Introducción a R

Martín Montané

27-06-2018

Introducción

En el marco de nuestro curso vamos a precisar aprender a **programar**. Esto es así porque para realizar un proyecto de ciencia de datos necesitamos traer datos desde distintas fuentes, manipularlos, visualizarlos y analizarlos: no hay una mejor manera de hacer todas estas tareas de una manera reproducible y eficiente que a través de la programación. Sin embargo, más allá de sus **enormes ventajas**, aprender a programar tiene un costo de entrada y, como cualquier disciplina, requiere de entrenamiento.

El objetivo de este tutorial es introducirlos a los conocimientos básicos de R y RStudio, construyendo de esta manera las primeras habilidades para programar. Todo esto vamos a hacerlo mediante una investigación aplicada al mercado inmobiliario de CABA.

Nuestra investigación: el precio de las propiedades

A lo largo de nuestro curso haremos especial énfasis en la evolución del mercado inmobiliario en CABA. Más específicamente, analizaremos los movimientos de los precios en los últimos 5 años a través de un muy interesante (y masivo) dataset que públicamente ofrece *Properati*[1]. Más allá de que analizar los datos de las propiedades es de por sí interesante y pedagógico para introducir varias de las herramientas que vamos a utilizar, el mercado inmobiliario se encuentra procesando varios cambios en parámetros claves como el precio del dólar y la oferta de créditos hipotecarios. Por esta razón, los precios en los últimos dos años han experimentado un alza, o eso dicen las fuentes del sector.

En este primer tutorial vamos a hacer uso de dos datasets: uno tiene el precio promedio del metro cuadrado en USD para el período 2013-2017 desagregado a nivel de barrios y el otro contiene la misma información a nivel de comuna. Los datasets que vamos a utilizar se descargaron del portal de datos de Properati y se manipularon para que sean aptos para nuestro primer análisis. Para cuando terminen este curso, van a poder realizar todos estos pasos por su cuenta.

En este tutorial vamos a intentar responder dos preguntas:

1. ¿Aumentaron los precios de los inmuebles (medido como USD en m2) en el 2017?
2. ¿Todos los barrios aumentaron en la misma proporción o existen heterogeneidades?

Creando nuestro primer proyecto

Si no pensamos en la organización para un proyecto de ciencia de datos, el resultado puede ser un conjunto de archivos con nombres poco intuitivos en las ubicaciones más variadas dentro de nuestra computadora (es decir, un verdadero desorden).

Para ayudar a corregir al menos parcialmente este problema, Rstudio - la interfaz gráfica desde la cual nos comunicaremos con R - ofrece la posibilidad de guardar todos los archivos vinculados a un proyecto en una misma dirección de nuestra computadora. De esta manera, evitamos tener que recordar dónde guardamos los archivos y podemos movernos rápidamente de proyecto en proyecto.

[1] De hecho, en su portal de datos se pueden acceder a información sobre precios de inmuebles en Argentina, Brasil y México. Se puede consultar en https://www.properati.com.ar/data/

Figura 1: Nota de TELAM sobre el precio de los inmuebles en CABA durante 2017

Crear un nuevo proyecto en RStudio es realmente simple. Solo tienen que ir a `File -> New Project... -> New Directory -> New Project`. Allí deben elegir un nombre para el proyecto y hacer click en `Create Project` ¡Listo! Ya creamos nuestro primer proyecto. Nuestra pantalla ahora debería estar dividida en cuatro paneles (Ver Figura 2)

Por default, RStudio divide la pantalla en cuatro paneles. Ariba a la izquierda vemos el *Panel de edición de archivos*, donde vamos a modificar los archivos donde guardamos nuestro código, a la derecha vemos el *Panel de estructuras de datos*, donde aparecen las estructuras de datos que están creadas (más sobre esto abajo en el documento). Abajo a la izquierda tenemos el panel donde se visualiza la *consola o terminal*, que es donde vamos a ver ejecutado nuestro código y a su derecha un *Panel multiuso*, que será de utilidad para explorar archivos, ver la salida gráfica de nuestro código, encontrar ayuda sobre nuestros comandos, entre otras funciones.

