6     17
## 4          0        1      4
## 5          0        0      0
## 6         124        4    435
```

Con la función **str()** realizaremos una revisión previa de la estructura de los datos, para conocer las variables incluidas en la base sus tipos. Con esto podemos definir cuales variables nos son útiles y cuales podemos descartar.

```

str(censo)

## 'data.frame':   63028 obs. of  28 variables:
## $ ENT      : chr  "09" "09" "09" "09" ...
## $ NOM_ENT  : chr  "Distrito Federal" "Distrito Federal" "Distrito Federal" "Distrito Federal" ...
## $ MUN      : chr  "002" "002" "002" "002" ...
## $ NOM_MUN  : chr  "Azcapotzalco" "Azcapotzalco" "Azcapotzalco" "Azcapotzalco" ...
## $ LOC      : chr  "0001" "0001" "0001" "0001" ...
## $ NOM_LOC  : chr  "Azcapotzalco" "Azcapotzalco" "Azcapotzalco" "Azcapotzalco" ...
## $ AGEB     : chr  "0542" "0542" "0542" "0542" ...
## $ MZA      : chr  "046" "047" "048" "050" ...
## $ CONJHAB  : chr  "1" "1" "1" "1" ...
## $ RECUCALL_ : chr  "5" "5" "5" "5" ...
## $ BANQUETA_ : chr  "5" "5" "5" "5" ...
## $ GUARNICI_ : chr  "5" "5" "5" "5" ...
## $ ARBOLES_  : chr  "5" "5" "5" "5" ...
## $ RAMPAS_   : chr  "5" "5" "5" "5" ...
## $ ALUMPUB_  : chr  "5" "5" "5" "5" ...
## $ SENALIZA_ : chr  "5" "5" "5" "5" ...
## $ TELPUB_   : chr  "5" "5" "5" "5" ...
## $ DRENAJEP_ : chr  "5" "5" "5" "5" ...
## $ TRANSCOL_ : chr  "5" "5" "5" "5" ...
## $ ACESOPER_ : chr  "5" "5" "5" "5" ...
## $ ACESOAUT_ : chr  "5" "5" "5" "5" ...
## $ PUESSEMI_ : chr  "5" "5" "5" "5" ...
## $ PUESAMBU_ : chr  "5" "5" "5" "5" ...
## $ VIVTOT    : int  48 120 132 102 144 135 77 50 32 77 ...
## $ TVIVHAB   : int  1 0 7 1 0 124 29 21 11 39 ...
## $ TVIVPARHAB: int  0 0 7 0 0 124 29 21 11 39 ...
## $ VPH_DEPTO : int  1 0 6 1 0 4 28 19 11 39 ...
## $ POBTOT    : int  1 0 17 4 0 435 80 48 27 109 ...
## - attr(*, "data_types")= chr [1:28] "C" "C" "C" "C" ...

```

### 3.4.3 Introducción a manejo de datos

Vemos que descargamos una base de datos con 63,028 observaciones y 28 variables. Para mantener este ejercicio más sencillo, vale la pena adelantar algunas funciones de manejo de datos, especialmente las funciones **filter()** y **select()**. Con **filter()** podemos elegir que observaciones mantener de acuerdo a uno o varios criterios, mientras que con **select()** elegiremos solamente las variables con las que vamos a trabajar. Esto es útil tanto para tener más claridad al trabajar con bases de datos muy grandes, así como para permitir que R utilice menos memoria al procesar los datos. Estas funciones pertenecen al paquete *dplyr* que a su vez es parte de una serie de paquetes llamados *tidyverse*.

Para fines de este ejemplo, filtraremos la primer alcaldía que aparece: Azcapotzalco, y seleccionaremos solamente unas cuantas variables con las que podamos trabajar. La función **head()** maneja un argumento **n**, con el cual podemos elegir el número de elementos a mostrar, como se mencionó, el valor por defecto es 10.

```

library(dplyr)

##
## Attaching package: 'dplyr'

```

```

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union

censo_filtered <- filter(censo, NOM_MUN == "Azcapotzalco")
censo_filtered_select <- select(censo_filtered, NOM_MUN, AGEB, MZA, VIVTOT, POBTOT)

head(censo_filtered_select, n = 5)

##           NOM_MUN AGEB MZA VIVTOT POBTOT
## 1 Azcapotzalco 0542 046     48      1
## 2 Azcapotzalco 0542 047    120      0
## 3 Azcapotzalco 0542 048    132     17
## 4 Azcapotzalco 0542 050    102      4
## 5 Azcapotzalco 0542 049    144      0

```

Los elementos de un data frame pueden ser seleccionados de la misma forma que los elementos de vectores y matrices, es decir, si queremos extraer la quinta observación de la tabla de datos que acabamos de crear, podemos hacerlo de la siguiente manera:

```

censo_filtered_select[5,]

##           NOM_MUN AGEB MZA VIVTOT POBTOT
## 5 Azcapotzalco 0542 049    144      0

```

También pudimos haber realizado la selección utilizando índices en lugar de la función **select()**, pero como podemos ver, puede resultar un poco más complejo si se tienen bases de datos de muchas variables.

```

censo_filtered_select2 <- censo[censo["NOM_MUN"] == "Azcapotzalco",c(4, 7:8, 24, 28)]
head(censo_filtered_select2, n = 5)

##           NOM_MUN AGEB MZA VIVTOT POBTOT
## 1 Azcapotzalco 0542 046     48      1
## 2 Azcapotzalco 0542 047    120      0
## 3 Azcapotzalco 0542 048    132     17
## 4 Azcapotzalco 0542 050    102      4
## 5 Azcapotzalco 0542 049    144      0

```

En este caso, en el índice de columnas le indicamos a R un vector de posiciones de acuerdo con la variable que debe mantener. Lo mismo se puede hacer utilizando los nombres de las variables. Sin embargo, esto requiere de índices anidados y más adelante veremos que el flujo de trabajo más moderno de R se beneficia mucho más de utilizar las funciones **filter()** y **select()**.

Por su parte, tal vez la función más importante a estudiar dentro del manejo de dataframes en R, es el operador **\$**. Cuando las columnas de la base de datos tienen nombres, el operador nos sirve para seleccionarlas en forma de vector, extrayendo todos los valores de una variable. En términos generales, las dos funciones mostradas a continuación producen el mismo resultado, pero **select()** mantiene la estructura de dataframe mientras que **\$** extrae los datos como vector.

```

class(select(censo_filtered_select, POBTOT))

## [1] "data.frame"

class(censo_filtered_select$POBTOT)

## [1] "integer"

```

Finalmente, la función **order()** nos devuelve un vector de posiciones con el orden de los valores. Utilizamos el argumento **decreasing = TRUE** para indicar que el orden debe ser descendente.

```

orden <- order(censo_filtered_select$POBTOT, decreasing = TRUE)
head(orden, n = 5)

```

```

## [1] 2860 371 712 1749 1866

```

Esto quiere decir que el elemento 2,860 es el primero en el orden de valores, para ordenar la base de datos, tenemos que indicarle que estos valores registran ese orden.

```

censo_orden <- censo_filtered_select[orden,]
head(censo_orden, n = 5)

```

```

##           NOM_MUN AGEB MZA VIVTOT POBTOT
## 2860 Azcapotzalco 0222 001     511   1685
## 371  Azcapotzalco 1095 012     470   1646
## 712  Azcapotzalco 0491 009     548   1612
## 1749 Azcapotzalco 0665 002     582   1609
## 1866 Azcapotzalco 0129 034     420   1589

```

## 3.5 Listas

### 3.5.1 Concepto de lista

Las listas son el último tipo de estructura de datos y el más complejo. La principal diferencia con los elementos anteriores es que las listas permiten almacenar series de datos de diferente longitud y de diferentes clases. En una misma lista se pueden almacenar variedades de objetos como matrices, vectores y dataframes, incluso otras listas. Incluso no es obligatorio que estos objetos se relacionen entre sí.

### 3.5.2 Creación de listas

Con los elementos generados hasta ahora, podemos almacenar el vector de velocidades, la matriz de modos y el dataframe del censo en una lista.

```

mi_lista <- list(velocidades, modos_matrix_total, censo_orden)
str(mi_lista)

```

```

## List of 3
## $ : Named num [1:3] 21.33 4.78 16
## ..- attr(*, "names")= chr [1:3] "auto" "peaton" "bici"
## $ : num [1:3, 1:3] 9 49 12 3.2 3.9 ...
## ..- attr(*, "dimnames")=List of 2
## ... $ : chr [1:3] "auto" "peaton" "bici"
## ... $ : chr [1:3] "tiempo" "distancia" "velocidades"
## $ :'data.frame': 2991 obs. of 5 variables:
## ... $ NOM_MUN: chr [1:2991] "Azcapotzalco" "Azcapotzalco" "Azcapotzalco" "Azcapotzalco" ...
## ... $ AGEB : chr [1:2991] "0222" "1095" "0491" "0665" ...
## ... $ MZA : chr [1:2991] "001" "012" "009" "002" ...
## ... $ VIVTOT : int [1:2991] 511 470 548 582 420 438 472 444 424 464 ...
## ... $ POBTOT : int [1:2991] 1685 1646 1612 1609 1589 1458 1453 1420 1376 1351 ...
## ..- attr(*, "data_types")= chr [1:28] "C" "C" "C" "C" ...

```

Se puede generar una lista y asignar nombres a cada elemento, esto es especialmente útil dadas las características de una lista, que como se mencionó, pueden incluir combinaciones de estructuras de datos.

```

mi_lista <- list(vel = velocidades, matrix = modos_matrix_total,
                  censo = censo_orden)
str(mi_lista)

```

```

## List of 3
## $ vel : Named num [1:3] 21.33 4.78 16
## ..- attr(*, "names")= chr [1:3] "auto" "peaton" "bici"
## $ matrix: num [1:3, 1:3] 9 49 12 3.2 3.9 ...
## ..- attr(*, "dimnames")=List of 2
## ... $ : chr [1:3] "auto" "peaton" "bici"
## ... $ : chr [1:3] "tiempo" "distancia" "velocidades"
## $ censo :'data.frame': 2991 obs. of 5 variables:
## ... $ NOM_MUN: chr [1:2991] "Azcapotzalco" "Azcapotzalco" "Azcapotzalco" "Azcapotzalco" ...
## ... $ AGEB : chr [1:2991] "0222" "1095" "0491" "0665" ...
## ... $ MZA : chr [1:2991] "001" "012" "009" "002" ...
## ... $ VIVTOT : int [1:2991] 511 470 548 582 420 438 472 444 424 464 ...
## ... $ POBTOT : int [1:2991] 1685 1646 1612 1609 1589 1458 1453 1420 1376 1351 ...
## ..- attr(*, "data_types")= chr [1:28] "C" "C" "C" "C" ...

```

Con la lista con nombres, podemos utilizar también los corchetes para acceder a sus elementos, en este caso, para acceder a cada elemento de la lista, se utiliza un doble corchete. Ya con el elemento extraído, podemos utilizar los corchetes sencillos (ya que habremos extraído un elemento que no es una lista).

```

mi_lista[["vel"]]

```

```

##      auto    peaton     bici
## 21.33333 4.77551 16.00000

```

```

mi_lista[["vel"]][2]

```

```

## peaton
## 4.77551

```

```
mi_lista[["censo"]][1:5,]
```

```
##          NOM_MUN AGEB MZA VIVTOT POBTOT
## 2860 Azcapotzalco 0222 001     511   1685
## 371  Azcapotzalco 1095 012     470   1646
## 712  Azcapotzalco 0491 009     548   1612
## 1749 Azcapotzalco 0665 002     582   1609
## 1866 Azcapotzalco 0129 034     420   1589
```

## 4 Funciones intermedias

En este capítulo veremos diferentes funciones dentro de R que nos sirven para comparar datos y generar bucles o ciclos (loops) y condicionales para eficientar los análisis realizados, incluyendo las funciones tipo “apply”, que sirven para aplicar funciones a diferentes estructuras de datos. En la sección de utilidades veremos el uso de expresiones regulares, manipulación de datos y el manejo de fechas y horas.

### 4.1 Condicionales y control de flujo

#### 4.1.1 Operadores relacionales

Los operadores relacionales o comparadores, son operadores que nos ayudan a ver como un objeto se relaciona con otro en R. Por ejemplo, podemos revisar si dos objetos son iguales. Para obtener este resultado, utilizamos el operador doble igual `==`.

```
TRUE == TRUE
```

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

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

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

```
2 == 2
```

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

```
3 == 2
```

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

El operador contrario al signo de doble igual es el de desigualdad, utilizamos un signo de exclamación seguido de un signo de igual, `!=`. Esto nos devuelve un resultado lógico dependiendo de si la desigualdad es cierta o falsa.

```
TRUE != TRUE
```

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

```
"hola" != "adios"
```

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

```
2 != 2
```

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

```
3 != 2
```

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

R también utiliza como operadores los signos “menor que” y “mayor que”, así como las combinaciones “menor o igual” y “mayor o igual”. Con ellos podemos evaluar valores numéricos y caractéres. En este último caso, R realiza una evaluación por la posición alfabética de los caractéres, es decir, “adios” es mayor que “hola” por que la “a” viene primero que la “h”. Al realizar la evaluación de un vector lógico, TRUE es mayor que FALSE, ya que dentro del sistema, TRUE equivale a 1 y FALSE a 0.

```
2 > 1
```

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

```
"hola" > "adios"
```

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

```
TRUE < FALSE
```

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

Es posible utilizar los operadores relacionales con vectores. Regresando al vector de tiempos, podemos comparar un vector con un valor y devolver un vector lógico. Utilizando el vector de tiempos generado previamente, podemos saber los valores que son menores o iguales a 10 minutos.

```
tiempos
```

```
##    auto      tp    bici peatón
##      5       10     15     20
```

```
tiempos_10 <- tiempos <= 10
tiempos_10
```

```
##    auto      tp    bici peatón
##    TRUE    TRUE   FALSE  FALSE
```

Con el vector creado, podemos utilizar la indexación para extraer los tiempos en donde se cumple que son menores o iguales a 10 minutos.

```
tiempos[tiempos_10]
```

```
##    auto      tp
##      5       10
```

A su vez, es posible comparar la relación entre dos vectores, en este ejemplo, suponemos que tenemos dos vectores de tiempo, para diversos modos de transporte, con viajes realizados en días diferentes. Podemos saber en qué días el tiempo en cada modo de transporte es mayor o igual entre el martes y el jueves. Vemos que viajar en martes resultó más rápido para todos los modos excepto la bicicleta.

```

modo <- c("auto", "tp", "bici", "peaton")
martes <- c(7, 8, 12, 25)
jueves <- c(6, 9, 13, 25)

cbind(modo, martes, jueves)

```

```

##      modo     martes   jueves
## [1,] "auto"    "7"      "6"
## [2,] "tp"       "8"      "9"
## [3,] "bici"     "12"     "13"
## [4,] "peaton"   "25"     "25"

```

```
martes >= jueves
```

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

#### 4.1.2 Operadores lógicos

Los operadores lógicos sirven para cambiar o combinar los resultados de los operadores relacionales. En R, los operadores lógicos son **AND**, **OR** y **NOT** (**&**, **|**, **!**).

El operador **AND** se utiliza para combinar condiciones que deben cumplirse al mismo tiempo. **OR** se utiliza para cuando requerimos que se cumpla una de varias condiciones.

```
# Operador AND
TRUE & TRUE
```

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

```
FALSE | TRUE
```

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

```
x <- 12
x > 5 & x < 15
```

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

```
x <- 17
x > 5 & x < 15
```

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

```
# Operador OR
TRUE | TRUE
```

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

```
FALSE | TRUE
```

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

```
y <- 4  
y < 5 | y > 15
```

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

```
y <- 14  
y < 5 | y > 15
```

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

Por su parte, el operador **NOT** sirve para obtener el inverso de **AND**, es decir, los registros que NO cumplen la condición.

```
is.numeric(5)
```

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

```
!is.numeric(5)
```

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

#### 4.1.3 Condicionales

Las sentencias condicionales utilizan la función **if()** con la cual evalúan una o varias condiciones. Si la condición se evalúa como **TRUE**, se ejecuta el código asociado con la sentencia. La condición a revisar se escribe dentro del paréntesis, mientras que el código a ejecutar se escribe entre llaves.

```
if(condicion) {  
  expresion  
}
```

Podemos evaluar si un número es positivo o negativo, e imprimir una respuesta del sistema de acuerdo con la condición, si la condición se evalúa como falsa, no se produce respuesta.

```
x <- -3  
if(x < 0) {  
  print("x es negativo")  
}
```

```
## [1] "x es negativo"
```

Para conseguir un resultado en caso de que no se cumpla la condición, utilizamos la sentencia **else**, que no necesita una expresión condicional explícita, pero debe utilizarse en conjunto con **if**. El código asociado a la sentencia **else** se ejecuta cuando la condición no se satisface.

```

if(condicion) {
  expr1
} else {
  expr2
}

```

Para el ejemplo anterior, podemos añadir un texto que indique que el número es positivo

```

x <- 3
if(x < 0) {
  print("x es negativo")
} else {
  print("x es positivo o cero")
}

## [1] "x es positivo o cero"

```

Finalmente, para combinar condiciones, podemos utilizar la sentencia **else if**, que permite revisar condiciones en pasos.

```

if(x < condicion1) {
  expr1
} else if(condicion2) {
  expr2
} else {
  expr3
}

```

Así podemos evaluar cuando el valor es negativo, cero o positivo.

```

x <- 0
if(x < 0) {
  print("x es negativo")
} else if(x == 0) {
  print("x es cero")
} else {
  print("x es positivo")
}

## [1] "x es cero"

```

## 4.2 Bucles (loops)

Los loops son útiles para realizar procesos iterativos, los bucles **while** son procesos **if** repetitivos, mientras que los bucles **for** son iteraciones sobre los elementos en una secuencia.

### 4.2.1 Bucle while

Los bucles **while** son similares a las sentencias **if** debido a que ejecutan el código dentro de ellos dependiendo de si se cumple o no una condición. Sin embargo, el bucle **while** se seguirá ejecutando mientras esta condición

se siga cumpliendo. La sintáxis de un bucle **while** es muy similar a la de las sentencias condicionales, en el siguiente ejemplo, vemos como ejecutamos el bucle en donde simplemente se incrementa un contador hasta que llega a un valor especificado, en este caso 7.

```
n <- 1
while(n <= 7) {
  print(paste("contador es igual a", n))
  n <- n + 1
}
```

```
## [1] "contador es igual a 1"
## [1] "contador es igual a 2"
## [1] "contador es igual a 3"
## [1] "contador es igual a 4"
## [1] "contador es igual a 5"
## [1] "contador es igual a 6"
## [1] "contador es igual a 7"
```

La sentencia **break** “rompe” el bucle cuando R la encuentra, interrumpiendo el código activo en el bucle. En el ejemplo, detenemos el bucle cuando el valor asignado a “n” es divisible entre 5, a través del operador “modulo” `%%`.

```
n <- 1
while(n <= 7) {
  if(n %% 5 == 0) {
    break
  }
  print(paste("contador es igual a", n))
  n <- n + 1
}
```

```
## [1] "contador es igual a 1"
## [1] "contador es igual a 2"
## [1] "contador es igual a 3"
## [1] "contador es igual a 4"
```

#### 4.2.2 Bucle for

Es una sentencia de control que permite iterar una expresión o código sobre los elementos de un vector en una secuencia definida. La condición se prueba primero y después se ejecuta el código contenido. La sintáxis es como sigue:

```
for(valor in secuencia) {
  expr
}
```

Con el siguiente código, podríamos emular lo que realizaba el bucle **while** en la sección anterior, con la diferencia de que la condición está como una secuencia, además de que no es necesario inicializar la variable, pues se mantiene interna dentro del bucle.

```

for(n in 1:7) {
  print(paste("contador es igual a", n))
}

```

```

## [1] "contador es igual a 1"
## [1] "contador es igual a 2"
## [1] "contador es igual a 3"
## [1] "contador es igual a 4"
## [1] "contador es igual a 5"
## [1] "contador es igual a 6"
## [1] "contador es igual a 7"

```

Lo anterior vuelve más práctico definir tanto las variables internas del bucle, como las condiciones. La instrucción **break** también puede utilizarse dentro de un bucle **for** de la misma manera que se utilizó en la sección anterior. Por su parte, existe la función **next**, que omite iterar sobre los valores que cumplen con la condición descrita dentro de la función. Como se observa, el número 4 no se imprime.

```

for(n in 1:7) {
  if(n == 4) {
    next
  }
  print(paste("contador es igual a", n))
}

```

```

## [1] "contador es igual a 1"
## [1] "contador es igual a 2"
## [1] "contador es igual a 3"
## [1] "contador es igual a 5"
## [1] "contador es igual a 6"
## [1] "contador es igual a 7"

```

## 4.3 Funciones

### 4.3.1 Concepto de funciones

Como hemos visto, las funciones son muy importantes en R y en casi cualquier otro lenguaje de programación. Las funciones que se han discutido hasta ahora, fueron implementadas en R desde un principio (en R base) o generadas para cumplir cierta tarea a través del trabajo de la comunidad y a la para del desarrollo en la tecnología de procesamiento computacional. Por ejemplo, funciones que realizan procesos de Machine Learning no hubieran sido posibles (o viables) con los procesadores de 1997. Así como funciones que utilizan procesos espaciales y geográficos.

Una función es una especie de caja negra, que se alimenta de un input, realiza los procesos y devuelve un output. Por ejemplo, la función **sd()** devuelve la desviación estándar (standard deviation) de una serie de valores. Conocemos que la expresión para calcular la desviación estándar es:

$$\sigma = \sqrt{\frac{\sum_{i=1}^N (x_i - \bar{x})^2}{N - 1}}$$

De acuerdo con esta fórmula, podemos ver que la función está compuesta por las operaciones necesarias, pero además probablemente utilice otras funciones, ya que la instrucción **sd()** está diseñada para trabajar con cualquier cantidad de datos numéricos, en el ejemplo que sigue colocamos cuatro. Vemos que la función y las operaciones regresan el mismo resultado.

```

sd(c(1, 5, 6, 7))

## [1] 2.629956

valores <- c(1, 5, 6, 7)
media <- mean(valores)
xi <- valores - media
xi2 <- xi^2
suma <- sum(xi2)
suma_n <- suma/(length(valores)-1)
sd <- sqrt(suma_n)
sd

## [1] 2.629956

```

Como se vio anteriormente, las función ? antes del nombre de cualquier función nos lleva a su documentación. La función `sd()` es muy sencilla en cuanto a sus argumentos, y dentro de ella solo existe una expresión, sin embargo, las funciones también incluyen generalmente mensajes de error, advertencias, etc. En resumen, las funciones son una caja negra por lo cual es necesario conocer su comportamiento y tener idea de los resultados esperados.

#### 4.3.2 Creación de funciones

Escribir una función no es complicado y puede ser muy útil cuando sabemos que vamos a realizar un proceso a lo largo de todo el flujo de trabajo. Por ejemplo, es posible convertir el código escrito anteriormente para calcular la desviación estándar en una función, con la siguiente sintaxis:

```

mi_fun<- function(arg1, arg2) {
  cuerpo
}

```

De tal forma, que a manera simplificada, la función podría escribirse como:

```

mi_sd <- function(valores) {
  media <- mean(valores)
  xi <- valores - media
  xi2 <- xi^2
  suma <- sum(xi2)
  suma_n <- suma/(length(valores)-1)
  sd <- sqrt(suma_n)
  sd
}

```

Lo que guarda la expresión como una función, que podemos revisar en la parte inferior del panel “Environment”. Si corremos la función, nos dará como resultado la desviación estándar de los valores que coloquemos. Cabe destacar que los argumentos de una función no son variables dentro del entorno de R, y las variables declaradas dentro de ella, son locales, por lo que tampoco son añadidas al entorno.