Nuestro primer código

R es mucho más que una calculadora, pero nuestro primer código va a ser una multiplicación. En el panel de la consola, escriban el siguiente código y presionen `Enter`

```
5 * 10 # No deberíamos necesitar a R para esto...
```

```
## [1] 50
```

Debería devolver, también en el panel de la consola, el número 50. Ya ejecutamos nuestro primer código en R, pero antes de seguir ejecutemos lo mismo de una manera un poco distinta. Escriban exactamente el mismo código, pero ahora en el *Panel de edición de archivos*. Una vez que lo hayan escrito, seleccionen el código completo con el mouse y presionen `Control + Enter`. En la consola deberían obtener exactamente el mismo resultado.

¿Qué fue lo que pasó? Nuestro código está en un archivo (que todavía no guardamos). Al seleccionar el código y apretar `Control + Enter` RStudio pasa este código a la consola y se ejecuta, tal como antes. Para guardar el archivo que acabamos de editar, presionen `Control + S`, pongan el nombre que quieran al nuevo archivo, que tendrá la terminación `.R`, que es como vamos a identificar a los archivos que contienen código de R.

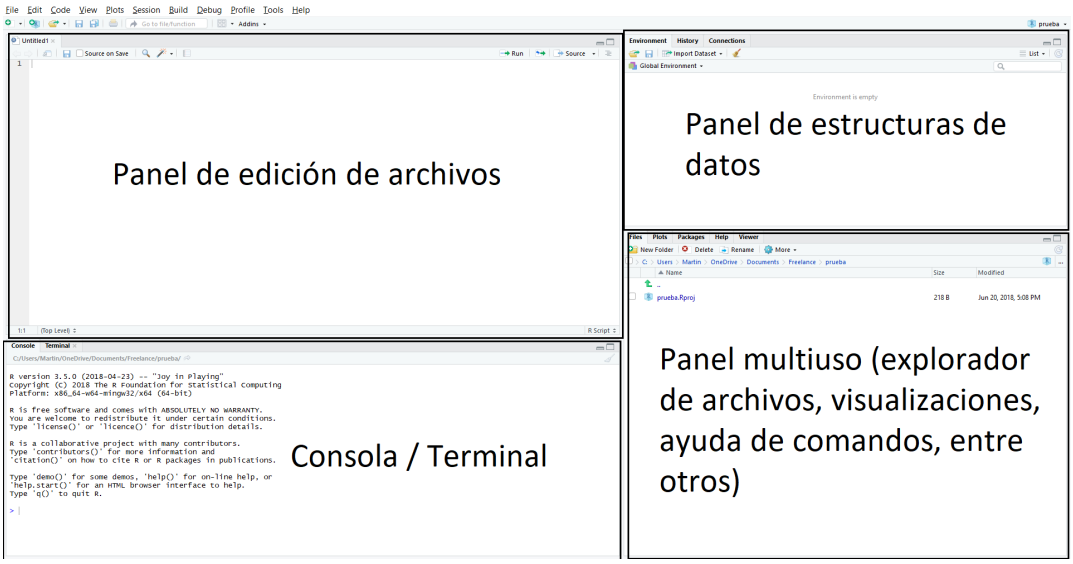


Figura 2: Impresión de pantalla de RStudio

Instalando paquetes

Antes de leer los datos en nuestra sesión de R vamos a necesitar un **package** que nos va a permitir trabajar con los **datos espaciales**. Un **package** o **library** no es otra cosa que un conjunto de código, exactamente como el que escribimos anteriormente, desarrollado con el objetivo de brindar algunas funcionalidades que simplifican nuestro trabajo. Como vamos a ver más adelante en nuestro curso, los datos especiales tienen características específicas que generan ciertos desafíos para su manipulación, visualización y análisis. El paquete **sf** (Simple Features) nos va a brindar la ayuda necesaria para lidiar con este problema.

Instalar paquetes en R es realmente simple. Solo tenemos que ejecutar el comando `install.packages()` y pasar el nombre del paquete que queremos instalar entre comillas. En este caso particular, el comando a ejecutar sería

```
install.packages("sf")
```

Una vez que estamos seguros que tenemos el paquete instalado, lo único que debemos hacer para poder usarlo es usar el comando `require(sf)`, pudiendo reemplazar **sf** por cualquier otra librería/paquete que hayan instalado y cuyas funciones precisen utilizar. Todo esto puede resultar un poco confuso, pero va a resultar realmente intuitivo en breve.

```
require(sf)
```

```
## Loading required package: sf
```

```
## Linking to GEOS 3.6.1, GDAL 2.2.3, proj.4 4.9.3
```