```

mi_sd(c(1, 5, 6, 7))

```

```

## [1] 2.629956

```

Ahora, podemos calcular la desviación estándar para cualquier número de valores (ya podíamos con la función `sd` pero ahora lo haremos con nuestra propia función).

```
vector <- c(1,6,7,4,8,23,7,6,54,7,25,27)
sd(vector)
```

```
## [1] 15.21637
```

```
mi_sd(vector)
```

```
## [1] 15.21637
```

## 4.4 Familia “apply”

La familia “apply” está compuesta por una serie de funciones que permiten aplicar una función a una secuencia de datos, de tal forma que es posible en muchos casos cambiar un `for` loop por alguna función de la familia `apply`, principalmente `lapply()` y sus variantes `sapply()` y `vapply()`.

### 4.4.1 Función `lapply()`

La función `lapply()` toma tres argumentos, una lista `X`, una función (o el nombre de una función) `FUN`, y otros argumentos a través del argumento `....`. Si `X` no es una lista, `lapply()` la convierte en una.

Como ejemplo, un vector que contiene los números del 1 al 4, se puede pasar como argumento de la función `runif()` (random uniform) que genera un número de números aleatorios. En el siguiente ejemplo, generamos 5 números aleatorios entre 0 y 1, ya que los argumentos por defecto de la función son `min = 0` y `max = 1`.

```
runif(5)
```

```
## [1] 0.4390184 0.1818913 0.8835260 0.6892357 0.6246632
```

Con `lapply()`, podemos recorrer sobre la serie de `1:4` y generar 1, 2, 3 y 4 números aleatorios entre 0 y 1. El siguiente código toma el vector “`x`”, lo convierte en una lista y le aplica la función `runif` a cada elemento. Esto hace posible que `lapply()` trabaje sobre todos los elementos ya que permite que cada uno sea de tipos o clases diferentes.

```
x <- 1:4
lapply(x, runif)
```

```
## [[1]]
## [1] 0.7529031
##
## [[2]]
## [1] 0.253445468 0.009344053
##
## [[3]]
## [1] 0.8709816 0.3454784 0.5115231
##
## [[4]]
## [1] 0.37650877 0.05310051 0.49138539 0.66969940
```

El resultado es una lista de elementos de diferente longitud, como se mencionó, por practicidad el resultado de la función siempre es una lista, a la cual se le puede aplicar la función **unlist()** para transformarla en un vector, que solo agrega los elementos conforme van apareciendo en la lista.

```
unlist(lapply(x, runif))

## [1] 0.8239178 0.4403011 0.8901076 0.2284826 0.9627415 0.3590993 0.8997396
## [8] 0.1266923 0.8814917 0.6697903
```

El argumento **...** se utiliza para hacer uso de *funciones anónimas*, que son funciones creadas dentro de **lapply()** que solamente son utilizadas dentro de la misma.

Por ejemplo, si alimentamos una lista de matrices a la función, y queremos extraer la primera columna de cada una, podemos utilizar **lapply()** con una función anónima.

```
x <- list(a = matrix(1:4, 2, 2), b = matrix(1:6, 3, 2))
x
```

```
## $a
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## $b
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```
lapply(x, function(arg) arg[,1])
```

```
## $a
## [1] 1 2
##
## $b
## [1] 1 2 3
```

#### 4.4.2 Función **sapply()**

El nombre de la función **sapply()** viene de “simplify apply”, y se puede utilizar cuando sabemos que los resultados de la función serán todos del mismo tipo. En resumen, es el paso directo de utilizar las funciones **lapply()** seguida de **unlist()**.

```
x <- list(a = 1:4, b = rnorm(10), c = rnorm(20,1), d = rnorm(100,5))
sapply(x, mean)
```

```
##           a           b           c           d
## 2.5000000 0.4059358 0.9313485 4.9740284
```

Como vemos, la lista x está compuesta por vectores de diferente longitud, sin embargo, **sapply()** es capaz de operar sobre cada uno de los elementos y obtener la media para cada caso, devolviendo un vector con nombres. Por su parte, la función **mean()** no puede operar sobre una lista.

```
mean(x)

## Warning in mean.default(x): argument is not numeric or logical: returning NA

## [1] NA
```

Existen otras variantes para la familia de funciones apply, cada una para casos específicos. Con **vapply()**, la diferencia es que podemos indicar explícitamente el formato de salida.

## 4.5 Utilidades

En este apartado enumeramos una serie de funciones y procesos que son útiles para el día a día en R. Primero veremos algunas funciones matemáticas generales, seguido de una introducción a expresiones regulares (regex), que sirven para identificar caracteres dentro de una cadena de texto. Finalmente pasamos, de manera general, a revisar el formato de trabajo en R para manejar fechas y horas.

### 4.5.1 Utilidades matemáticas

La función **abs()** devuelve el valor absoluto de un arreglo numérico de datos.

```
abs(-5)

## [1] 5

abs(c(-3.65, 4.12, 5.84, -8.26))

## [1] 3.65 4.12 5.84 8.26
```

La función **round()** redondea los datos de entrada, se puede utilizar el argumento **digits** para definir el número de dígitos a mostrar.

```
round(c(-3.65, 4.12, 5.84, -8.26), digits = 1)

## [1] -3.6 4.1 5.8 -8.3

round(c(-3.65, 4.12, 5.84, -8.26), digits = 0)

## [1] -4 4 6 -8
```

Por otra parte, funciones que ya han aparecido previamente en este documento son **sum()** y **mean()**, que calculan la suma y media de los datos, respectivamente.

```
sum(round(c(-3.65, 4.12, 5.84, -8.26), digits = 2))

## [1] -1.95
```

```
mean(round(c(-3.65, 4.12, 5.84, -8.26), digits = 2))
```

```
## [1] -0.4875
```

El siguiente grupo de funciones corresponde a funciones que son utilizadas comúnmente al crear y manipular estructuras de datos.

La función **seq()** la hemos visto previamente, sirve para generar una secuencia de números, en donde los primeros dos argumentos representan el límite de la secuencia, mientras que el argumento **by** indica los incrementos para realizar esa secuencia. Para añadir a lo visto anteriormente, utilizar un número negativo en el argumento **by** genera una serie descendente.

```
seq(8, 2, by = -2)
```

```
## [1] 8 6 4 2
```

Por otra parte, la función **rep()** se utiliza para replicar su input, que generalmente es un vector o una lista. Con el argumento **times** podemos especificar el número de repeticiones a realizar, mientras que el argumento **each** repite cada valor seguido del siguiente.

```
rep(c(8, 6, 4, 2), times = 2)
```

```
## [1] 8 6 4 2 8 6 4 2
```

```
rep(c(8, 6, 4, 2), each = 2)
```

```
## [1] 8 8 6 6 4 4 2 2
```

La función **str()** da la estructura de un objeto, se puede utilizar para cualquier objeto pero es más útil en objetos de varios elementos, como listas o dataframes. El siguiente ejemplo nos da la estructura de los elementos de la lista “li”, en donde el primer elemento es de tipo lógico y contiene un valor TRUE, el segundo elemento es de tipo carácter y contiene la cadena de texto “hola”, mientras que el último elemento es un vector generado por las funciones **seq()** y **rep()** y ordenado con **sort()**.

```
li <- list(log = TRUE,
           ch = "hola",
           int_vec = sort(rep(seq(8, 2, by = -2),
                             times = 2)))
str(li)
```

```
## List of 3
## $ log      : logi TRUE
## $ ch       : chr "hola"
## $ int_vec: num [1:8] 2 2 4 4 6 6 8 8
```

Las funciones **is.()** (is punto), sirven para verificar si un objeto es de cierto tipo o clase, por ejemplo:

```
is.numeric(5)
```

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

```

is.list(li)

## [1] TRUE

is.numeric(li)

## [1] FALSE

```

Existen muchas más funciones y es probable que la mayoría de las tareas básicas que se busquen realizar ya tengan una o más funciones para llevarlas a cabo. Se recomienda entender el objetivo de las tareas a realizar en R, diseñar un algoritmo para realizarlas y después investigar sobre funciones que puedan servir para llevarlo a cabo.

#### 4.5.2 Expresiones regulares (regex)

Una expresión regular es una secuencia de caracteres y metacaracteres que conforman un patrón de búsqueda que se utiliza para empatar con cadenas de texto. Se puede utilizar una expresión regular para checar si cierto patrón existe en un texto, para reemplazar estos patrones con otros elementos o para extraerlos de la cadena.

Las regex son particularmente útiles cuando limpiamos datos, se necesita un curso específico de regex para comprenderlas por completo, pero podemos ver algunas funciones introductorias y expresiones comunes.

La función **grepl()** sirve para extraer los elementos de un vector de texto que cumplen con un patrón. El primer argumento en la función **grepl()** es dicho patrón, mientras que el segundo es un vector de caracteres en el cual se buscan las coincidencias. En el ejemplo, buscamos el carácter “a”, para encontrar las ciudades que lo contienen. Vemos que **grepl()** identifica entre mayúsculas y minúsculas, pues devuelve TRUE para Monterrey y FALSE para Tijuana.

```

ciudades <- c("Guadalajara", "Monterrey", "Tijuana", "Acapulco", "Aguascalientes")
grepl("t", ciudades)

```

```

## [1] FALSE  TRUE FALSE FALSE  TRUE

```

Donde empiezan a ser utilizadas las regex, es al incluir metacaracteres, que son caracteres que representan una instrucción en lugar de representar al carácter mismo. Por ejemplo, si quisieramos encontrar las ciudades que empiezan con alguna letra, debemos utilizar la expresión “^a”, mientras que para los elementos que terminan con cierta letra, utilizaremos la expresión “a\$”.

Con la función **grep()** encontramos un vector de índices, en lugar del vector lógico que devuelve **grepl()** (la “l” al final es por “logical”). La función **grep()** devuele un vector de índices que contiene los valores de posición de los elementos que contienen el patrón buscado.

```

grep("t", ciudades)

```

```

## [1] 2 5

```

```

ciudades[grep("t", ciudades)]

```

```

## [1] "Monterrey"      "Aguascalientes"

```

Por otra parte, R permite reemplazar ciertos caracteres con otros, para esto se utiliza la función **sub()**. Esta función toma tres argumentos: **pattern**, **replacement** y **x**. El primero corresponde a la expresión regular que se quiere coincidir y el tercero corresponde al vector de caracteres en donde se busca el patrón. El nuevo argumento **replacement**, toma el carácter o cadena de caracteres de reemplazo.

```
mosca <- c("una", "mosca", "pegada", "en", "la", "pared")  
  
sub("[aeiou]", "a", mosca)  
  
## [1] "ana"     "masca"   "pagada"  "an"      "la"      "pared"
```

Como vemos, introdujimos una expresión regular que indica que el patrón es cualquier vocal, lo que da la instrucción a R de aplicar el reemplazo de caracteres si es que encuentra alguna vocal. En el caso de la palabra “la”, R reemplaza una “a” por otra “a” tras bambalinas. Es importante notar que **sub()** solamente reemplaza la primera coincidencia en cada elemento del vector, es por eso que en la palabra “pared” solamente se reemplaza la primera vocal, mientras que la “e” se mantiene. Esto no es notorio con la palabra “pegada”, pues reemplazó la primera vocal “e” por una “a” y las demás vocales ya eran “a”.

Para realizar reemplazos sobre todas coincidencias en los elementos del vector, utilizamos la función **gsub()**, que toma los mismos argumentos que **sub()**.

```
gsub("[aeiou]", "a", mosca)  
  
## [1] "ana"     "masca"   "pagada"  "an"      "la"      "parad"
```

#### 4.5.3 Manejo de fechas

Finalmente, dentro de las utilidades de R existe el concepto del manejo de fechas y horas. Esta información es esencial en el manejo de datos de tráfico pues la demanda es dependiente de los meses, semanas, días y horas en que se analiza. R es específicamente útil al realizar series de tiempo para establecer temporalidad y estacionalidad.

Para esto, R trabaja con fechas y horas en formatos específicos, que le permiten interpretar los datos de manera adecuada.

Con la función **Sys.Date()** podemos conocer la fecha actual (si nuestro sistema se encuentra correcto), además de ver el formato por defecto de las fechas en R.

```
Sys.Date()  
  
## [1] "2022-04-15"
```

El resultado es una cadena de texto que contiene el año, el mes y el día, en ese orden, sin embargo, este es un tipo de objeto especial, un objeto “Date”.

```
class(Sys.Date())  
  
## [1] "Date"
```

Por otra parte, podemos obtener la hora del sistema, a través de la función **Sys.time()**, la cual devuelve otra cadena de caracteres, esta vez como un objeto de tipo POSIXct. POSIX es una convención que permite a los sistemas computacionales entender datos de tiempo en diferentes sistemas operativos.

```
Sys.time()  
  
## [1] "2022-04-15 02:48:17 CDT"  
  
class(Sys.time())  
  
## [1] "POSIXct" "POSIXt"
```

Para crear un objeto de tipo “Date”, utilizamos la función **as.Date()**, que con el argumento **format** por defecto, tiene que estar escrita en el formato mostrado anteriormente, que es el formato por defecto de R. Para procesar series de tiempo, no es necesario modificar el formato de representación de las fechas, sin embargo, puede ser de utilidad realizar adecuaciones al formato para realizar diferentes visualizaciones, por ejemplo, mostrar el formato que utilizamos en México: día/mes/año.

```
mi_fecha <- as.Date("1988-10-01")  
mi_fecha2 <- as.Date("01/10/1988", format = "%d/%m/%Y")  
  
mi_fecha
```

```
## [1] "1988-10-01"  
  
mi_fecha2  
  
## [1] "1988-10-01"
```

Por su parte, para crear un objeto de tipo POSIXct colocamos toda la cadena de caracteres con el formato que R espera, o utilizando el argumento **format** para establecer el formato de entrada.

```
mi_hora <- as.POSIXct("1988-10-01 10:00:00")  
mi_hora  
  
## [1] "1988-10-01 10:00:00 CST"
```

Es posible realizar operaciones sobre ambos tipos de objetos, en el caso del objeto “Date”, utilizan como unidad los días, así que si sumamos 1 a la fecha, esta se incrementará en un día. Por su parte, los objetos “POSIXct” tienen a los segundos como unidad, así que para sumar una hora hay que incrementar el valor en 3600.

```
mi_fecha + 1  
  
## [1] "1988-10-02"  
  
mi_fecha + 365  
  
## [1] "1989-10-01"
```

```
mi_hora + 1  
## [1] "1988-10-01 10:00:01 CST"  
  
mi_hora + 3600  
## [1] "1988-10-01 11:00:00 CST"
```

## 5 Tidyverse

Tidyverse es una colección de paquetes de R diseñado para la ciencia de datos, todos los paquetes dentro del Tidyverse comparten una misma filosofía, gramática y estructura. Cuando se instala y se carga la librería, se cargan principalmente 8 librerías más, que son *dplyr*, *ggplot2*, *readr*, *forcats*, *tibble*, *stringr*, *tidyverse* y *purr*. En conjunto, estos paquetes se utilizan para la importación, manipulación y visualización de datos. En este capítulo, realizaremos análisis y procesos con el Censo de Población de INEGI que abordamos en capítulos anteriores.

### 5.1 Manejo de datos

En este capítulo realizaremos tres procesos sobre una tabla, dos que ya hemos visto antes: filtrar datos de una tabla y ordenarlos en un arreglo deseado. El tercer proceso es uno de los más importantes en R, particularmente en *tidyverse*, **mutate()**.

#### 5.1.1 Datos

Los datos que utilizaremos serán los del censo, pero está vez se encuentran en la carpeta “data” del proyecto, en lugar de descargarlos con el paquete *importinegi* los importaremos con *readr*, que es parte del *tidyverse*.

```
library(tidyverse)

## -- Attaching packages ----- tidyverse 1.3.1 --

## v ggplot2 3.3.5     v purrr    0.3.4
## v tibble   3.1.6     v stringr  1.4.0
## v tidyverse 1.1.3     vforcats  0.5.1
## v readr    2.1.0

## -- Conflicts ----- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()

censo <- read_csv("data/censo.csv")

## Rows: 68941 Columns: 230

## -- Column specification -----
## Delimiter: ","
## chr (228): ENTIDAD, NOM_ENT, MUN, NOM_MUN, LOC, NOM_LOC, AGEB, MZA, POBFEM, ...
## dbl    (2): POBTOT, VIVTOT
##
## i Use 'spec()' to retrieve the full column specification for this data.
## i Specify the column types or set 'show_col_types = FALSE' to quiet this message.
```

Como vimos previamente, podemos hacer una revisión rápida de la estructura de la base de datos. Con la función **str()** podemos observar su estructura (no se imprime en este documento por espacio).

```
str(censo)
```

Viendo la estructura, podemos ver que la mayoría de las variables se importaron como “caracter”, esto debido a que estas columnas contienen un “\*”, como criterio de confidencialidad, se refiere a AGEBs y manzanas con menos de tres unidades, por lo que solo se presentan los datos agregados.

Lo primero que podemos observar es que el dataframe está compuesto por datos para manzanas, AGEBs, municipios y estados, pero también incluye los totales. Lo anterior nos puede perjudicar pues al realizar procesos y visualizaciones, se agragaran los totales y estaremos sumando dos o más veces las cantidades. Podemos identificar que las columnas que agregan estos totales tienen identificador “000”, que son las variables “LOC” por localidad, “MZA” que corresponde a las manzanas y “AGEB”.

```
head(censo)[5:8]
```

```
## # A tibble: 6 x 4
##   LOC    NOM_LOC          AGEB    MZA
##   <chr>  <chr>          <chr>  <chr>
## 1 0000  Total de la entidad 0000   000
## 2 0000  Total del municipio 0000   000
## 3 0001  Total de la localidad urbana 0000   000
## 4 0001  Total AGEB urbana 0010   000
## 5 0001  Azcapotzalco 0010   001
## 6 0001  Azcapotzalco 0010   002
```

### 5.1.2 Función filter()

Por lo anterior, conviene filtrar las observaciones que tienen estos identificadores para continuar con el trabajo. En este momento vale la pena introducir el operador `%>%`, conocido como “pipe”. Este operador nos permite escalar funciones, para ahorrar pasos al momento de escribirlas. Con la sintaxis de la función `filter()` podemos omitir el primer argumento, ya que se lo estamos alimentando a través del pipe. Hasta este momento no es muy evidente el beneficio pero posteriormente será más obvio.

```
censo_filtro <- cens %>%
  filter(MZA != "000")
```

El resultado del filtro se almacena en la variable correspondiente, filtramos todos los elementos en donde la variable “MZA” no es igual a tres ceros, recordando que la variable está almacenada como carácter, por lo cual agregamos las comillas.

Cabe destacar que para el caso de números identificadores, es conveniente mantenerlas como carácter, para evitar ordenamientos indeseados en el caso de tener ceros a la izquierda.

Es posible realizar filtros de dos o más variables, por ejemplo, podemos generar una base de datos de población de manzanas y AGEBs para la alcaldía Xochimilco.

```
censo_xochi <- cens %>%
  filter(MZA != "000", NOM_MUN == "Xochimilco")
```

### 5.1.3 Función arrange()

Previamente exploramos la función `sort()`, que ordena un vector de manera ascendente o descendente. Ahora añadiremos la función `arrange()` que sirve para ordenar observaciones dentro de un dataframe de acuerdo a el contenido de una variable. Con la función `desc()` ordenamos de mayor a menor.

```
censo_arrange <- censo %>%
  arrange(desc(POBTOT))
```

Por otra parte, podemos aprovechar para recordar la utilidad de la función **select()** para seleccionar solo tres variables y verificar que el orden se realizó de manera correcta. A su vez, realizamos el orden sobre el objeto censo original para mostrar que las filas de totales distorsionan la información. Asimismo, en el siguiente código utilizamos un doble pipe, para imprimir solamente los primeros 5 resultados a través de la función **head()**. Es de notar que de esta forma, el objeto de entrada **censo\_arrange** se escribe una sola vez, se realizan las operaciones de la función **select()** y el resultado de este proceso se convierte en el input de la siguiente función en el pipe.

```
censo_arrange %>%
  select("NOM_MUN", "AGEB", "POBTOT") %>%
  head(5)
```

```
## # A tibble: 5 x 3
##   NOM_MUN           AGEB    POBTOT
##   <chr>            <chr>   <dbl>
## 1 Total de la entidad Ciudad de México 0000  9209944
## 2 Iztapalapa          0000  1835486
## 3 Iztapalapa          0000  1835486
## 4 Gustavo A. Madero  0000  1173351
## 5 Gustavo A. Madero  0000  1173351
```

Con lo anterior, podemos generar una base que contenga las variables necesarias para trabajar, filtrando las observaciones que no son totales y ordenando de mayor a menor.

```
censo_cdmx <- censo %>%
  filter(MZA != "000") %>%
  arrange(desc(POBTOT)) %>%
  select("NOM_MUN", "AGEB", "MZA", "VIVTOT", "POBTOT", "POBMAS", "POBFEM", "PNACENT",
         "PNACOE", "P3YM_HLI", "P3HLINHE")
```

#### 5.1.4 Función **mutate()**

La función **mutate()** se utiliza para transformar el contenido de una variable o crear una variable nueva, basado en operaciones y procesos de otras variables (o de sí misma) dentro del dataframe.

Para ejemplificar, podemos realizar algunos cálculos sobre los datos que separamos, como por ejemplo, el porcentaje de la población femenina en cada AGEBC, o el porcentaje de la población que habla alguna lengua indígena. Para realizar lo anterior, es necesario convertir las variables a operar en variables numéricas.

Podemos observar que dentro de la selección que hicimos, todavía existen algunas variables con asteriscos, y a su vez sabemos que esto significa que en ese rubro existen tan pocos datos, que no se muestran por confidencialidad. Por lo que es válido, en este caso, cambiarlos por ceros.

Existen varias formas de realizar este proceso, sin embargo, para utilizar solamente las funciones que hemos visto hasta el momento, podemos generar dos bases de datos, una que contenga las variables descriptivas y otra que contenga las variables numéricas.

Si le indicamos a la función **select()** las variables a seleccionar, se genera una nueva tabla con las mismas, a su vez, con esas mismas variables podemos indicar el negativo de ese vector, lo que seleccionará todas **excepto** esas variables.

```

censo_cdmx_id <- censo_cdmx %>%
  select("NOM_MUN", "AGEB", "MZA")

head(censo_cdmx_id, 1)

## # A tibble: 1 x 3
##   NOM_MUN      AGEB    MZA
##   <chr>        <chr> <chr>
## 1 Miguel Hidalgo 1349  013

censo_cdmx_datos <- censo_cdmx %>%
  select(-c("NOM_MUN", "AGEB", "MZA"))
head(censo_cdmx_datos, 1)

## # A tibble: 1 x 8
##   VIVTOT POBTOT POBMAS POBFEM PNACENT PNACOE P3YM_HLI P3HLINHE
##   <dbl>  <dbl> <chr>  <chr>  <chr>  <chr>   <chr>
## 1 3284   10484 4952    5532   8866   1547    78     *

```

Para remover los asteriscos, podemos utilizar la función `gsub()`, que nos permitirá reemplazarlos por el carácter que necesitemos, en este caso colocamos el cero. Para esto, usamos `sapply()` para aplicar la función `gsub()` a cada columna de la base de datos, lo que nos devuelve una matriz, aprovechamos para convertir todos los datos en numéricos con `as.numeric()` y después hacemos un pipe para convertir esa matriz en un dataframe con `as.data.frame()`.

```

censo_cdmx_num <- sapply(censo_cdmx_datos, function(y) as.numeric(gsub("\\*", "0", y))) %>%
  as.data.frame()

## Warning in FUN(X[[i]], ...): NAs introduced by coercion
## Warning in FUN(X[[i]], ...): NAs introduced by coercion
## Warning in FUN(X[[i]], ...): NAs introduced by coercion
## Warning in FUN(X[[i]], ...): NAs introduced by coercion

```

Vemos que la función para convertir en números introdujo NAs en los casos en donde no es posible identificar un valor numérico, esto corresponde a registros que en la base se identifican como “N/D”, R los convierte en NA, que en inglés significa lo mismo, así que no tenemos que hacer transformaciones adicionales, más adelante veremos como trabajar con este tipo de registros faltantes.

Finalmente, utilizamos la función `cbind()` para volver a colocar las columnas en el mismo dataframe.

```

censo_cdmx_bind <- cbind(censo_cdmx_id, censo_cdmx_num)
str(censo_cdmx_bind)

## 'data.frame': 66456 obs. of 11 variables:
## $ NOM_MUN : chr "Miguel Hidalgo" "Iztapalapa" "Gustavo A. Madero" "Cuajimalpa de Morelos" ...
## $ AGEB    : chr "1349" "1994" "0154" "0369" ...
## $ MZA     : chr "013" "001" "001" "098" ...
## $ VIVTOT  : num 3284 3 4 2137 1285 ...

```

```

## $ POBTOT : num 10484 9686 8184 4975 4368 ...
## $ POBMAS : num 4952 8515 8184 2422 2116 ...
## $ POBFEM : num 5532 1171 0 2553 2252 ...
## $ PNACENT : num 8866 7454 6389 3521 3690 ...
## $ PNACOE : num 1547 1264 984 848 617 ...
## $ P3YM_HLI: num 78 146 78 23 54 17 74 51 20 20 ...
## $ P3HLINHE: num 0 0 0 0 0 0 0 0 0 0 ...

```

Una forma alternativa de quitar los asteriscos en un solo paso sin separar las bases de datos, es la siguiente, sin embargo, contiene algunos operadores que no han sido mencionados hasta ahora, como los corchetes vacíos y el punto.

```

censo_cdmx[] <- censo_cdmx %>%
  lapply(., function(x) gsub("\\\\*", "0", x))

```

Ahora que tenemos una base trabajable, trabajémosla. Utilizando la función **mutate()** podemos realizar operaciones y aplicar funciones a las variables dentro del dataframe. En este caso, empecemos con estimar una nueva columna, que contenga la proporción de población femenina en cada manzana. Para esto, necesitamos declarar el nombre de la nueva columna dentro de la función, seguido por un signo de igual para después colocar la operación a realizar. En este momento cabe mencionar la importancia de no utilizar el signo de igual para declarar variables, ya que es utilizado por requisito en este tipo de funciones.

```

censo_cdmx_bind %>%
  mutate(pct_fem = POBFEM / POBTOT) %>%
  head(5)

```

```

##           NOM_MUN AGEB MZA VIVTOT POBTOT POBMAS POBFEM PNACENT PNACOE
## 1      Miguel Hidalgo 1349 013   3284 10484  4952  5532    8866   1547
## 2      Iztapalapa 1994 001      3   9686  8515  1171    7454   1264
## 3 Gustavo A. Madero 0154 001      4   8184  8184      0    6389    984
## 4 Cuajimalpa de Morelos 0369 098   2137   4975  2422    2553    3521    848
## 5 Cuajimalpa de Morelos 0119 009   1285   4368  2116    2252    3690    617
##   P3YM_HLI P3HLINHE pct_fem
## 1      78        0 0.5276612
## 2     146        0 0.1208961
## 3      78        0 0.0000000
## 4      23        0 0.5131658
## 5      54        0 0.5155678

```

Con lo anterior hemos estimado la proporción de población femenina por manzana en la Ciudad de México, sin embargo, podría ser interesante para las visualizaciones mostrar este dato en porcentajes, por lo que con la misma función realizamos la transformación.

```

censo_cdmx_bind %>%
  mutate(pct_fem = paste(round(POBFEM / POBTOT * 100, 2), "%")) %>%
  head(5)

```

```

##           NOM_MUN AGEB MZA VIVTOT POBTOT POBMAS POBFEM PNACENT PNACOE
## 1      Miguel Hidalgo 1349 013   3284 10484  4952  5532    8866   1547
## 2      Iztapalapa 1994 001      3   9686  8515  1171    7454   1264
## 3 Gustavo A. Madero 0154 001      4   8184  8184      0    6389    984
## 4 Cuajimalpa de Morelos 0369 098   2137   4975  2422    2553    3521    848

```

```

## 5 Cuajimalpa de Morelos 0119 009    1285    4368    2116    2252    3690    617
##   P3YM_HLI P3HLINHE pct_fem
## 1      78          0 52.77 %
## 2     146          0 12.09 %
## 3      78          0     0 %
## 4      23          0 51.32 %
## 5      54          0 51.56 %

```

Aunque esto no es realmente recomendable, pues veremos más adelante que el formato de la visualización puede alterarse dentro de la gramática del gráfico, pero es una demostración de la funcionalidad de **mutate()**.

## 5.2 Visualización

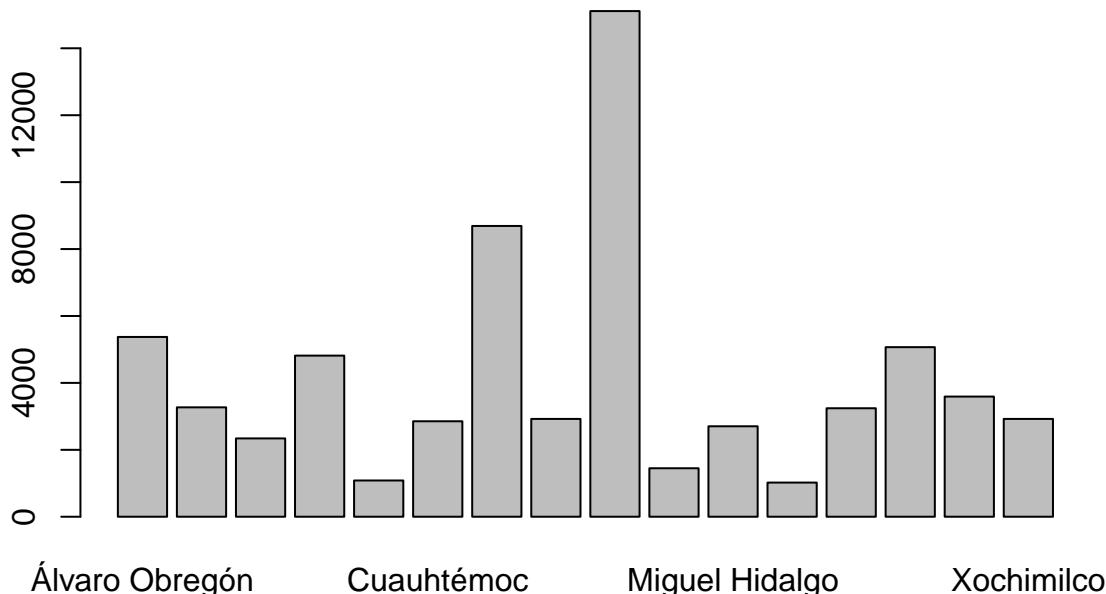
Hasta este punto, hemos realizado análisis sobre tablas y valores, sin embargo, una parte importante de la comunicación de resultados es la creación de visualizaciones atractivas e informativas. R cuenta con la función **plot()** que puede crear visualizaciones rápidas y que es capaz de determinar el tipo de gráfico a generar de acuerdo con las variables que se le suministran.

Por ejemplo, si el input es una variable de tipo factor, **plot()** generará un histograma. Sin embargo, este tipo de gráfico no es útil para representar los datos de población, el resultado del gráfico es el número (cuenta) de manzanas en cada alcaldía.

```

censo <- censo_cdmx_bind
plot(factor(censo$NOM_MUN))

```

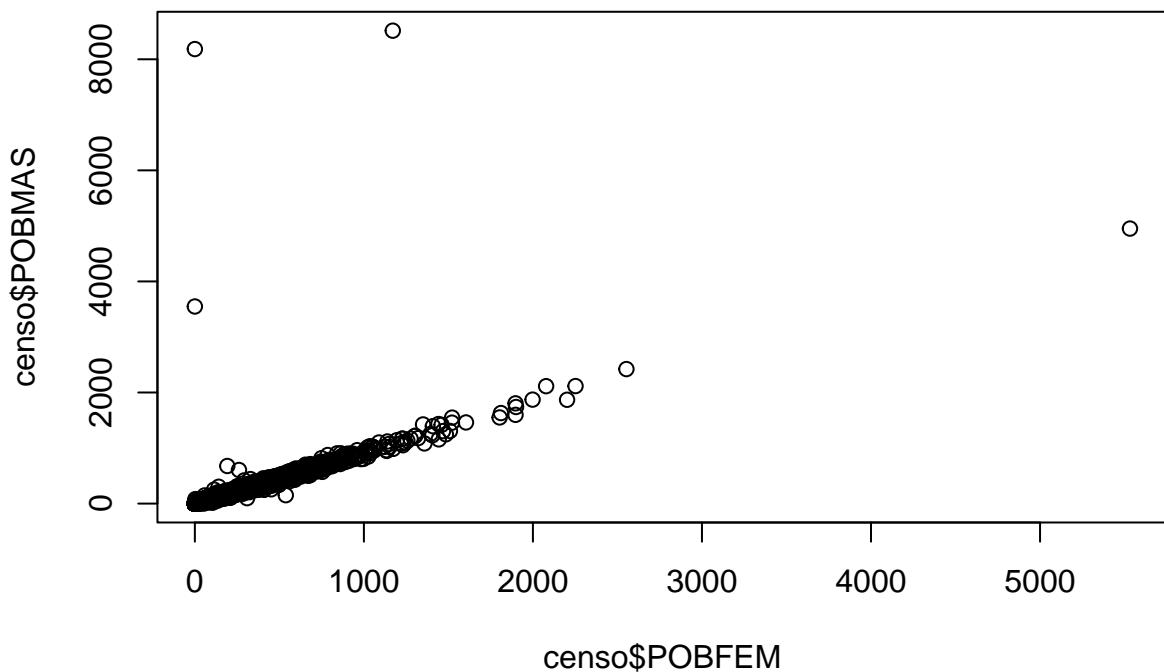


Por otra parte, si el input son dos variables numéricas, la visualización generada será un gráfico de puntos (scatterplot). En el siguiente ejemplo, podemos ver la relación de población femenina vs población masculina.

```

plot(censo$POBFEM, censo$POBMAS)

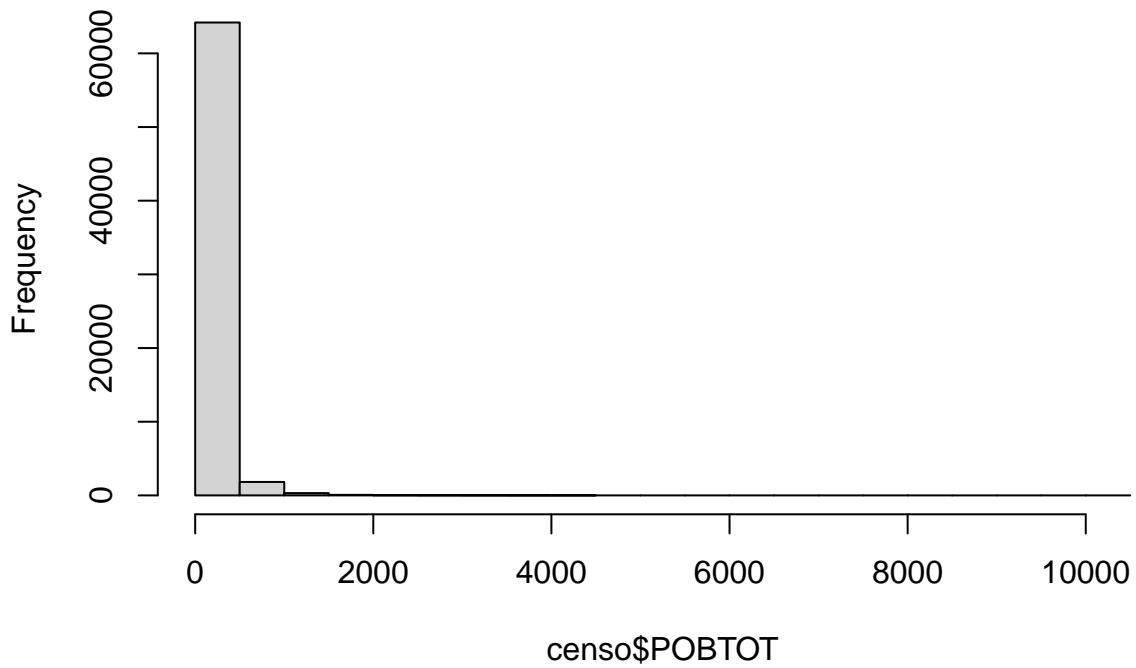
```



Para forzar la visualización de un histograma, podemos utilizar la función **hist()**.

```
hist(censo$POBTOT)
```

**Histogram of censo\$POBTOT**



En este caso, se genera un histograma que contiene la población en cada una de las manzanas, sin embargo, muchas de ellas tienen población cero o muy pequeña, por lo que conviene realizar agrupaciones en las variables del dataframe. A su vez, la librería *gplot2* (parte del tidyverse) se creó para permitir el uso de la agrupación y síntesis en la creación de gráficos que siguen una gramática específica en la escritura del código.

En las próximas dos secciones se desarrollan las funciones necesarias para generar síntesis desde un dataframe y generar visualizaciones que permiten más control del usuario.

### 5.3 Agrupación y síntesis

Los análisis de datos usualmente oinvolucran varios pasos para realizar transformaciones y visualizaciones de datos, así como otros componentes en el flujo de trabajo, como la modelación de datos, que se no se verá en este curso. Hemos visto que con la función `filter()` podemos extraer observaciones individuales o grupos de observaciones. En esta sección mostramos como convertir una serie de observaciones en una sola.

La función `summarise()` permite aplicar una función y generar una síntesis a una o varias variables. De esta forma, podemos generar la suma de la población en toda la Ciudad de México, con el siguiente código:

```
censo %>%
  summarise(POBTOT = sum(POBTOT))
```

```
##      POBTOT
## 1  9145590
```

La sintaxis de la función es parecida a la de `mutate()`, en donde se alimenta el nombre de la variable resultante, seguido de las operaciones o funciones que se aplicarán a los datos. Lo anterior no suena tan crucial cuando se trata de una variable, pero es posible realizar síntesis de diversas variables y aplicando diferentes funciones.

```
censo %>%
  summarise(POBTOT = sum(POBTOT),
            MEDIA_FEM = mean(POBFEM))
```

```
##      POBTOT MEDIA_FEM
## 1  9145590  71.71288
```

La función `summarise()` devuelve un valor que resulta de la síntesis de una o varias variables de acuerdo a la función indicada. Sin embargo, puede ser conveniente realizar la síntesis por grupos. Lo anterior es parecido al concepto de tablas dinámicas en excel, en donde podemos realizar operaciones a los valores de acuerdo con variables categóricas. La información del censo funciona para este ejemplo pues podemos obtener la población por alcaldía, al agruparlas y generar la suma.

```
censo %>%
  group_by(NOM_MUN) %>%
  summarise(POBTOT = sum(POBTOT))
```

```
## # A tibble: 16 x 2
##   NOM_MUN          POBTOT
##   <chr>           <dbl>
## 1 Álvaro Obregón  759003
## 2 Azcapotzalco    432205
## 3 Benito Juárez   434153
## 4 Coyoacán        614447
## 5 Cuajimalpa de Morelos 212735
## 6 Cuauhtémoc     545842
## 7 Gustavo A. Madero 1173351
```

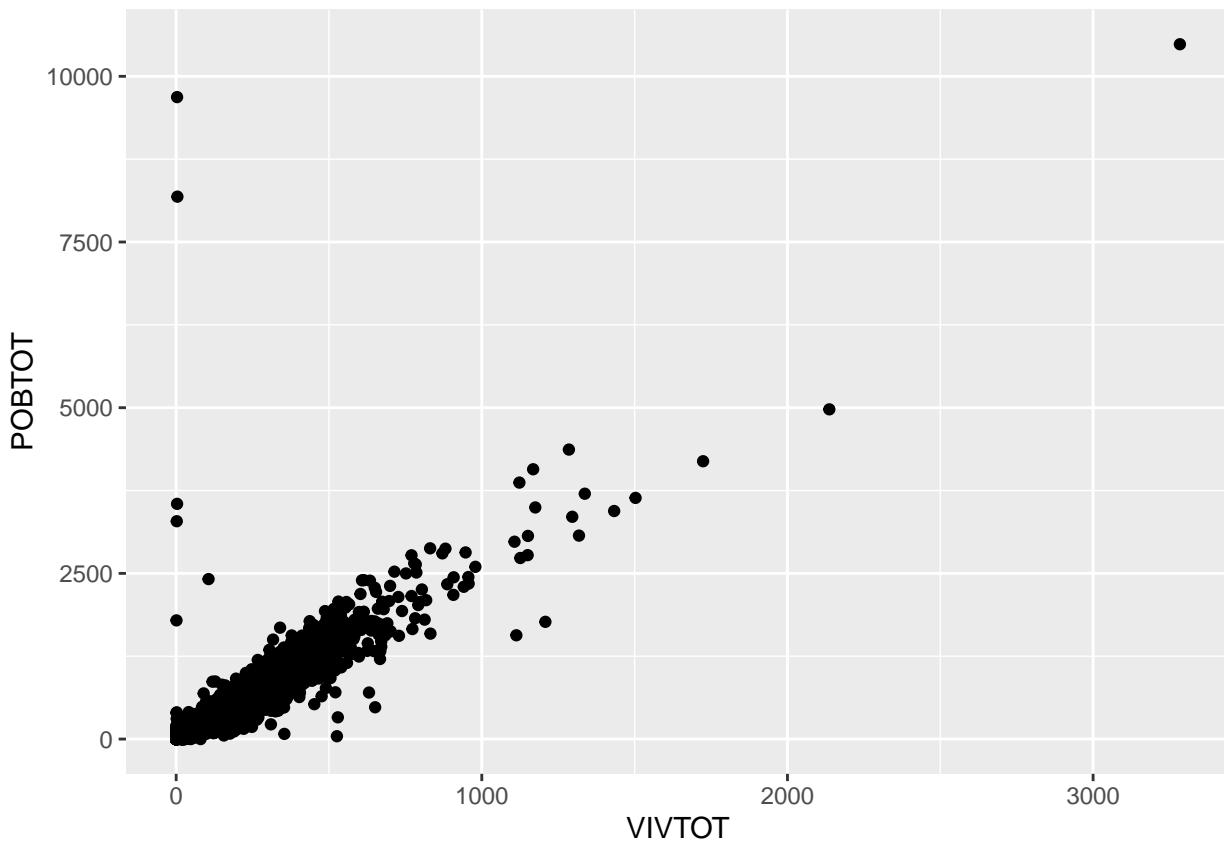
```