Cargando datos

Ahora que tenemos todo preparado para comenzar nuestro trabajo, empezamos por la primera etapa de todo proyecto de ciencia de datos: leer los datos.Los archivos los vamos a leer mediante la función **st_read** (más información sobre lo que es una *función* más abajo) que nos brinda el paquete **sf**. El código es el siguiente:

```
comunasPrecio <- st_read('https://datalab-utdt.github.io/datasets/comunas_price.geojson',
                         stringsAsFactors = F)
```

```
barriosPrecio <- st_read('https://datalab-utdt.github.io/datasets/barrios_price.geojson',
                         stringsAsFactors = F)
```

Si todo salió bien, ahora deberían tener dos *objetos* llamados **barriosPrecio** y **comunasPrecio** en el panel de arriba a la derecha. Es mejor aclarar paso por paso qué fue lo que pasó, porque hicimos muchas cosas que involucran diversos conceptos en un solo paso.

1. En primer lugar, definimos el nombre del **objeto** que va a contener nuestros datos. Los objetos son realmente importantes en R y consisten en estructuras que contienen datos. En este caso particular, elegí llamar a estos objetos **barriosPrecios** y **comunasPrecios**, pero en realidad podría ir cualquier texto ahí. Lo importante es que de ahora en más cuando queramos referirnos a esos datos lo podemos hacer con este *apodo*.

2. Inmediatamente después del nombre del objeto escribimos `<-`. Con este símbolo le decimos a R que el resultado de lo que escribamos a la derecha tiene que ser **asignado** al objeto de la izquierda. Siempre en R podemos **asignar** de está manera y (casi siempre) podemos hacer lo mismo usando el símbolo `=`.

3. Después usamos la función `st_read()`. Una **función** no es otra cosa que un conjunto de código que toma objetos como *inputs*, realiza transformaciones y devuelve otros objetos, transformados, como *outputs*. En este caso, a nuestra función le pasamos el valor de un sólo **parámetro**, para mantener las cosas simples. Se trata de la dirección donde están alojados datasets con información sobre el precio de los inmuebles para compra y venta para CABA según la información de Properati.

¿Cómo sabemos qué datos tenemos cargados? Tenemos al menos dos funciones que sirven para inspeccionar rápidamente los datos. Por un lado, podemos usar la función `View()` (Notar la **V** mayúscula al principio de la función). Lo que hay que hacer es pasarle uno de nuestros objetos que están cargados, y nos devuelve la tabla entera en el panel de edición.

```
View(barriosPrecio) # Debería abrirse una tabla dentro del panel de edición
```

Deberían ver una matriz con 7 columnas y unas 48 filas. Las filas representan observaciones (en este caso, barrios), mientras que las columnas hacen referencia a las variables, que tienen los siguientes nombres: BARRIOS, USDm2_2013, USDm2_2014, USDm2_2015, USDm2_2016, USDm2_2017 y geometry ¿Qué significan?

- BARRIOS: es una variable que indica cuál es el nombre del barrio al que pertenece cada observación
- USDm2_201x: estás 5 columnas hacen referencia al valor promedio de los inmuebles creados en cada año para cada barrio (período 2013-2017)
- geometry: esta columna contiene lo que hace únicos a los datos espaciales, es decir una dimensón y un determinado posicionamiento dentro de la tierra. Por ahora, basta con este conocimiento superficial, aunque luego profundizaremos.

Otra función que puede ser muy útil para describir nuestros objetos es `str()`. Nos devuelve las primeras observaciones de cada una de las variables, también la **clase** de nuestro objeto (en este caso, un **sf** y al mismo tiempo un **Data Frame**), la cantidad de observaciones (o filas), que en este caso son 48, y la cantidad de variables (o columnas), que son 7. También nos indica la **clase** de cada una de las variables, que en este caso son `int` o `num`. Para entender bien las clases de datos y las diversas estructuras que los contienen tenemos que ir un casillero más atrás y explicar cómo es R organiza las cosas.

```
str(barriosPrecio)
```

```
## Classes 'sf' and 'data.frame':   48 obs. of  7 variables:
##  $ BARRIOS   : chr  "AGRONOMIA" "ALMAGRO" "BALVANERA" "BARRACAS" ...
##  $ USDm2_2013: num  1749 2034 1893 2229 2620 ...
##  $ USDm2_2014: num  1458 1930 1818 1901 2544 ...
##  $ USDm2_2015: num  1689 2086 1858 2390 2695 ...
##  $ USDm2_2016: num  2057 1950 1834 1751 3287 ...
##  $ USDm2_2017: num  1946 2299 1989 1920 3199 ...
```

```
##  $ geometry  :sfc_MULTIPOLYGON of length 48; first list element: List of 1
##   ..$ :List of 1
##   .. ..$ : num [1:34, 1:2] -58.5 -58.5 -58.5 -58.5 -58.5 ...
##   ..- attr(*, "class")= chr  "XY" "MULTIPOLYGON" "sfg"
##  - attr(*, "sf_column")= chr "geometry"
##  - attr(*, "agr")= Factor w/ 3 levels "constant","aggregate",..: NA NA NA NA NA NA
##   ..- attr(*, "names")= chr  "BARRIOS" "USDm2_2013" "USDm2_2014" "USDm2_2015" ...
```