##  8 Iztacalco          404695
##  9 Iztapalapa         1835486
## 10 La Magdalena Contreras 246428
## 11 Miguel Hidalgo      414470
## 12 Milpa Alta          128710
## 13 Tláhuac              385321
## 14 Tlalpan              685559
## 15 Venustiano Carranza 443704
## 16 Xochimilco           429481

```

El código previo genera un dataframe nuevo, que solo contiene la variable agrupada y la síntesis del valor seleccionado.

Con lo anterior, utilizando *ggplot2* podemos generar visualizaciones más atractivas y sobretodo, informativas. Como introducción a esta librería, tenemos la siguiente visualización generada con una gramática específica, que veremos en la siguiente sección.



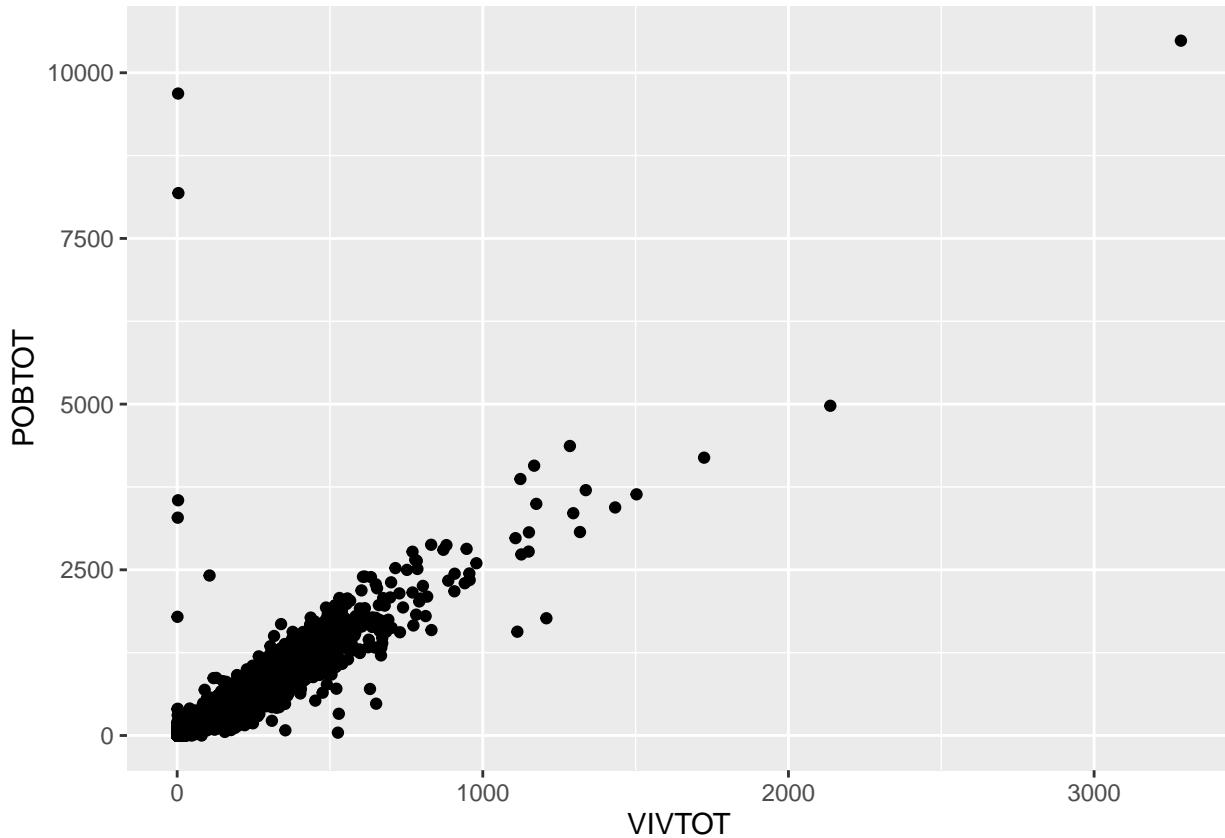
## 5.4 Tipos de visualización

Utilizando *ggplot2*, podemos generar el gráfico mostrado anteriormente, que con la sintaxis y la gramática propias del código, nos permite tener más control sobre el resultado.

Primero, el input de la función **ggplot()** se compone de la serie de datos, las aesthetics (estéticas) que indican la forma de representar a las variables y pueden ser diferentes de acuerdo con el tipo de gráfico. Lo anterior seguido de una geometría, que define el tipo de representación a mostrar.

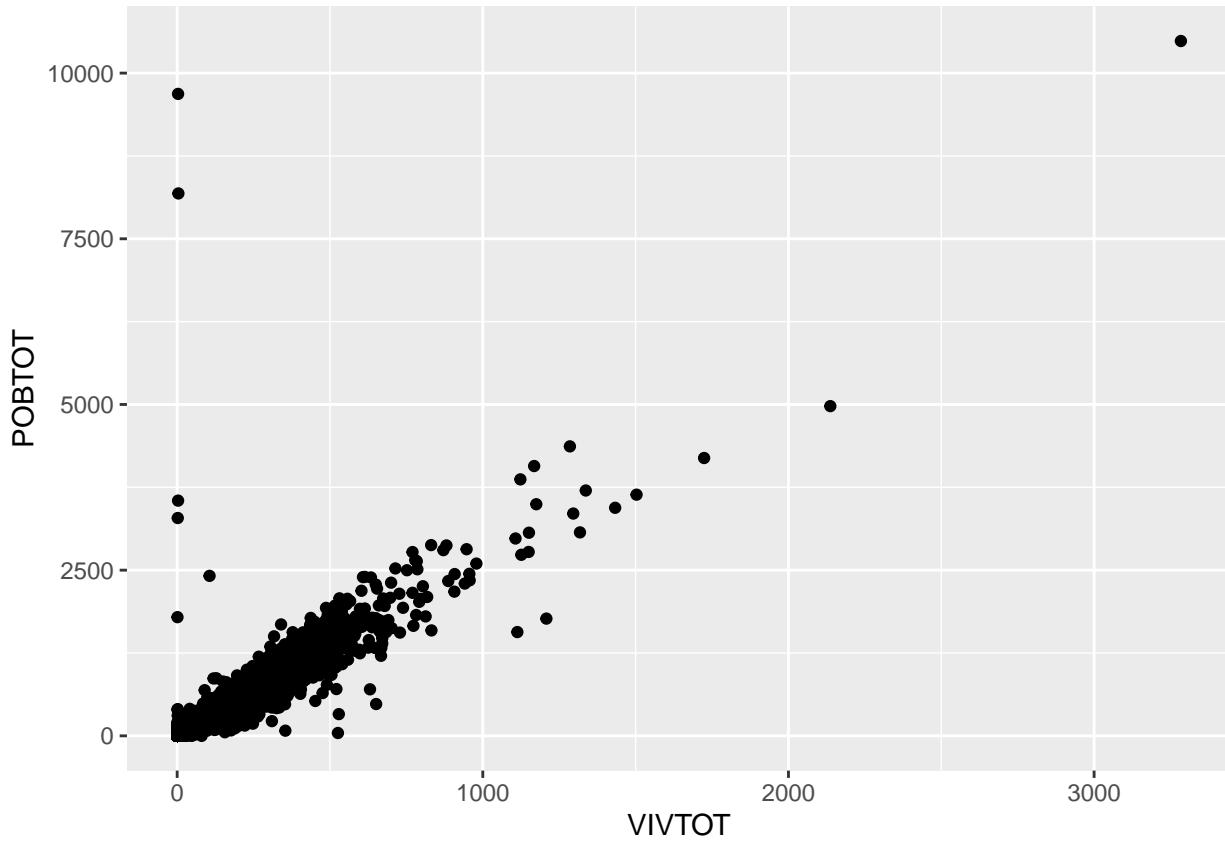
Por último, hay que notar que cada elemento del gráfico se agrega con un signo +.

```
ggplot(censo, aes(x = VIVTOT, y = POBTOT)) +  
  geom_point()
```



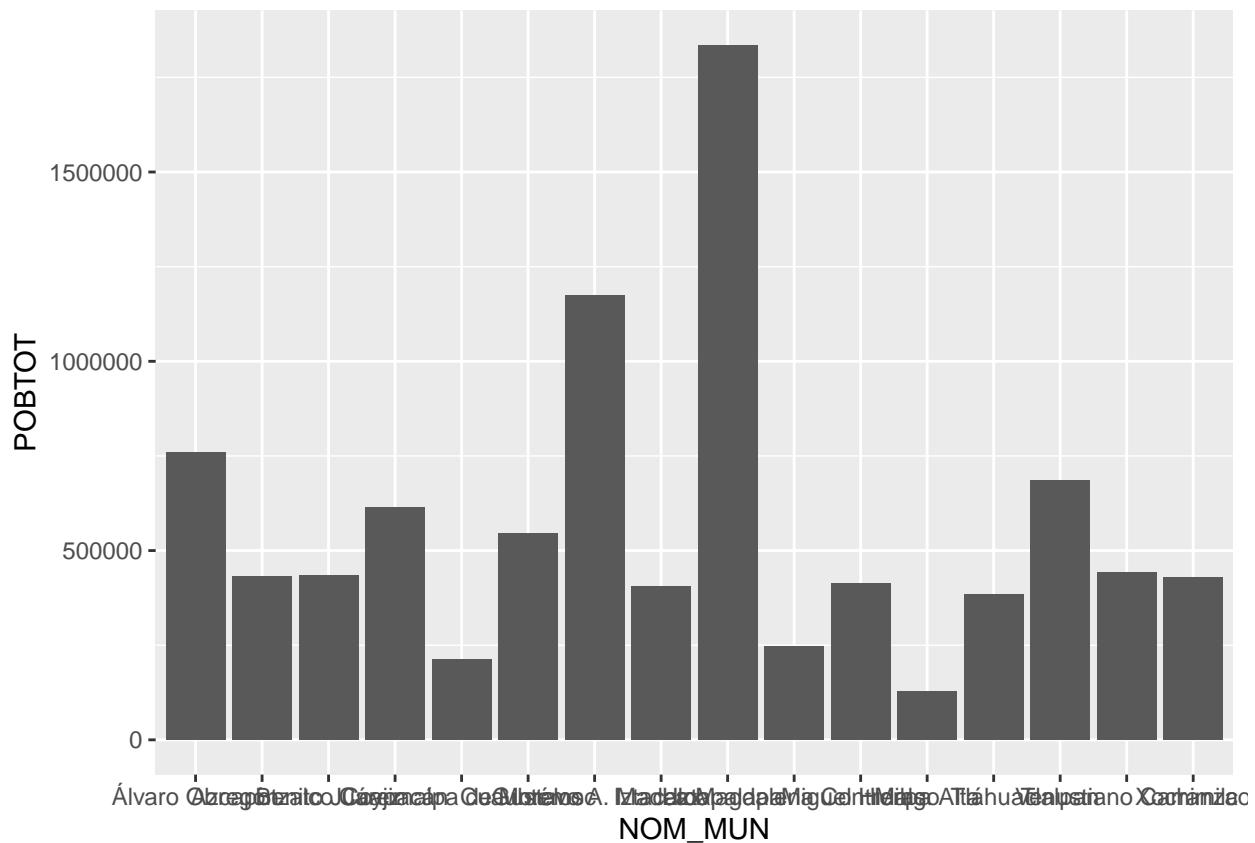
Otra forma de escribir la misma instrucción, y que funciona como convención, especialmente cuando ponemos varias geometrías, es utilizar el operador pipe y colocar las estéticas dentro de los argumentos de la geometría. En resumen, el código anterior utilizaría las variables x y y en todas las geometrías consecuentes, mientras que el código siguiente solamente utilizará estas variables para realizar la gráfica de puntos.

```
censo %>%  
  ggplot() +  
  geom_point(aes(x = VIVTOT, y = POBTOT))
```



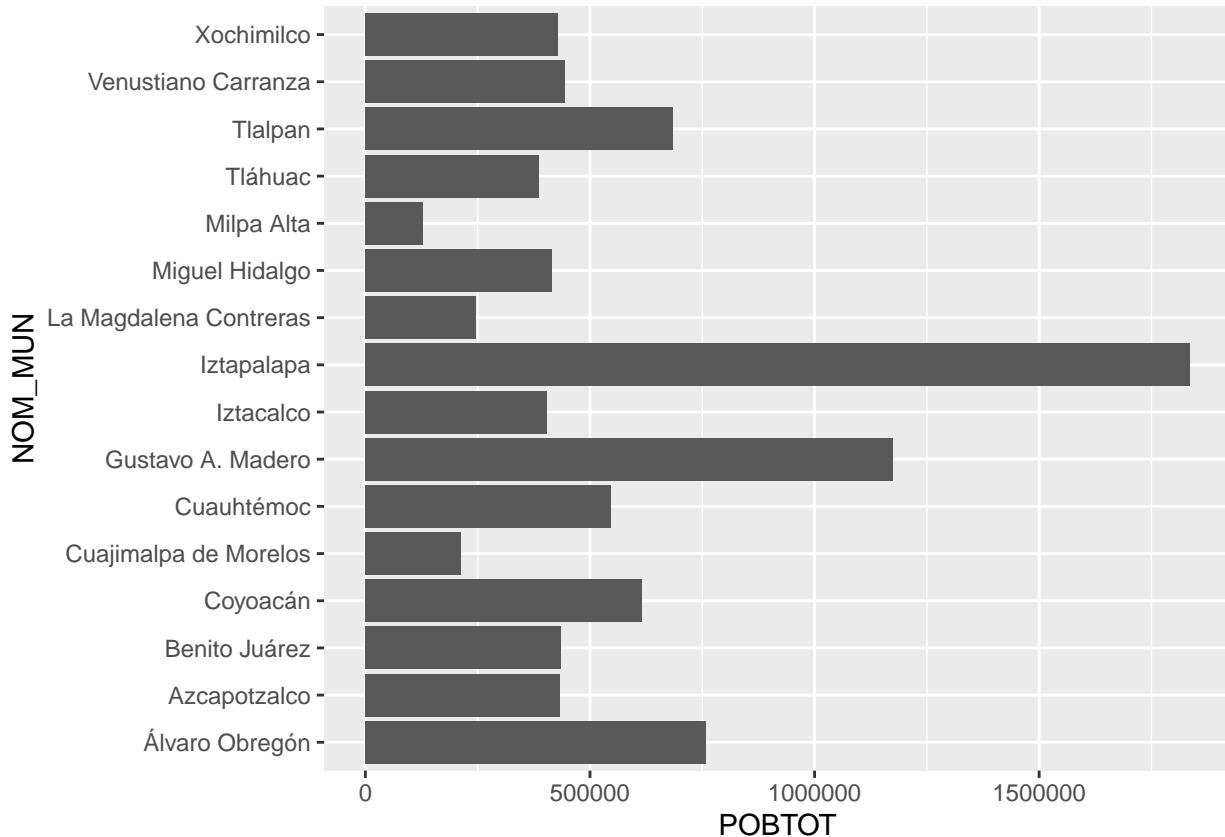
Otro tipo de gráfico es el de barras, que tiene dos geometrías relacionadas, la primera **geom\_bar()**, genera un gráfico que toma una variable categórica en “x” y dibuja en forma de barras el número de apariciones de esa variable (conteo), en ese sentido, es lo mismo que la visualización que realizamos previamente con la función **plot()**. La otra geometría asociada es **geom\_col()** (de columnas), en donde además de la variable “x”, podemos incluir una variable “y” que indique los valores incluidos en esta variable.

```
censo %>%
  group_by(NOM_MUN) %>%
  summarise(POBTOT = sum(POBTOT)) %>%
  ggplot() +
  geom_col(aes(x = NOM_MUN, y = POBTOT))
```



Un componente adicional de las visualizaciones es el tema. Utilizando diferentes gramáticas de temas, podemos modificar la estética del gráfico, en este caso, podríamos modificar la dirección o el tamaño del texto en el eje x, pero probablemente sería más fácil leer todo el gráfico si cambiamos la dirección de las barras, lo que se puede lograr añadiendo la función **coord\_flip()**, que invierte las variables del plano.

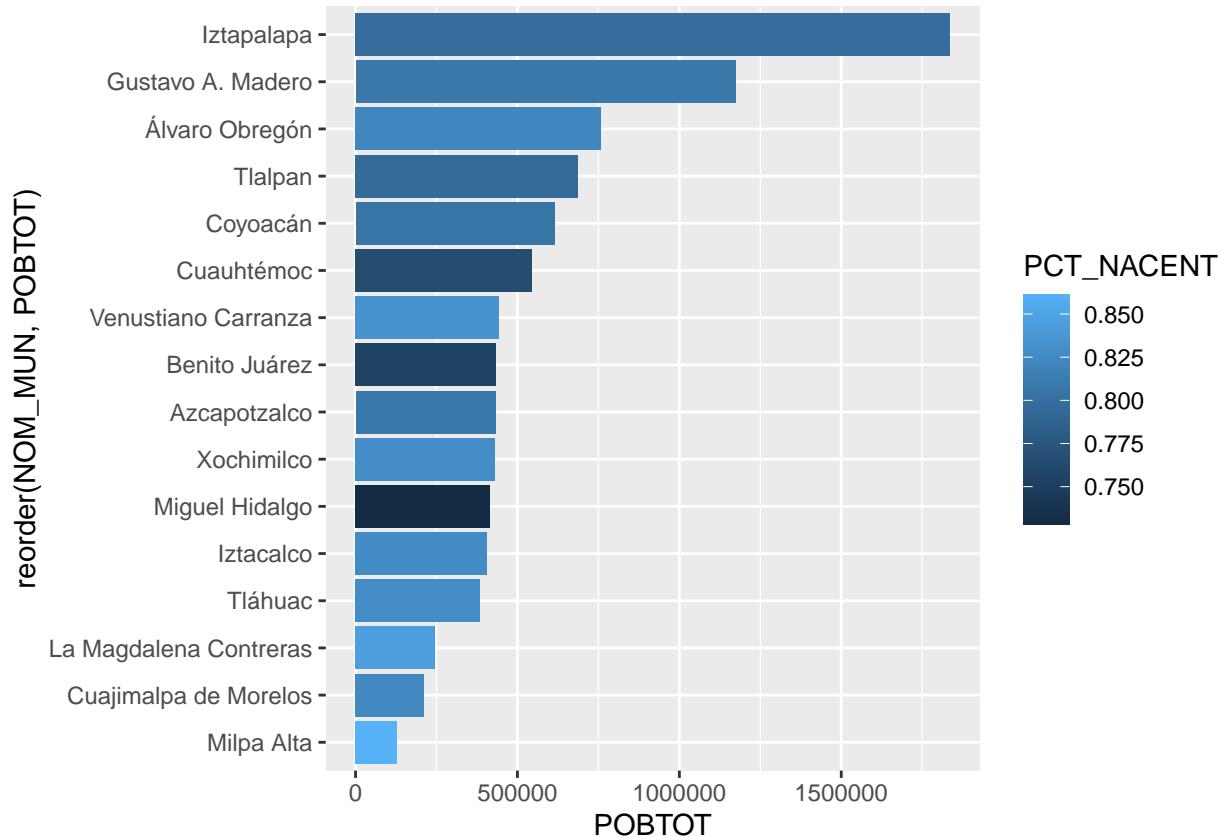
```
censo %>%
  group_by(NOM_MUN) %>%
  summarise(POBTOT = sum(POBTOT)) %>%
  ggplot() +
  geom_col(aes(x = NOM_MUN, y = POBTOT)) +
  coord_flip()
```



Dentro de las aesthetics, podemos agregar algunas más, que son propiedades del gráfico que pueden tomar en cuenta otras variables. Para esto, podemos generar otra variable basada en el porcentaje de personas nacidas en la entidad, realizamos una síntesis a las variables requeridas, utilizando `summarise_at()`, que es una variante de `summarise()` que permite seleccionar muchas variables a la vez y aplicarles una función, además hace posible indicarle que no contemple los registros “NA”. Es seguida realizamos un `mutate()` para estimar los porcentajes, y el resultado lo pasamos directamente a `ggplot()`.

El resultado del porcentaje de personas nacidas en la entidad, lo pasamos a la estética `fill`, que define una escala de colores que puede ser discreta o continua (como en este caso) con el color de relleno de las barras. Finalmente, utilizando la función `reorder` dentro de la variable `x`, reordenamos las barras con base en la población total, que es la variable que estamos visualizando.

```
censo %>%
  group_by(NOM_MUN) %>%
  summarise_at(vars(VIVTOT, POBTOT, PNACENT), sum, na.rm = TRUE) %>%
  mutate(PCT_NACENT = PNACENT / POBTOT) %>%
  ggplot() +
  geom_col(aes(x = reorder(NOM_MUN, POBTOT), y = POBTOT, fill = PCT_NACENT)) +
  coord_flip()
```



## 6 Importación y Unión de datos

En este capítulo En este capítulo revisamos la importación de datos en forma de dataframes y el manejo de la unión de tablas.

### 6.1 Importación de datos

Además de poder importar datos en formato .csv, R es capaz de manejar una gran variedad de archivos, incluyendo archivos de Excel, archivos espaciales como shapefiles, Google Earth, imágenes, etc. Casi cualquier archivo puede ser manejado dentro de R con la librería adecuada.

#### 6.1.1 Archivos de excel

Para importar un archivo de excel, la librería básica es *readxl*, con la cual podemos utilizar la función *read\_excel()* y con un archivo bien estructurado, no necesita argumentos más que la ruta y el nombre del archivo.

```
library(readxl)  
copexa <- read_excel("data/COPEXA_ejemplo.xlsx")
```

La tabla anterior contiene datos históricos desde 2013 para unos activos carreteros, y utilizando lo que hemos visto hasta ahora, podemos generar la tabla de Tránsito Diario Promedio Anual. Primero convertimos la columna “fecha” a un objeto Date, en seguida extraemos el año de esta columna, generamos la agregación por tipo de vehículo, agrupamos por activo y año, y luego sumamos el total y lo dividimos entre el total de valores en cada año.

```
copexa$fecha <- as.Date(copexa$fecha)  
  
copexa_tpda <- copexa %>%  
  mutate(year = format(fecha, format = "%Y")) %>%  
  mutate(auto = A1 + AUTO + A2_1 + A2_2 + A2_3_,  
         bus = B2 + B3 + B4,  
         cu = CU2 + CU3 + CU4,  
         ca1 = CA1_5 + CA1_6,  
         ca2 = CA2_7 + CA2_8 + CA2_9 + CA2_10_) %>%  
  group_by(activo, year) %>%  
  summarise(A = sum(auto)/n(),  
            B = sum(bus)/n(),  
            CU = sum(cu)/n(),  
            CA1 = sum(ca1)/n(),  
            CA2 = sum(ca2)/n())
```

```
## `summarise()` has grouped output by 'activo'. You can override using the  
## `groups` argument.
```

```
copexa_tpda  
  
## # A tibble: 18 x 7  
## # Groups:   activo [2]
```

```

##   activo                  year     A     B    CU   CA1   CA2
##   <chr>                   <chr> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 Autopista Perote - Banderilla 2013 4031. 352. 304. 229. 168.
## 2 Autopista Perote - Banderilla 2014 4295. 412. 332. 313. 244.
## 3 Autopista Perote - Banderilla 2015 4775. 427. 373. 375. 252.
## 4 Autopista Perote - Banderilla 2016 4749. 437. 399. 394. 233.
## 5 Autopista Perote - Banderilla 2017 4480. 427. 398. 439. 313.
## 6 Autopista Perote - Banderilla 2018 4603. 447. 459. 602. 389.
## 7 Autopista Perote - Banderilla 2019 4359. 444. 526. 769. 517.
## 8 Autopista Perote - Banderilla 2020 3383. 192. 528. 777. 523.
## 9 Autopista Perote - Banderilla 2021 4374. 204. 586. 889. 616.
## 10 Libramiento de Xalapa      2013 1237. 97.1 125. 329. 337.
## 11 Libramiento de Xalapa      2014 1348. 152. 116. 389. 353.
## 12 Libramiento de Xalapa      2015 1659. 161. 138. 450. 331.
## 13 Libramiento de Xalapa      2016 1745. 171. 171. 464. 293.
## 14 Libramiento de Xalapa      2017 1685. 156. 221. 481. 369.
## 15 Libramiento de Xalapa      2018 1851. 163. 262. 586. 430.
## 16 Libramiento de Xalapa      2019 1760. 162. 345. 755. 583.
## 17 Libramiento de Xalapa      2020 1330. 65.2 357. 796. 602.
## 18 Libramiento de Xalapa      2021 1806. 69.9 422. 928. 720.

```

Además de contener información sobre el tráfico en los activos carreteros, la base de datos “copexa” también contiene los registros de ingresos en las casetas de cobro. Por lo que con unas líneas de código muy parecidas, podemos obtener el resumen de ingresos anuales.

```

copexa_ing <- copexa %>%
  mutate(year = format(fecha, format = "%Y")) %>%
  mutate(auto = ING_A1 + ING_AUTO + ING_A2_1 + ING_A2_2 + ING_A2_3_,
         bus = ING_B2 + ING_B3 + ING_B4,
         cu = ING_CU2 + ING_CU3 + ING_CU4,
         ca1 = ING_CA1_5 + ING_CA1_6,
         ca2 = ING_CA2_7 + ING_CA2_8 + ING_CA2_9 + ING_CA2_10_) %>%
  group_by(activo, year) %>%
  summarise(A = sum(auto),
            B = sum(bus),
            CU = sum(cu),
            CA1 = sum(ca1),
            CA2 = sum(ca2))

```

```

## `summarise()` has grouped output by 'activo'. You can override using the
## `.`groups` argument.

```

```
copexa_ing
```

```

## # A tibble: 18 x 7
## # Groups:   activo [2]
##   activo                  year     A     B    CU   CA1   CA2
##   <chr>                   <chr> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 Autopista Perote - Banderilla 2013 126845000 11634634. 1.10e7 9.74e6 1.42e7
## 2 Autopista Perote - Banderilla 2014 143044001. 14376535. 1.28e7 1.41e7 2.19e7
## 3 Autopista Perote - Banderilla 2015 163520250. 15450473. 1.48e7 1.73e7 2.34e7
## 4 Autopista Perote - Banderilla 2016 175812363. 17188521. 1.72e7 1.99e7 2.37e7
## 5 Autopista Perote - Banderilla 2017 195635370. 20013526. 2.04e7 2.67e7 3.83e7

```

```

## 6 Autopista Perote - Banderilla 2018 206477149. 23385021. 2.51e7 4.24e7 5.48e7
## 7 Autopista Perote - Banderilla 2019 203758476. 25506743. 3.01e7 5.77e7 7.65e7
## 8 Autopista Perote - Banderilla 2020 158771322. 11086645. 3.11e7 6.27e7 8.31e7
## 9 Autopista Perote - Banderilla 2021 211132822. 12132256. 3.55e7 7.85e7 9.92e7
## 10 Libramiento de Xalapa 2013 126845000 11634634. 1.10e7 9.74e6 1.42e7
## 11 Libramiento de Xalapa 2014 52945005. 6197678. 8.61e6 3.70e7 4.10e7
## 12 Libramiento de Xalapa 2015 67230971. 6792076. 1.06e7 4.42e7 3.96e7
## 13 Libramiento de Xalapa 2016 73963853. 7601022. 1.38e7 4.76e7 3.68e7
## 14 Libramiento de Xalapa 2017 73580528. 7139500 1.83e7 5.08e7 4.77e7
## 15 Libramiento de Xalapa 2018 82940380. 8203865. 2.32e7 6.64e7 6.09e7
## 16 Libramiento de Xalapa 2019 82222906. 9022023. 3.08e7 8.71e7 8.97e7
## 17 Libramiento de Xalapa 2020 62446296. 3640402. 3.20e7 9.61e7 9.71e7
## 18 Libramiento de Xalapa 2021 87141402. 4018966. 3.89e7 1.16e8 1.18e8

```

En este caso, cambiamos las variables, para utilizar las que corresponden a los ingresos en lugar del aforo, además de mantener solamente la suma de ingresos, ya que estos no se ponderan por los días del año.

## 6.2 Unión interna

Cuando tenemos datos en diferentes tablas, que pueden tener una relación entre ellos, podemos realizar uniones y generar nuevos grupos de datos. Esta acción es homóloga a las búsquedas verticales en Excel, en donde buscamos un valor en común entre dos tablas y lo pegamos en una de ellas.

Los **joins** en R tienen una serie de posibilidades y alternativas que mejoran la funcionalidad respecto a las búsquedas en Excel. En primer lugar, nos permite traer todas las columnas de la segunda tabla en una sola operación (o en su caso seleccionar cuales unir). A su vez, nos permite mantener rastreabilidad sobre las columnas unidas, identificando las que pertenecen a una y otra.

En este caso, las dos tablas generadas previamente se pueden unir para tener la información en una sola, a través de la función **inner\_join()**, pero también se pudo generar la agrupación y síntesis en un solo paso. Lo anterior dependerá de la conveniencia y propósito de los procesos. Por ejemplo, puede convenir separar los resultados de tráfico para los procesos de modelación y los resultados de ingresos para análisis financieros.

```

copexa_resumen <- inner_join(copexa_tpda, copexa_ing)

## Joining, by = c("activo", "year", "A", "B", "CU", "CA1", "CA2")

```

```

copexa_resumen

## # A tibble: 0 x 7
## # Groups:   activo [0]
## # ... with 7 variables: activo <chr>, year <chr>, A <dbl>, B <dbl>, CU <dbl>,
## #   CA1 <dbl>, CA2 <dbl>

```

Como vemos, R selecciona todas las variables que coinciden para realizar la unión. En este caso, por como se generaron las tablas, todas las variables coinciden, lo cual provoca que se genere una tabla vacía. Sin embargo, tenemos control para seleccionar la base de variables de búsqueda.

```

copexa_resumen <- inner_join(copexa_tpda, copexa_ing,
                                by = c("activo", "year"))

copexa_resumen

```

```

## # A tibble: 18 x 12
## # Groups:   activo [2]
##   activo year    A.x    B.x    CU.x  CA1.x  CA2.x    A.y    B.y    CU.y  CA1.y  CA2.y
##   <chr>  <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 Autop~ 2013  4031.  352.   304.   229.   168.  1.27e8 1.16e7 1.10e7 9.74e6 1.42e7
## 2 Autop~ 2014  4295.  412.   332.   313.   244.  1.43e8 1.44e7 1.28e7 1.41e7 2.19e7
## 3 Autop~ 2015  4775.  427.   373.   375.   252.  1.64e8 1.55e7 1.48e7 1.73e7 2.34e7
## 4 Autop~ 2016  4749.  437.   399.   394.   233.  1.76e8 1.72e7 1.72e7 1.99e7 2.37e7
## 5 Autop~ 2017  4480.  427.   398.   439.   313.  1.96e8 2.00e7 2.04e7 2.67e7 3.83e7
## 6 Autop~ 2018  4603.  447.   459.   602.   389.  2.06e8 2.34e7 2.51e7 4.24e7 5.48e7
## 7 Autop~ 2019  4359.  444.   526.   769.   517.  2.04e8 2.55e7 3.01e7 5.77e7 7.65e7
## 8 Autop~ 2020  3383.  192.   528.   777.   523.  1.59e8 1.11e7 3.11e7 6.27e7 8.31e7
## 9 Autop~ 2021  4374.  204.   586.   889.   616.  2.11e8 1.21e7 3.55e7 7.85e7 9.92e7
## 10 Libra~ 2013 1237.   97.1  125.   329.   337.  1.27e8 1.16e7 1.10e7 9.74e6 1.42e7
## 11 Libra~ 2014 1348.   152.   116.   389.   353.  5.29e7 6.20e6 8.61e6 3.70e7 4.10e7
## 12 Libra~ 2015 1659.   161.   138.   450.   331.  6.72e7 6.79e6 1.06e7 4.42e7 3.96e7
## 13 Libra~ 2016 1745.   171.   171.   464.   293.  7.40e7 7.60e6 1.38e7 4.76e7 3.68e7
## 14 Libra~ 2017 1685.   156.   221.   481.   369.  7.36e7 7.14e6 1.83e7 5.08e7 4.77e7
## 15 Libra~ 2018 1851.   163.   262.   586.   430.  8.29e7 8.20e6 2.32e7 6.64e7 6.09e7
## 16 Libra~ 2019 1760.   162.   345.   755.   583.  8.22e7 9.02e6 3.08e7 8.71e7 8.97e7
## 17 Libra~ 2020 1330.   65.2  357.   796.   602.  6.24e7 3.64e6 3.20e7 9.61e7 9.71e7
## 18 Libra~ 2021 1806.   69.9  422.   928.   720.  8.71e7 4.02e6 3.89e7 1.16e8 1.18e8

```

El resultado anterior genera una tabla que verifica el activo y el año y después realiza la unión de todos los datos entre ambas tablas que coinciden con esa búsqueda, como las variables tienen el mismo nombre (aunque diferente contenido) en las dos tablas, R asigna sufijos que en este caso con “x” y “y”, a la primera y segunda tabla, respectivamente.

Dentro de la función `inner_join()`, además del argumento `by`, tenemos el argumento `suffix`, con el cual podemos definir manualmente los sufijos que R utilizará para identificar la información de cada tabla, en este caso podemos utilizar el sufijo `_tpda` para la tabla de aforos y `_ingreso` para la tabla de ingresos.

```

copexa_resumen <- inner_join(copexa_tpda, copexa_ing,
                                by = c("activo", "year"),
                                suffix = c("_tpda", "_ingreso"))

copexa_resumen

```

```

## # A tibble: 18 x 12
## # Groups:   activo [2]
##   activo      year    A_tpda    B_tpda    CU_tpda  CA1_tpda  CA2_tpda  A_ingreso  B_ingreso
##   <chr>     <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 Autopista ~ 2013  4031.  352.   304.   229.   168.  1.27e8 11634634.
## 2 Autopista ~ 2014  4295.  412.   332.   313.   244.  1.43e8 14376535.
## 3 Autopista ~ 2015  4775.  427.   373.   375.   252.  1.64e8 15450473.
## 4 Autopista ~ 2016  4749.  437.   399.   394.   233.  1.76e8 17188521.
## 5 Autopista ~ 2017  4480.  427.   398.   439.   313.  1.96e8 20013526.
## 6 Autopista ~ 2018  4603.  447.   459.   602.   389.  2.06e8 23385021.
## 7 Autopista ~ 2019  4359.  444.   526.   769.   517.  2.04e8 25506743.
## 8 Autopista ~ 2020  3383.  192.   528.   777.   523.  1.59e8 11086645.
## 9 Autopista ~ 2021  4374.  204.   586.   889.   616.  2.11e8 12132256.
## 10 Libramient~ 2013 1237.   97.1  125.   329.   337.  1.27e8 11634634.
## 11 Libramient~ 2014 1348.   152.   116.   389.   353.  5.29e7 6197678.
## 12 Libramient~ 2015 1659.   161.   138.   450.   331.  6.72e7 6792076.

```

```

## 13 Libramient~ 2016    1745.   171.     171.     464.     293.    7.40e7  7601022.
## 14 Libramient~ 2017    1685.   156.     221.     481.     369.    7.36e7  7139500
## 15 Libramient~ 2018    1851.   163.     262.     586.     430.    8.29e7  8203865.
## 16 Libramient~ 2019    1760.   162.     345.     755.     583.    8.22e7  9022023.
## 17 Libramient~ 2020    1330.   65.2     357.     796.     602.    6.24e7  3640402.
## 18 Libramient~ 2021    1806.   69.9     422.     928.     720.    8.71e7  4018966.
## # ... with 3 more variables: CU_ingreso <dbl>, CA1_ingreso <dbl>,
## #   CA2_ingreso <dbl>

```

### 6.3 Uniones izquierda y derecha

Las uniones internas, a través de la función `inner_join()` devuelven solamente las observaciones con coincidencias en ambas tablas. En el caso del ejemplo utilizado, sabemos que ambas tablas tendrían coincidencias para todos los valores, por lo que no se perdió información.

Sin embargo, es posible requerir búsquedas que devuelvan valores faltantes, para lo cual podemos utilizar otros tipos de uniones.

La función `left_join()` mantiene todas las variables de la primer tabla, que correspondería al argumento `x`, y devuelve los valores de la tabla `y` que coincidan, colocando valores faltantes o `NA` en las observaciones no encontradas. La función `right_join()` hace lo mismo que la anterior, pero manteniendo todos los valores de `y`. Así, podemos decir que:

```
left_join(x, y) = right_join(y, x)
```

### 6.4 Otras uniones

Finalmente, otras funciones para unión de tablas que son muy útiles son `full_join()` y `anti_join()`.

En el caso de la primera, `ful_join()`, permite mantener todos los valores de ambas tablas, y colocar `NAs` en la tabla `x` cuando la búsqueda aparezca solo en `y`, así como en la tabla `y` cuando la búsqueda aparezca solo en `x`, mientras que todos los argumentos coincidentes se mantienen. En resumen, es como la combinación de `inner_join()`, `left_join()` y `right_join()`.

Para la segunda, `anti_join()`, R encontrará los elementos de la primera tabla que no aparezcan en la segunda.

## 7 Visualización de datos

Como vimos brevemente en capítulos previos, la visualización es parte importante del análisis de datos. Plasmar la información de manera visual permite a los analistas identificar patrones en la información que pueden indicar qué acciones tomar con los datos para procesarlos y generar resultados que añadan valor.

### 7.1 Gramática de los gráficos

La base de la generación de visualizaciones en *ggplot2* es la gramática de los gráficos. Una especie de sub-lenguaje que se desarrolló para definir los diversos componentes de un gráfico. Los componentes de las visualizaciones realizadas con *ggplot2* se encuentran en capas, en donde cada una alberga diferentes elementos que le indican a R qué dibujar y cómo dibujarlo.

Los tres principales elementos de una visualización realizada con *ggplot2* son los datos, las estéticas o mapeo y las geometrías. Además, existen otros elementos que pueden ayudar a complementar el gráfico. En este curso, veremos los primeros tres, así como una introducción a los temas.

Elemento	Descripción
Data	Los datos a graficar
Aesthetics	Las escalas en las cuales mapeamos los datos
Geometries	Los elementos visuales de los gráficos

Elemento	Descripción
Themes	La “tinta” no relacionada con los datos
Statistics	Representaciones de los datos para ayudar la comprensión
Coordinates	El espacio en el cual se realizará la visualización
Facets	Sirve para graficar pequeños múltiples

Como vimos brevemente, la estructura de una visualización en *ggplot2* es como se presenta a continuación: (Por cierto, el nombre de **ggplot** viene de “grammar of graphics”)

```
ggplot(data, aes(x = map_x, y = map_y)) +  
  geom_point()
```

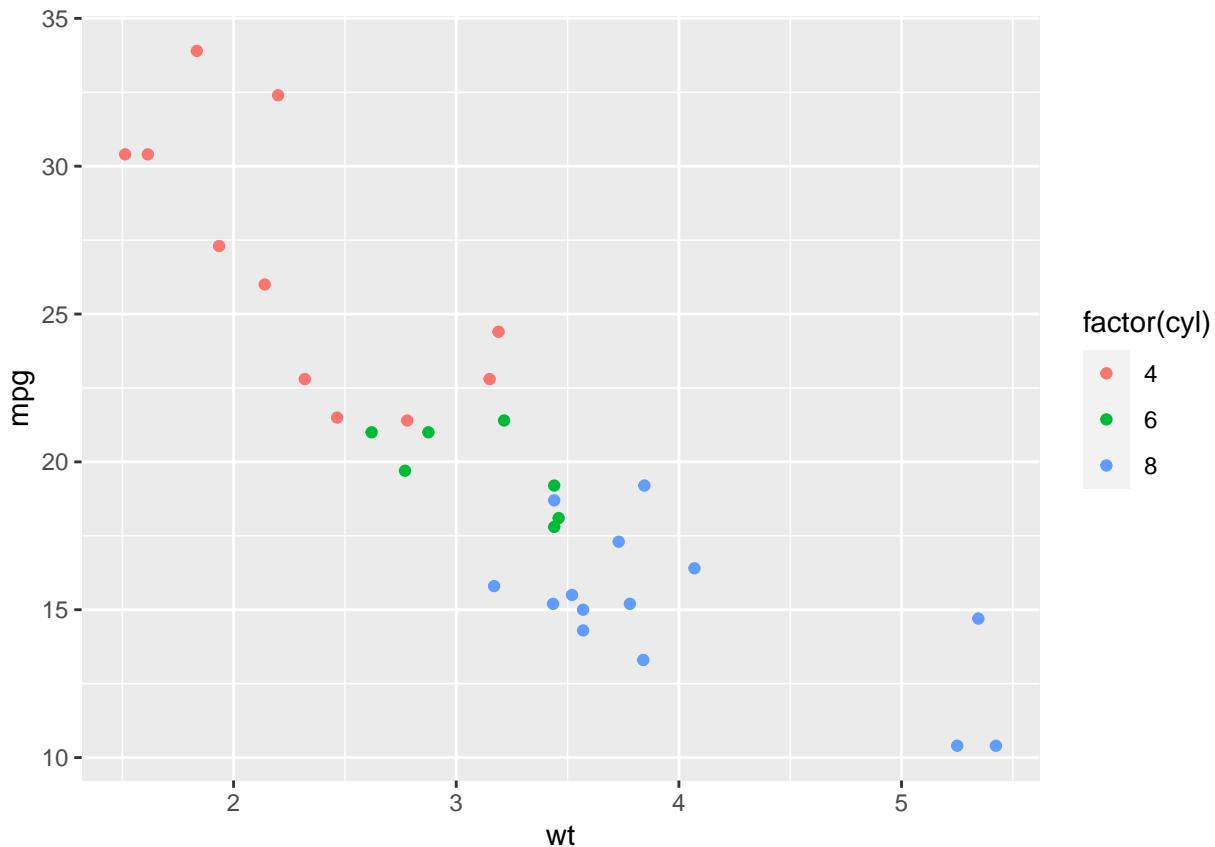
En el anterior bloque, tenemos la llamada a la función **ggplot()**, en la cual alimentamos el objeto con los datos que contienen las variables que vamos a graficar, seguido por las variables a mapear. Estas variables se incluyen dentro del argumento **aes()** (aesthetics). Después con el operador **+** agregamos la geometría, que debe adaptarse a los datos a graficar.

En este punto, cabe mencionar que existen varios tipos de geometrías, que pueden ser puntos, líneas, barras, boxplots, etc. En el siguiente enlace se encuentran las principales geometrías con las que trabaja *ggplot2* y sus parámetros.

Los parámetros son modificadores que pueden trabajar con valores o incluso con variables, y ayudan a generar visualizaciones más explicativas. Por ejemplo, el parámetro **color** puede recibir el nombre de una variable de tipo factor y dibujar de diferente color cada punto de acuerdo con su categoría. Por otra parte, si se mapea una variable numérica a este parámetro, se genera una gama de colores continua.

A continuación vemos un ejemplo clásico, utilizando la base de datos **mtcars** incluida en R, que describe características de automóviles como millas por galón, tamaño del motor, número de velocidades, etc. Para aprovechar, utilizamos la sintaxis de dplyr con el operador **%>%**.

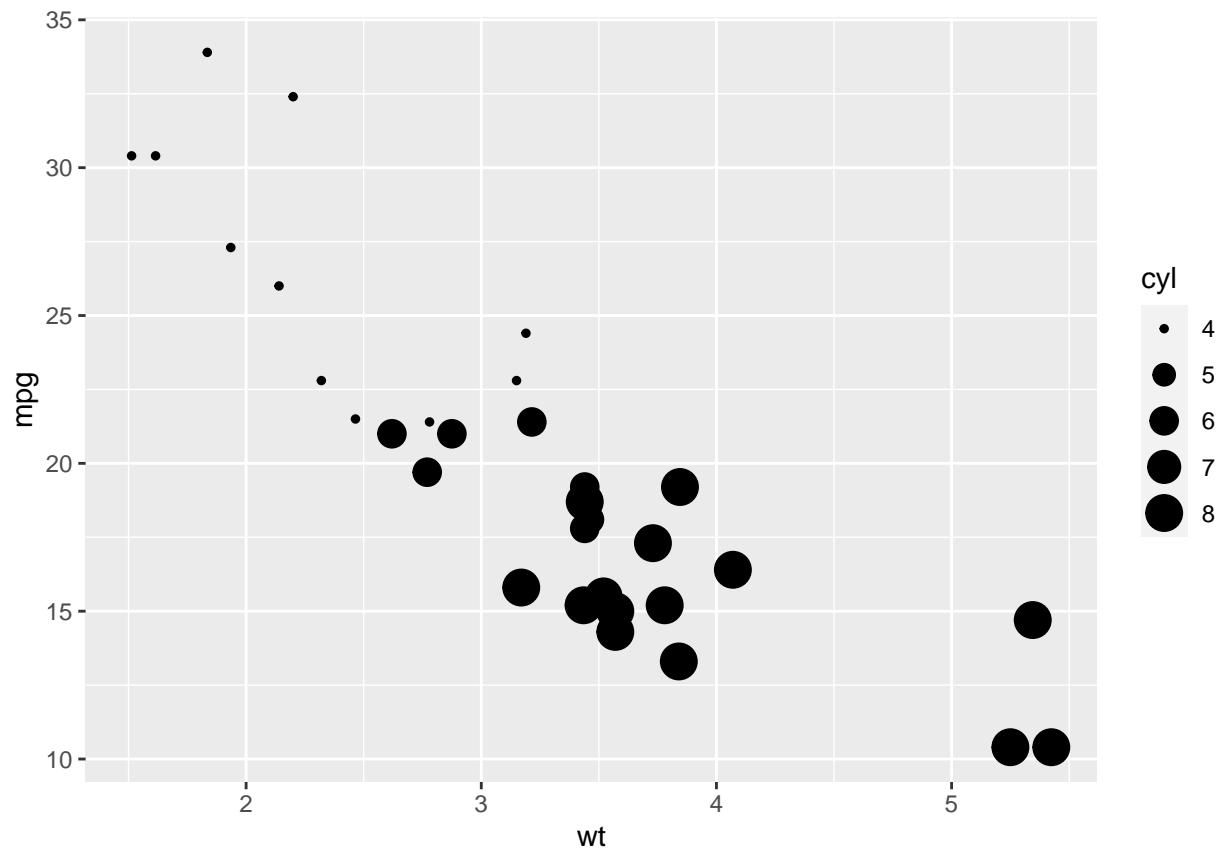
```
mtcars %>%
  ggplot(aes(wt, mpg, color = factor(cyl))) +
  geom_point()
```



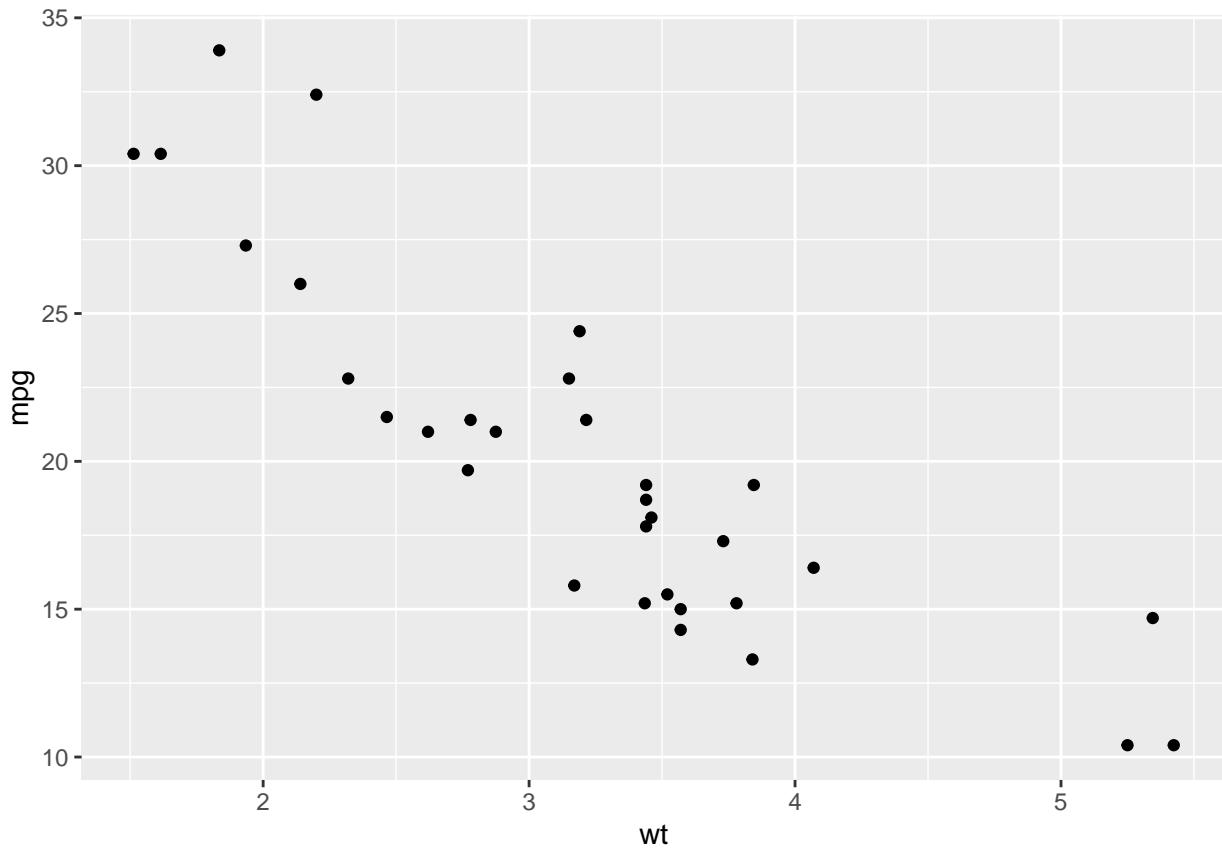
## 7.2 Estéticas

El elemento conocido como aesthetics funciona para definir una serie de parámetros visuales a los cuales se les asignará una variable. En el caso más elemental de un gráfico, mapeamos o asignamos una variable a la estética **x** y otra variable a la estética **y**. Sin embargo, como vimos, es posible asignar variables a otras estéticas visibles como color, relleno (fill), tamaño (size), forma (shape), transparencia (alpha), entre otras. Muchas geometrías comparten nombres de estéticas visibles con sus parámetros, por lo que puede ser fácil confundirse al momento de definirlos. La diferencia la hace en donde colocamos la definición del argumento, dentro o fuera del argumento **aes()**, como vemos en el siguiente ejemplo.