Todo es una estructura de datos

Antes de continuar con nuestra pregunta (*¿Subieron los precios de los inmuebles en 2017?*) resulta importante dar una introducción a cómo R organiza nuestros datos. Esta sección va a dar a la impresión que está fuera de contexto con respecto a nuestra pregunta inicial (y, de alguna manera, es una apreciación correcta), pero es preciso tener un conocimiento básico sobre cómo funciona R.

R organiza a los datos en diferentes **estructuras** según el *dominio* de los datos. El dominio no es otra cosa que los valores que una variable puede llegar a tomar. Por ejemplo, si una variable solo puede tomar números reales y enteros, R pondrá esa variable en un **vector** númerico o *integer*, para ser más preciso. En esta sección vamos a introducirnos en las principales estructuras de datos que maneja R.

Vectores

Un **vector** es la estructura más básica con la que vamos a lidiar. Un vector no es otra cosa que una *colección numerada de valores*, es decir una estructura que puede contener uno o más valores, y puede accederse a través de índices que denotan el orden de cada número dentro del vector ¿Simple, no? Creemos nuestro primer vector

```
primerVector <- c(20,40,60,80,100)
```

Ya está: el objeto `primerVector` es un vector númerico que tiene cinco elementos: `20, 40, 60, 80 y 100`. Como les prometí, el vector es una *colección numerada de valores*, por lo que podemos acceder a cualquiera de los valores llamando al objeto por su nombre y escribiendo la posición entre corchetes `[]` (la numeración arranca desde 1 en R)

```
primerVector[3] # Devuelve el valor del elemento en la tercera posición de nuestro vector
```

```
## [1] 60
```

La mayor parte de nuestros datos no consiste solo en números, sino en datos mixtos: números, texto, variables categóricas, variables lógicas. Todos estos tipos de datos pueden ser representados en nuestros versátiles vectores:

```
vectorTexto <- c("Croacia","Argentina","Nigeria","Islandia") # Ojalá termine así la fase de grupos
vectorLogico <- c(FALSE, TRUE, TRUE, FALSE)
```

Los vectores de texto no requieren demasiada discusión: son vectores cuyos elementos son texto. Por su parte, los vectores lógicos quizás sí requieran algo de explicación. Los elementos de estos vectores solo pueden tomar los valores TRUE o FALSE, y resultan de mucha utilidad para hacer preguntas del estilo ¿Es este vector un vector de texto?

```
is.character(vectorTexto)
```

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

Volvamos a nuestros datos sobre los precios de los inmuebles. Ya comentamos que había siete variables, pero no dijimos que a su vez son 7 vectores. Uno es CHARACTER, como nuestro `vectorTexto` anterior, cinco

son INTEGER, o numeric, como nuestro `primer_vector` y el último es sfc_MULTIPOLYGON (vamos a ver un poco más en detalle este último tipo de vector en próximas clases) ¿Cómo podemos acceder a ellos? La llave es el operador `$`

```
precios2017 <- barriosPrecio$USDm2_2017
str(precios2017) # ¿Los 100 barrios porteños en realidad son 48?
```

```
##  num [1:48] 1946 2299 1989 1920 3199 ...
```

¿Qué fue lo que hicimos? Creamos el objeto `precios2017` al que le asignamos (`<-`) el vector/variable `USDm2_2017` que está en el **Data Frame** barriosPrecio (vamos a ver más sobre los Data Frame más adelante). Al usar la función `str()` (son las primeras tres letras de la palabra *structure* en inglés) vemos que se trata de un vector *integer* (o númerico) con 48 elementos ¡La misma cantidad de observaciones que tenemos en `barriosPrecio`!