```
mtcars %>%
  ggplot(aes(wt, mpg, size = cyl)) +
  geom_point()
```

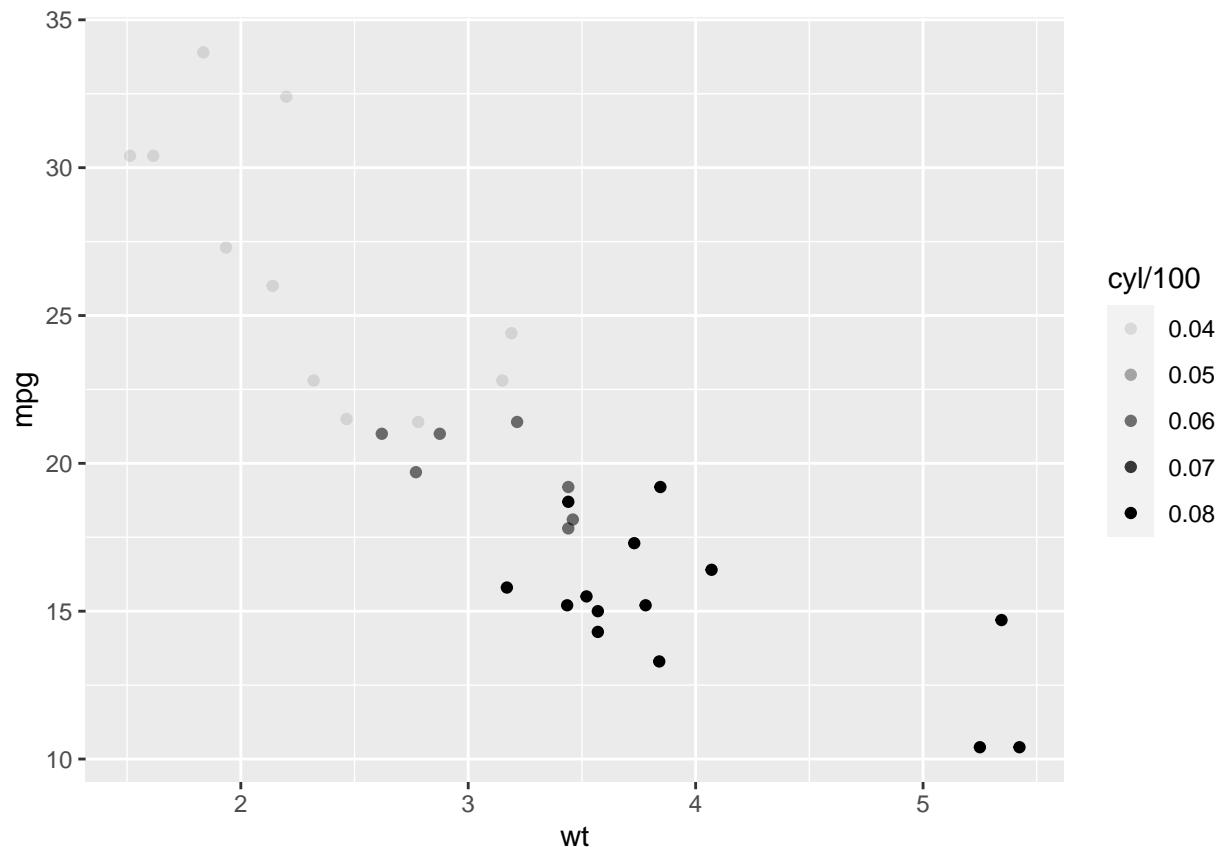


```
mtcars %>%
  ggplot(aes(wt, mpg), size = cyl) +
  geom_point()
```

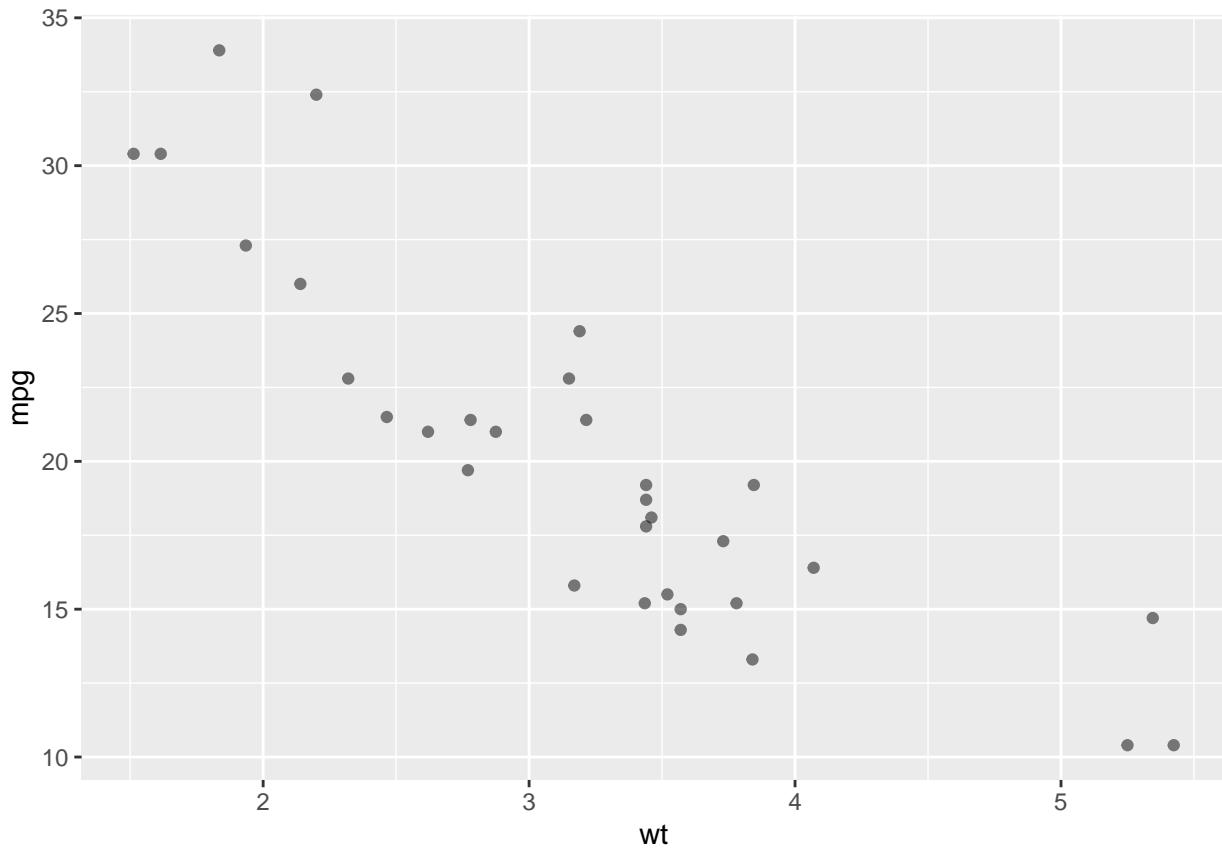


Como vemos, el segundo caso no produjo un cambio en el tamaño de los puntos de acuerdo con el cilindraje de los vehículos, ya que al estar fuera del argumento **aes()**, no puede mapear los valores de la variable. Sin embargo, es posible cambiar el parámetro para todos los puntos asignando un valor. Por ejemplo, utilizando el parámetro **alpha**, podemos tener un valor fijo o variable, de acuerdo en donde coloquemos el argumento.

```
mtcars %>%
  ggplot(aes(wt, mpg, alpha = cyl/100)) +
  geom_point()
```



```
mtcars %>%
  ggplot(aes(wt, mpg)) +
  geom_point(alpha = 0.5)
```



De ahora en adelante, veremos que otra convención es colocar los elementos de la geometría dentro del argumento de la misma, lo que nos permite mapear diferentes variables en caso de utilizar varias geometrías, como se muestra en el siguiente ejemplo:

```
mtcars %>%
  ggplot() +
  geom_point(aes(wt, mpg, color = factor(cyl)))
```

### 7.3 Geometrías

Las geometrías en esencia definen como se verá el gráfico. Como se mencionó anteriormente, existen actualmente alrededor de 50 geometrías, y se puede acceder a cada una de ellas utilizando el prefijo **geom\_** correspondiente.

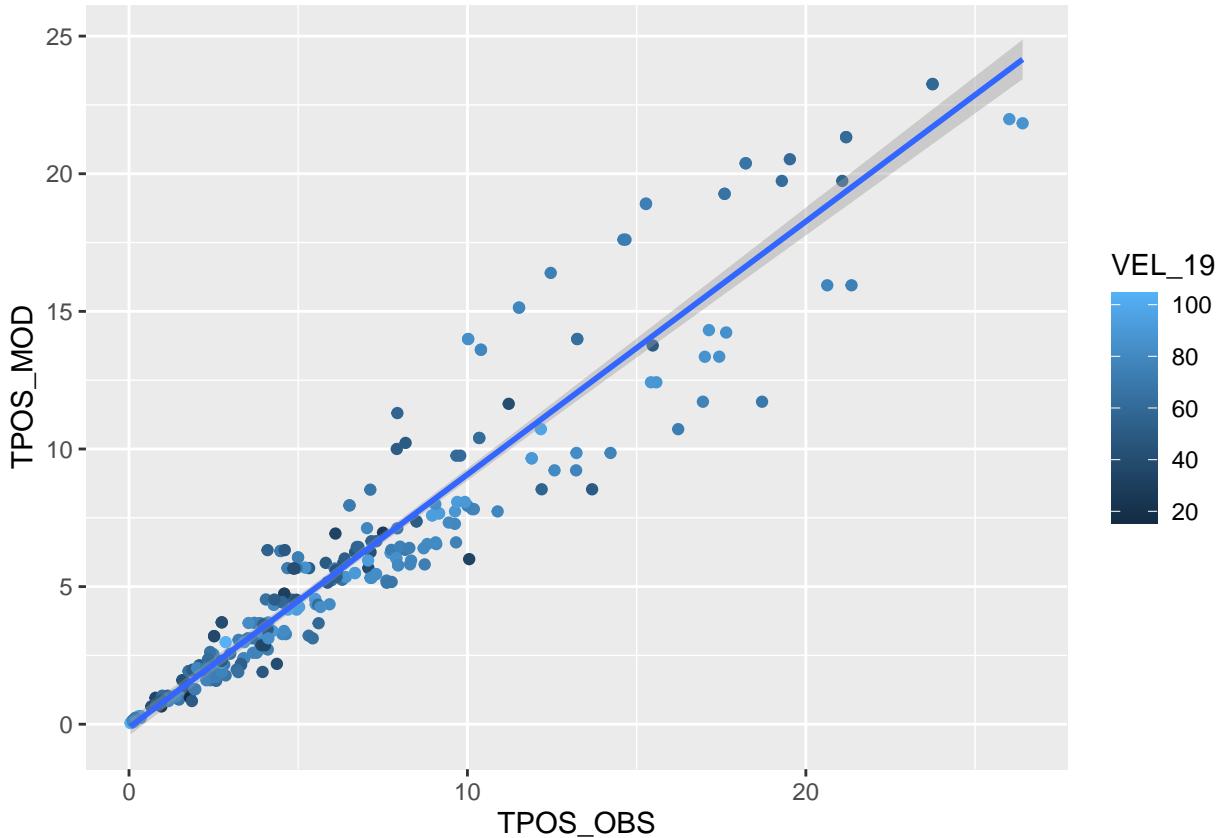
Durante el curso, hemos utilizado diversas geometrías, incluso establecimos la base de muchas de ellas.

La geometría que puede considerarse básica es **geom\_point()**, que ya vimos. Sin embargo, en este momento podemos introducir una base de datos relacionada con los resultados de un modelo de transporte realizado en Visum. Para validar la calibración del modelo, realizamos el análisis de tiempos observados contra tiempos modelados y analizamos la R resultante del ajuste lineal de los datos.

```
tiempos_calibracion <- read_excel("data/tiempos_calibracion.xlsx")

tiempos_calibracion %>%
  ggplot(aes(TPOS_OBS, TPOS_MOD)) +
  geom_point(aes(color = VEL_19)) +
  geom_smooth(method = "lm")
```

```
## `geom_smooth()` using formula 'y ~ x'
```



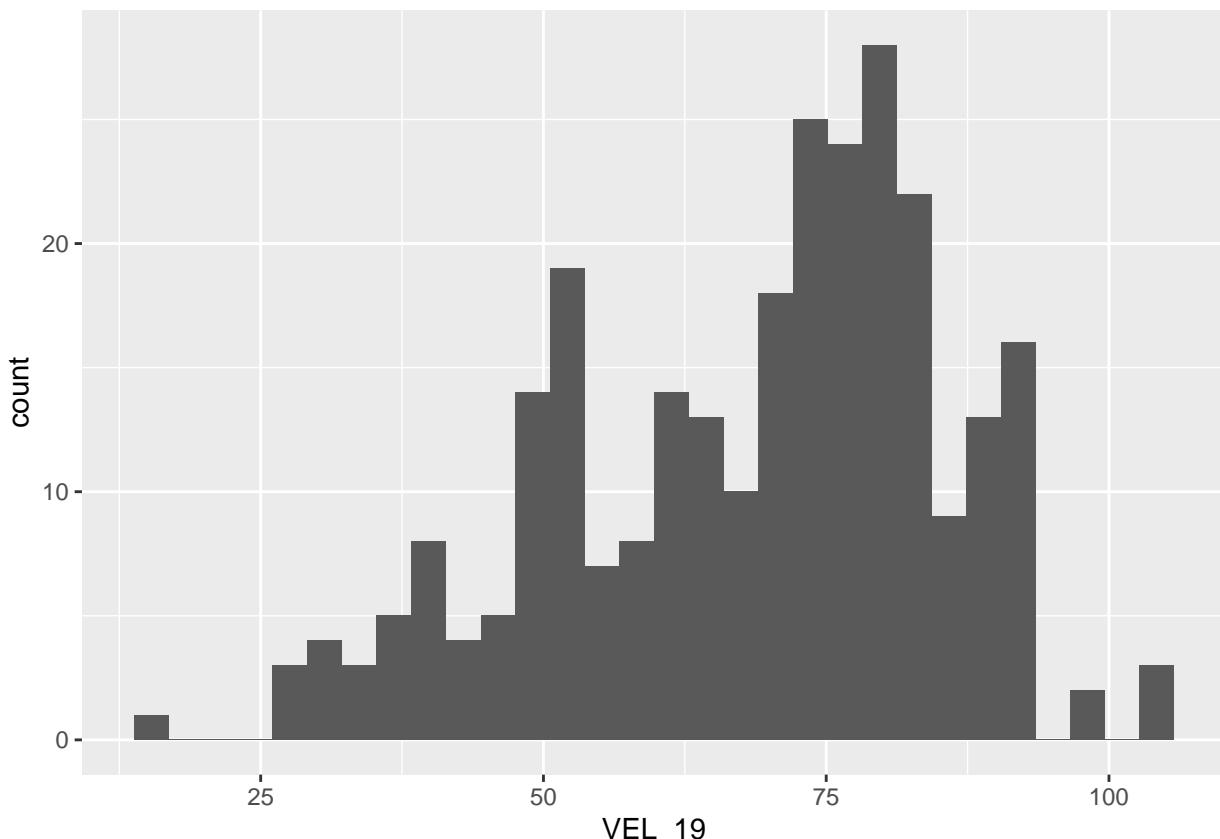
Aquí, agregamos la geometría `geom_smooth()` con el argumento `method = "lm"`, que calcula y dibuja la línea de regresión lineal de los datos. Por otra parte, adelantando unos capítulos del siguiente nivel de este curso, podemos extraer la R cuadrada al realizar el análisis del modelo de regresión lineal de la siguiente manera:

```
ml = lm(TPOS_OBS-TPOS_MOD, data = tiempos_calibracion)  
summary(ml)$r.squared
```

```
## [1] 0.9067904
```

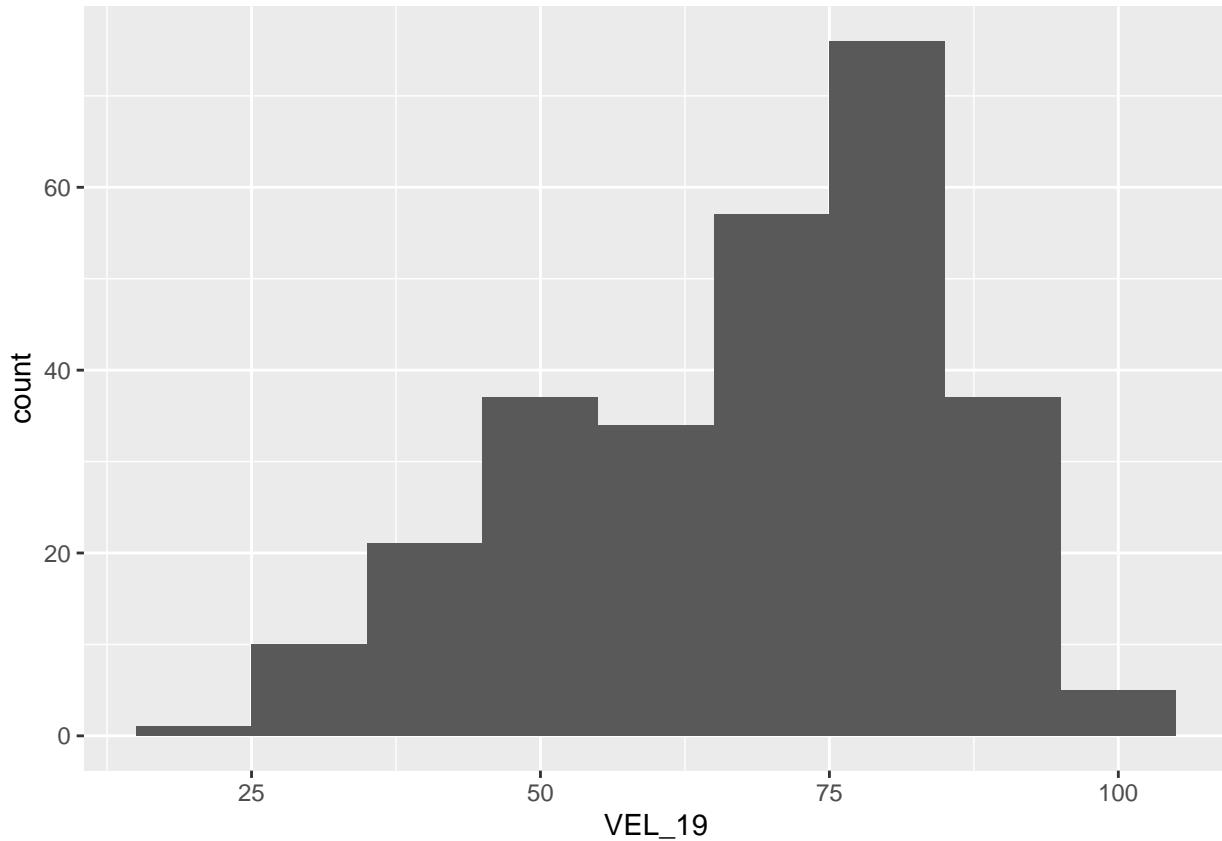
Otra visualización importante son los histogramas, para el caso de la base de datos de tiempos observados y modelados, tenemos una variable que refiere a la velocidad de la vía. Podemos generar un histograma de velocidades que nos indique la distribución de las mismas en la red.

```
tiempos_calibracion %>%  
  ggplot() +  
  geom_histogram(aes(VEL_19))  
  
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



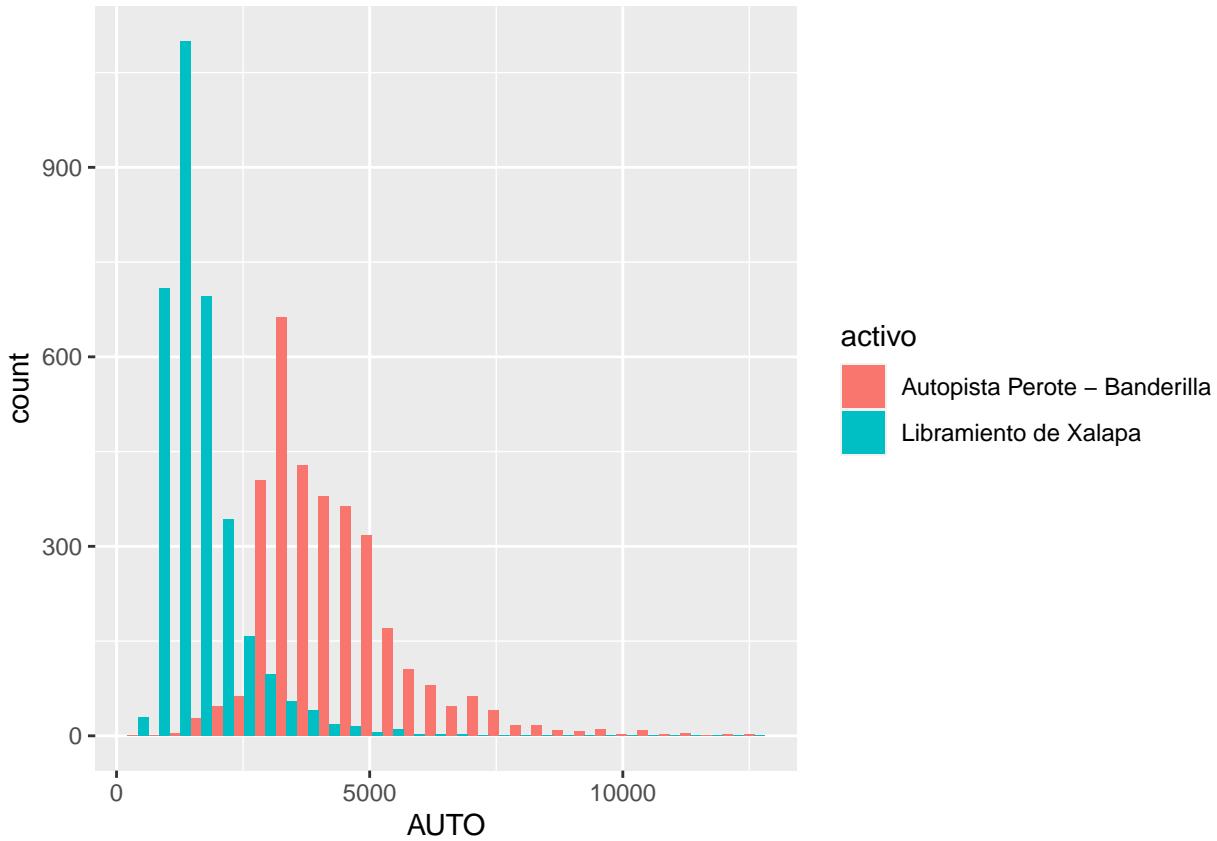
Como vemos, R selecciona el número de “cajas” a utilizar, que viene predefinido en 30. Sin embargo, podemos modificar este número al definirlo manualmente, o declarando el rango de cada una de estas cajas. La intención es que no queden espacios en blanco, ya que esto significa que no tenemos datos para ese rango específico de velocidad. Con esto vemos que las velocidades se concentran entre los 75 y 80 km/h.

```
tiempos_calibracion %>%
  ggplot(aes(VEL_19)) +
  geom_histogram(binwidth = 10)
```



Por otra parte, podemos utilizar algunos parámetros que aplican para este tipo de visualización, que hace transformaciones estadísticas en los datos (en este caso un conteo). Utilizando la base de datos de aforos del proyecto Corredor Perote - Xalapa que vimos previamente, generamos el histograma de volúmenes de autos. A su vez, al definir la estética **fill** con la variable **activo**, podemos generar dentro de la misma visualización, dos histogramas independientes, y con el parámetro **position = "dodge"** le indicamos a R que dibuje las barras separadas.

```
copexa %>%
  ggplot(aes(AUTO, fill = activo)) +
  geom_histogram(position = "dodge")
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

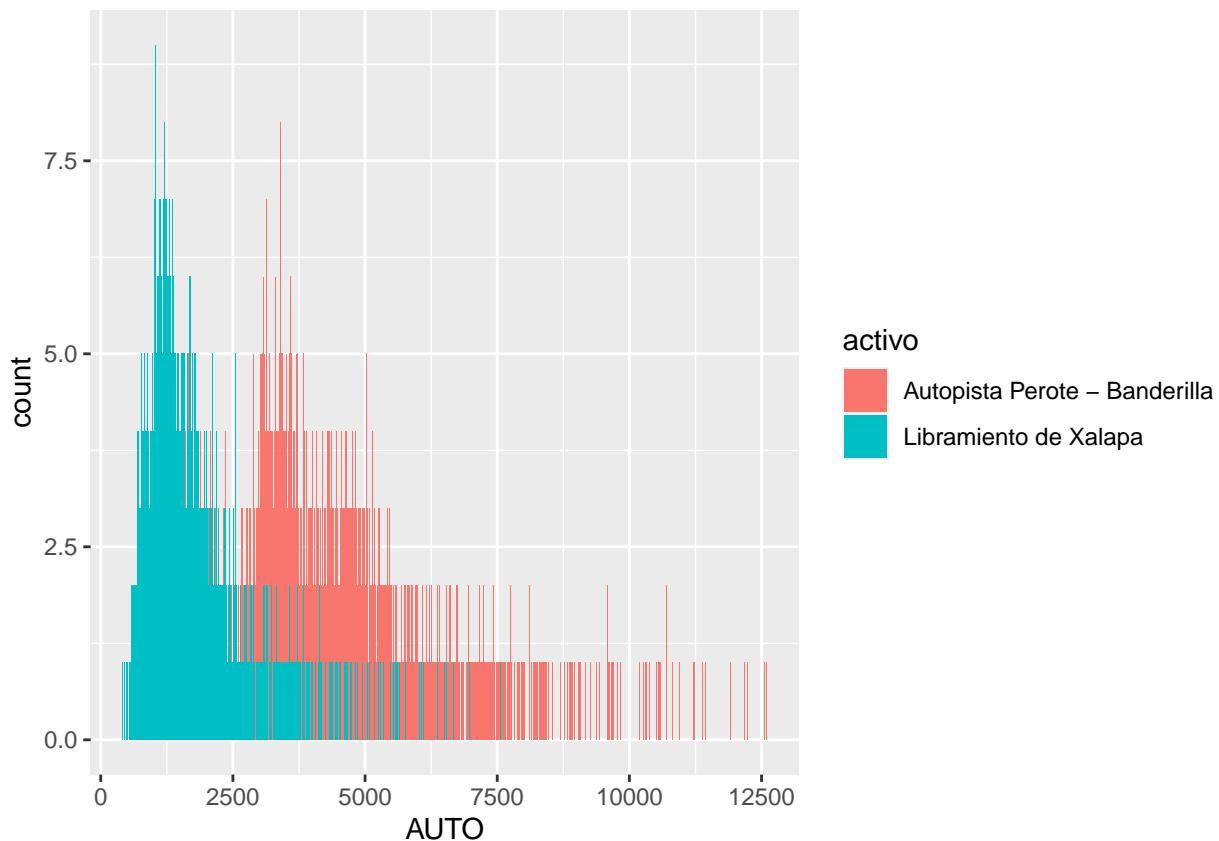


En

este caso el gráfico nos dice el conteo de días que presentaron cierto número de vehículos. Se observa que en el Libramiento de Xalapa el aforo está concentrado alrededor de los 1200 -1500 vehículos, mientras que en la Autopista Perote - Banderilla se concentra cerca de los 3000 vehículos. Esto es consistente con el análisis de TPDA que realizamos previamente. Sin embargo, sabemos que estos datos crecen con el paso del tiempo, por lo que también podríamos agregar otra variable (año) y mapearla a otra estética visible.

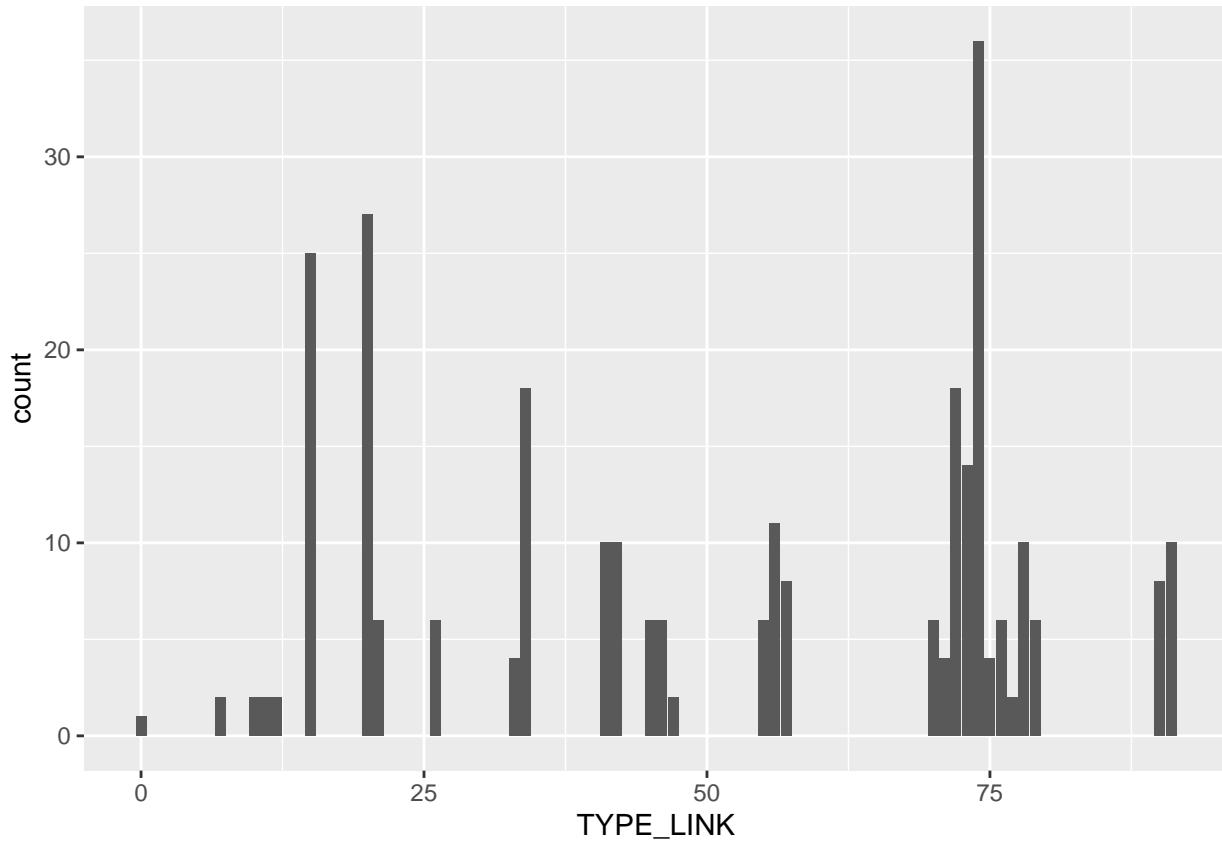
Los histogramas a su vez, dentro de la lógica de R, son una variante de la geometría **geom\_bar()**, que es un gráfico de barras con conteos, aunque también existe **geom\_col()**, que presenta un gráfico de barras con los datos actuales de las variables. El gráfico de barras, a diferencia del histograma, no agrupa en “cajas” por defecto y es mucho más efectivo cuando la variable mapeada a x es categórica. El mismo gráfico que realizamos en el histograma replicado con **geom\_bar()** resulta como sigue:

```
copexa %>%
  ggplot(aes(AUTO, fill = activo)) +
  geom_bar()
```



Por lo que este tipo de gráfico funciona mejor para mostrar conteos de variables categóricas. Utilizando el data frame de tiempos de calibración, podemos generar un conteo de los tipos de enlaces que se presentan en el modelo.

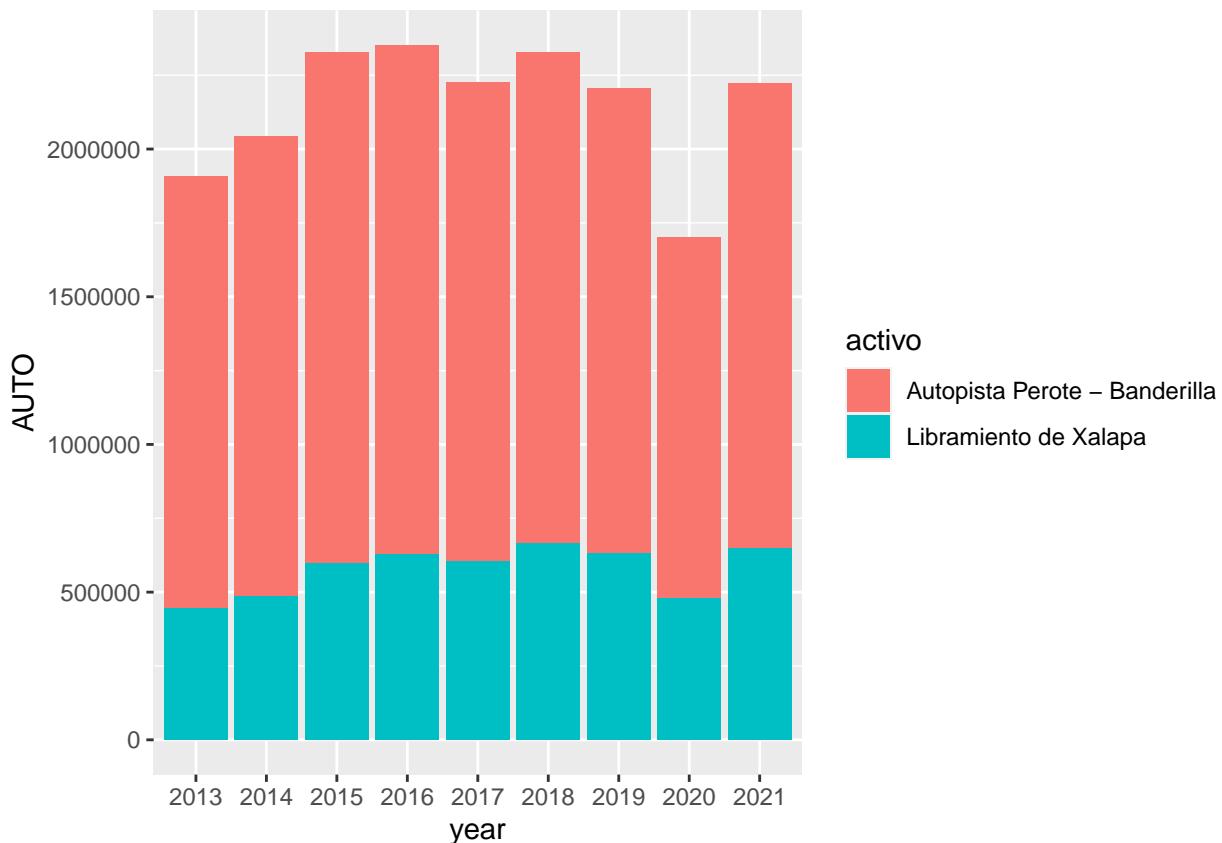
```
tiempos_calibracion %>%
  ggplot(aes(TYPE_LINK)) +
  geom_bar()
```



Lo anterior también puede ser útil al analizar encuestas origen - destino, ya que podríamos contabilizar las respuestas de una categoría, por ejemplo el motivo del viaje.

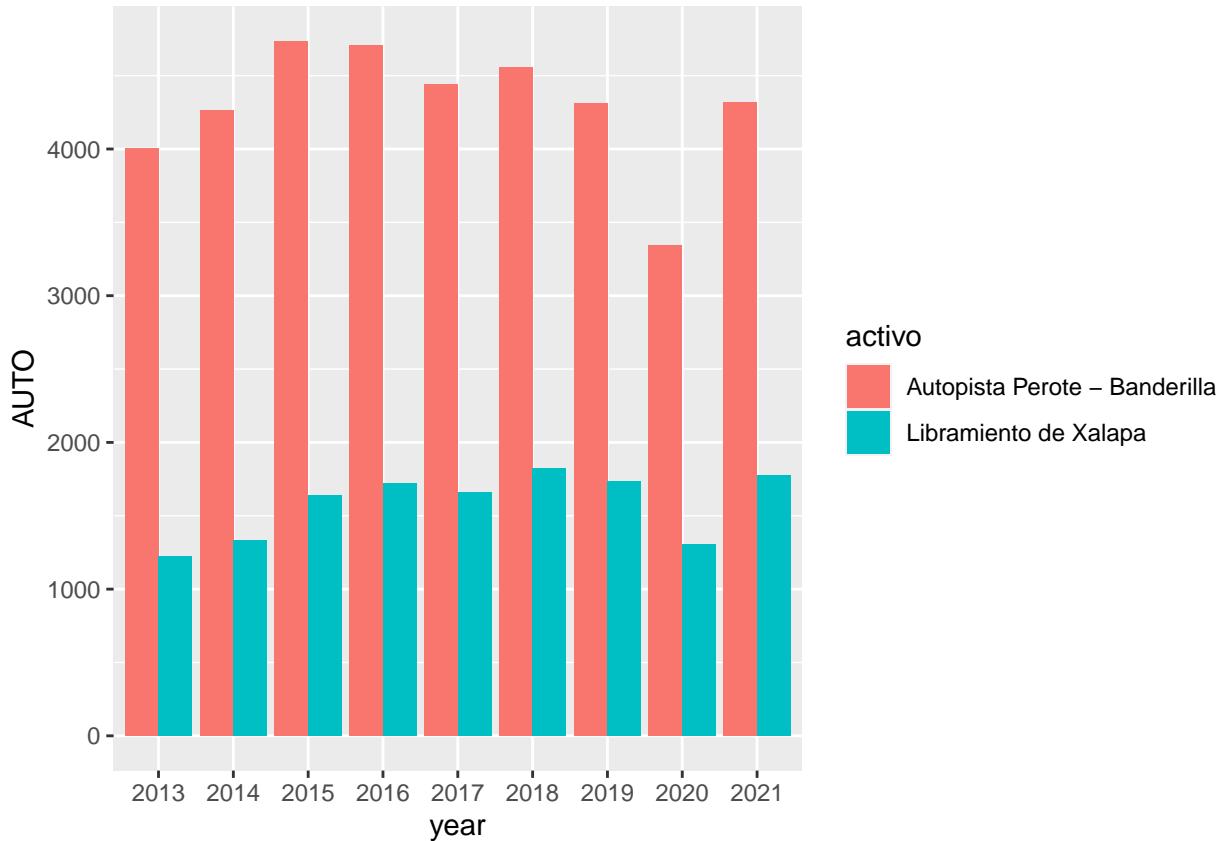
Finalmente, la geometría `geom_col()` nos permite visualizar los valores contenidos dentro de una variable, también utilizando una variable categórica en el eje x. Este ejercicio lo hicimos previamente en el censo, y podemos repetirlo con la información de COPEXA.

```
copexa %>%
  mutate(year = format(fecha, format = "%Y")) %>%
  ggplot(aes(year, AUTO, fill = activo)) +
  geom_col()
```



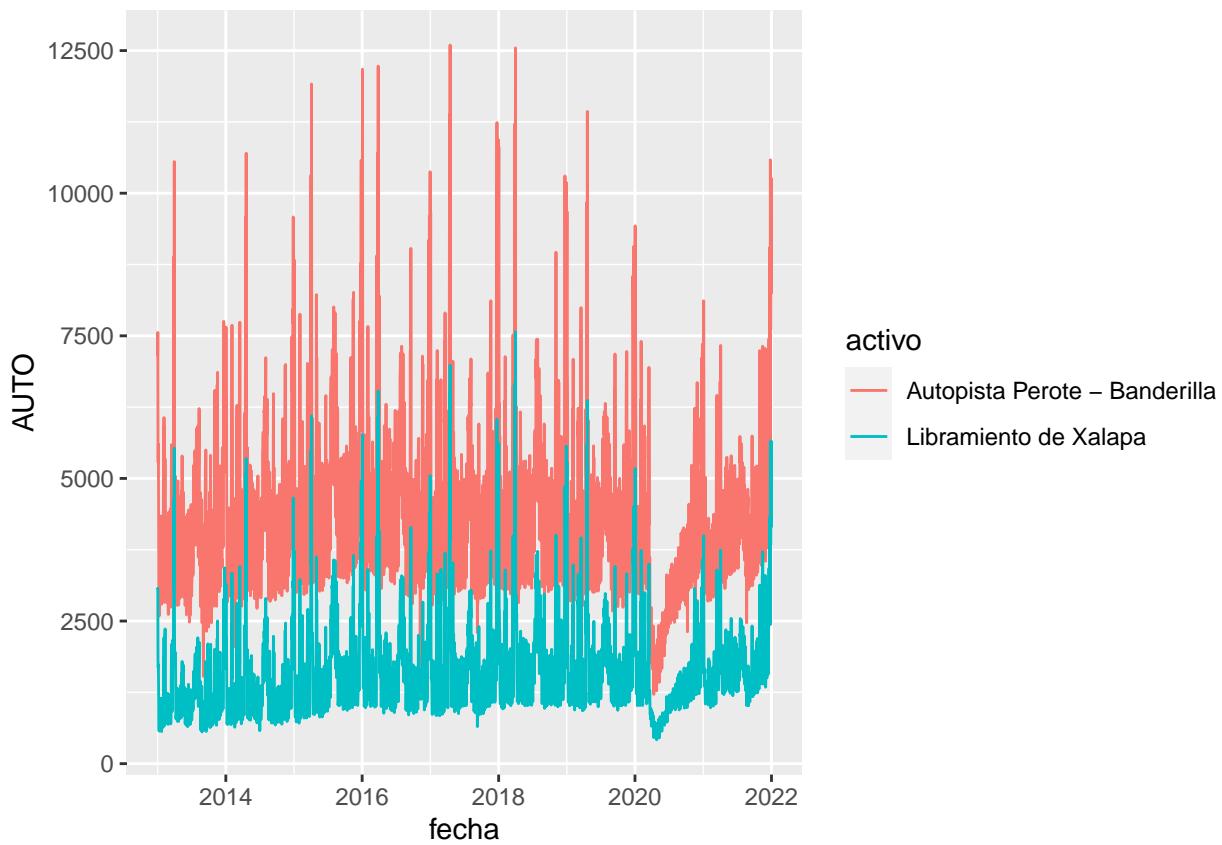
De esta forma, vemos que previamente utilizamos las funciones `group_by()` y `summarise()` para generar la tabla que contenía estos datos, esto es importante, ya que `geom_col()` devuelve la suma de todos los datos, que corresponden al volumen total anual de cada activo. Para obtener las barras por TPDA, utilizamos:

```
copexa %>%
  mutate(year = format(fecha, format = "%Y")) %>%
  group_by(activo, year) %>%
  summarise(AUTO = sum(AUTO)/n()) %>%
  ggplot(aes(year, AUTO, fill = activo)) +
  geom_col(position = "dodge")
## `summarise()` has grouped output by 'activo'. You can override using the
## `.` argument.
```



Por otra parte, existe la geometría `geom_line()`, la cual permite representar, por ejemplo, cambios en una variable a lo largo del tiempo. Utilizando la base de datos de COPEXA sin ninguna transformación, podemos obtener la siguiente visualización:

```
copexa %>%
  ggplot(aes(fecha, AUTO)) +
  geom_line(aes(color = activo))
```

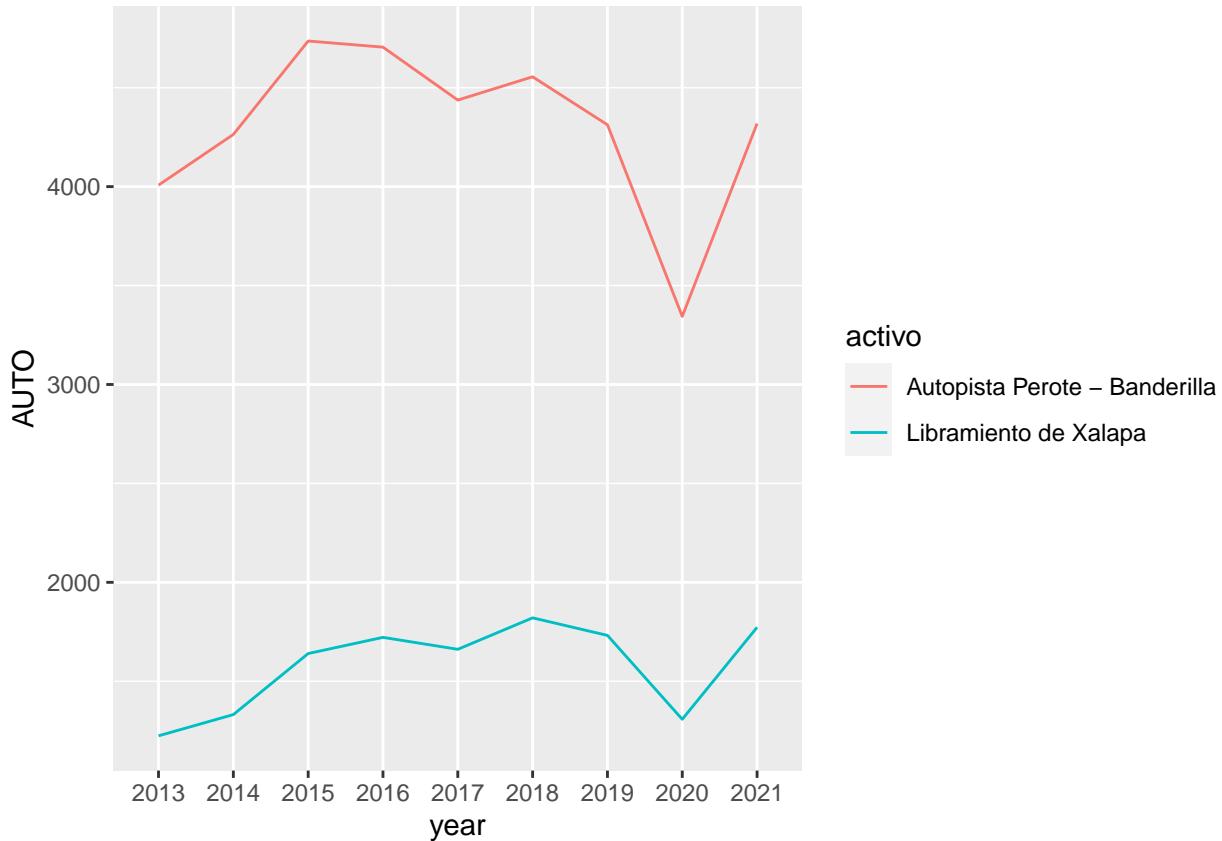


Dado que tenemos información diaria, el gráfico parece muy saturado, pero podemos observar patrones, como por ejemplo la pérdida de volumen en el año 2020 debido a la pandemia por COVID-19.

Si reducimos todo a TPDA y realizamos el gráfico de líneas, necesitamos especificar que la agrupación la queremos representar por activo a través del parámetro **group**.

```
copexa %>%
  mutate(year = format(fecha, format = "%Y")) %>%
  group_by(activo, year) %>%
  summarise(AUTO = sum(AUTO)/n()) %>%
  ggplot(aes(year, AUTO)) +
  geom_line(aes(color = activo, group = activo))
```

```
## `summarise()` has grouped output by 'activo'. You can override using the
## `.`groups` argument.
```



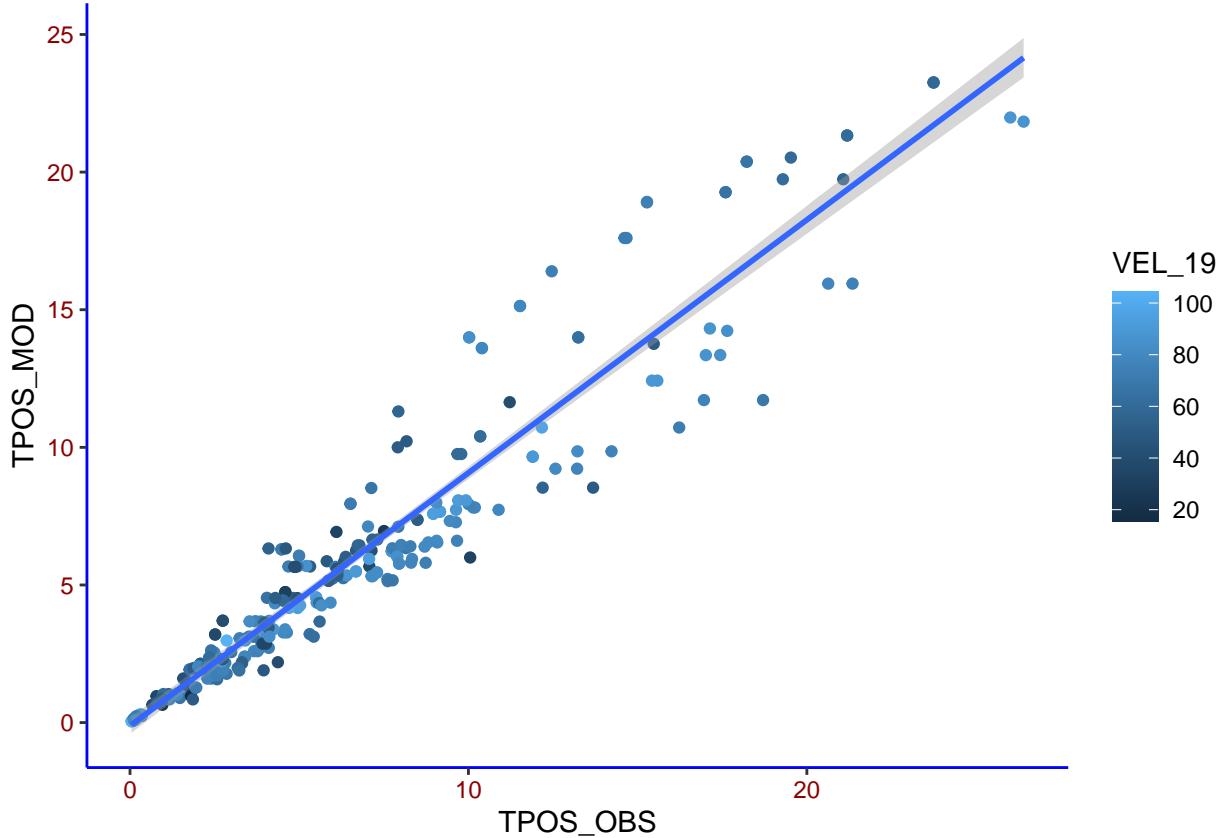
## 7.4 Temas

Los temas controlan todos los elementos visuales del gráfico que no están ligados a los datos. Estos elementos se clasifican en tres grupos: texto, línea y rectángulo, y pueden ser modificados utilizando la función adecuada, las cuales comparten el prefijo `element_`.

Volviendo a un gráfico que nos quedó decente, podemos modificarlo para mejorar su presentación. En este caso mostramos el gráfico de tiempos de calibración. Para este ejemplo, mostraremos una modificación en cada grupo de temas para después crear un tema completo que podamos añadir a nuestro gráfico.

```
tiempos_calibracion %>%
  ggplot(aes(TPOS_OBS, TPOS_MOD)) +
  geom_point(aes(color = VEL_19)) +
  geom_smooth(method = "lm") +
  theme(axis.text = element_text(color = "dark red"),
        axis.line = element_line(color = "blue"),
        panel.background = element_rect(fill = "white"))

## `geom_smooth()` using formula 'y ~ x'
```



Podemos definir un tema y asignarlo a una variable para después aplicarlo a varios gráficos.

```

tema1 <- theme(text = element_text(family = "sans", size = 12),
                 rect = element_blank(),
                 panel.grid = element_blank(),
                 title = element_text(color = "dark blue"),
                 axis.line = element_line(color = "black"))

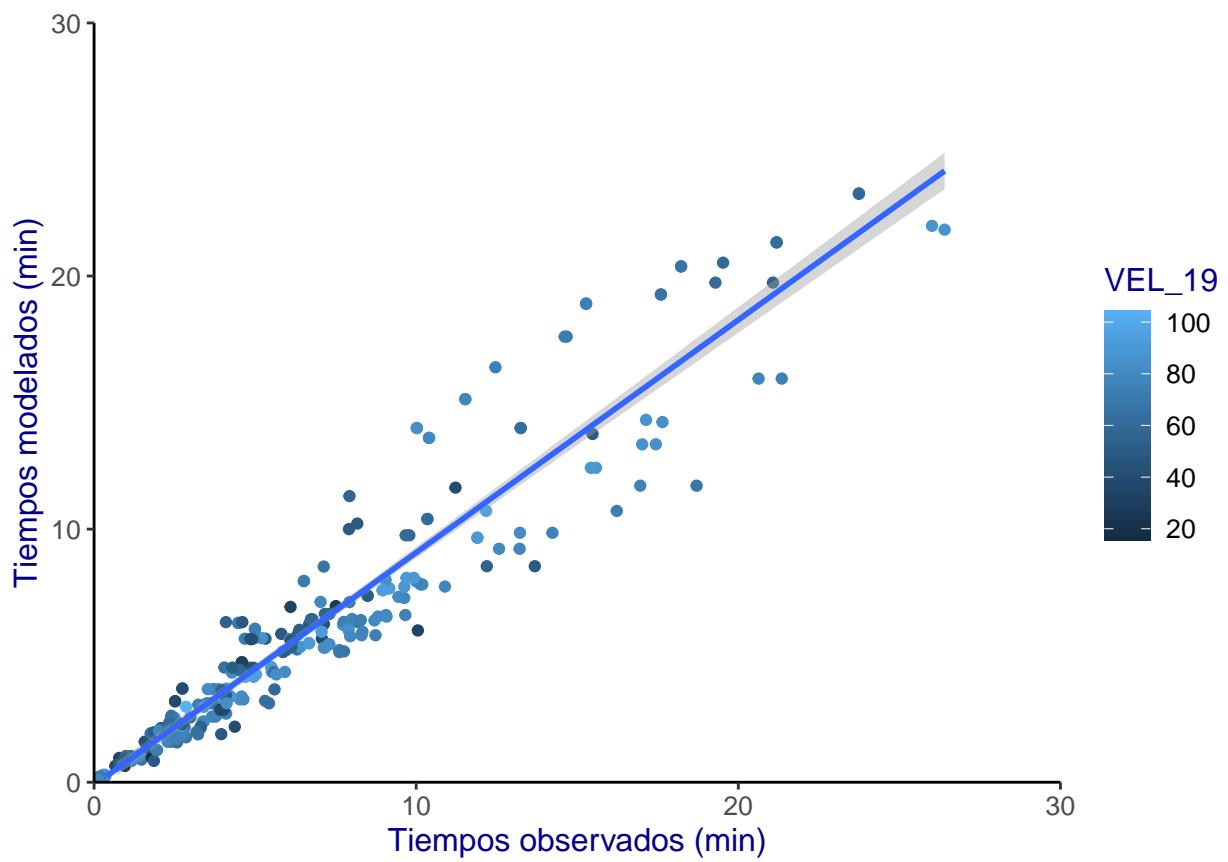
tiempos_points <- tiempos_calibracion %>%
  ggplot(aes(TPOS_OBS, TPOS_MOD)) +
  geom_point(aes(color = VEL_19)) +
  geom_smooth(method = "lm") +
  scale_x_continuous("Tiempos observados (min)",
                     limits = c(0,30),
                     expand = c(0,0)) +
  scale_y_continuous("Tiempos modelados (min)",
                     limits = c(0,30),
                     expand = c(0,0))

tiempos_points + tema1

## `geom_smooth()` using formula 'y ~ x'

## Warning: Removed 1 rows containing missing values (geom_smooth).

```

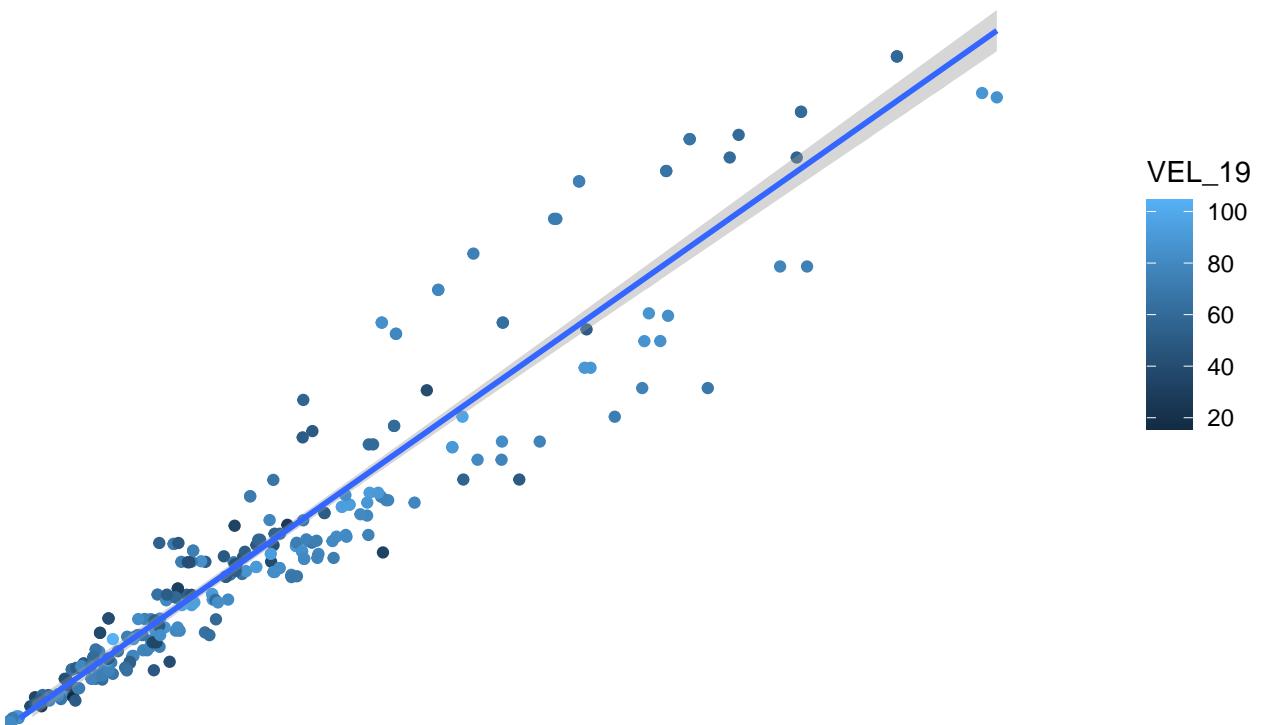


Por último, es importante mencionar que existen temas pre cargados en R, que son útiles para realizar gráficos con mejor presentación, pero incluso pueden ser de mucha utilidad al realizar análisis previos sobre los datos, ya que permiten modificar de manera rápida los elementos visuales del mismo y en algunos casos remover (o añadir) guías visuales.

```
tiempos_points + theme_void()
```

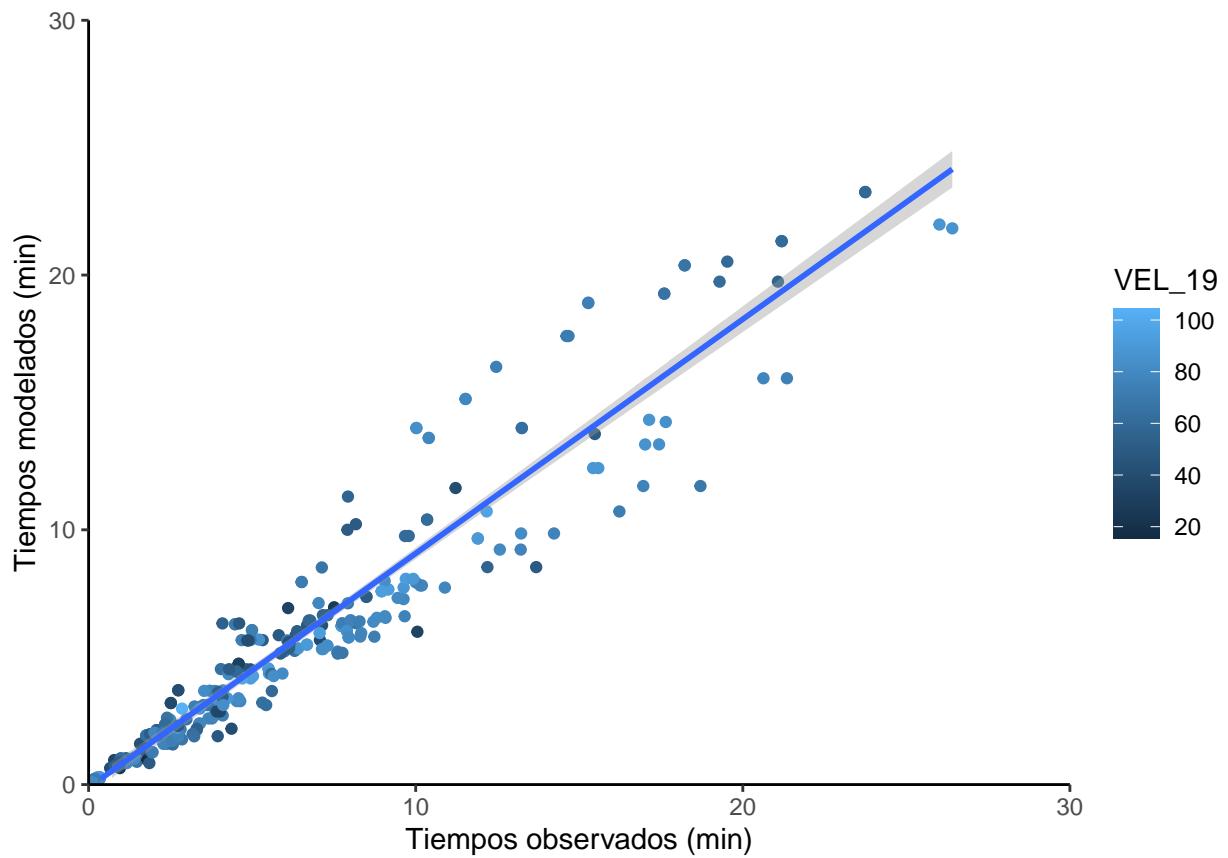
```
## `geom_smooth()` using formula 'y ~ x'

## Warning: Removed 1 rows containing missing values (geom_smooth).
```



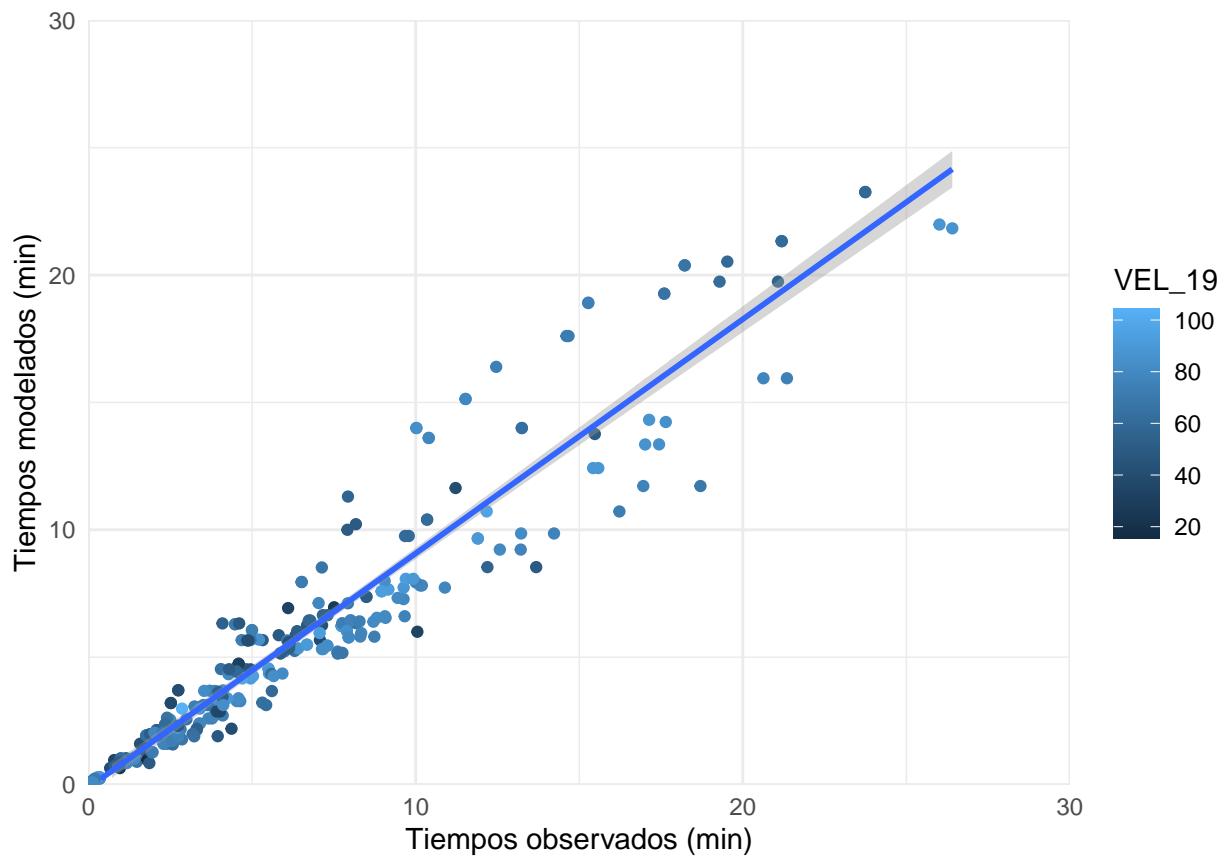
```
tiempos_points + theme_classic()
```

```
## `geom_smooth()` using formula 'y ~ x'  
## Warning: Removed 1 rows containing missing values (geom_smooth).
```



```
tiempos_points + theme_minimal()
```

```
## `geom_smooth()` using formula 'y ~ x'  
## Warning: Removed 1 rows containing missing values (geom_smooth).
```



## 8 Ejercicios finales

1. Storytelling: Elegir una base de datos de las tres utilizadas durante el curso (censo, copexa y tiempos de calibración) y generar dos gráficos significativos y auto-explicativos.
2. Tratar de reproducir el siguiente histograma (extra) Pista: revisar función `facet_wrap()`