Ahora es donde las cosas se ponen un poco más interesantes. Podemos hacer múltiples operaciones aritméticas sobre este vector de una manera muy simple:

```
str(precios2017*2) # Multiplicación
```

```
##  num [1:48] 3891 4598 3978 3839 6398 ...
```

```
str(precios2017/4) # División
```

```
##  num [1:48] 486 575 497 480 800 ...
```

```
str(precios2017 - 20) # Resta por solo un número
```

```
##  num [1:48] 1926 2279 1969 1900 3179 ...
```

```
str(precios2017 - precios2017) # Resta con otro vector de igual tamaño
```

```
##  num [1:48] 0 0 0 0 0 0 0 0 0 0 ...
```

Una de las ventajas de R es su *vectorización*. En las operaciones anteriores vimos como múltiplicar, dividir, restar o cualquier otra operación aritmética se **aplica de manera individual sobre cada uno de los elementos de los vectores**, con lo cual la operación se hace elemento por elemento.

Hagamos algo que sea útil para describir la distribución de los precios de los inmuebles ¿Cuál es el valor mínimo del metro cuadrado? ¿Cuál es el máximo? ¿Cuál es el precio promedio? Las funciones `min()`, `max()` y `mean()` hacen el trabajo por nosotros.

```
resumen_2017 <- c(min(precios2017), max(precios2017), mean(precios2017))
resumen_2017
```

```
## [1]  872.7541 6070.3288 2203.1119
```

Listas y Data Frames

Otras estructuras de datos importantes en R son las **listas** y **Data Frames**:

- Las **listas** son objetos que contienen a su vez un conjunto ordenados de objetos.
- Los **Data Frames** son un caso específico de listas.

Listas

Las listas tienen la posibilidad de almacenar objetos con distinta clase. Es decir, es posible crear una lista en la cual se almacenan otras estructuras de datos con clases distintas:

```
lista1 <- list(Nombres = c("Fernando","Martín"), Apellido="Montané", tienehijos = FALSE,
               edad = 26)
lista1
```

```
## $Nombres
## [1] "Fernando" "Martín"
##
## $Apellido
## [1] "Montané"
##
## $tienehijos
## [1] FALSE
##
## $edad
## [1] 26
```

En este caso creamos una lista que se llama *lista1* y almacena 4 vectores. El primero, es un vector *Nombres* que contiene dos elementos de clase *character*. El segundo, un vector de un solo elemento, *Apellido* de clase *character*. El tercero, un vector lógico de un elemento (*tienehijos*). Finalmente, el vector *edad*, que posee un elemento númerico.

En las listas hay que acceder a los objetos, que puede ser cualquiera de las estructuras que vimos hasta ahora (incluyendo las listas). Para acceder a ellos existen al menos 3 formas:

- Usando el signo $ `lista$Nombres`
- Usando doble corchetes e indicando la posición del objeto buscado `lista[[1]]`
- Usando doble corchetes e indicando el nombre del objeto buscado `lista[['Nombres']]`

```
lista1$Nombres
```

```
## [1] "Fernando" "Martín"
```

```
lista1[[1]]
```

```
## [1] "Fernando" "Martín"
```

```
lista1[['Nombres']]
```

```
## [1] "Fernando" "Martín"
```

Data Frames

¡Ahora estamos en condiciones de explicar qué es un `Data Frame`! Como se adelantó, un **Data Frame** es un caso específico de listas. En la gran mayoría de las aplicaciones se puede describir como una matriz en la cual variables (columnas) pueden ser de distintas clases.

```
df <- data.frame(Nombres = c('Juan','Pedro','Ana','Delfina'),
                 Edad = c(21,46,58,27),
                 EstadoCivil = c('Soltero','Casado','Casado','Soltero'),
                 SecundarioCompleto = c(TRUE, FALSE, TRUE, TRUE))
df
```

```
##   Nombres Edad EstadoCivil SecundarioCompleto
## 1    Juan   21     Soltero               TRUE
## 2   Pedro   46      Casado              FALSE
## 3     Ana   58      Casado               TRUE
## 4 Delfina   27     Soltero               TRUE
```

```r
str(df) # Resumen del objeto
```

```
## 'data.frame':    4 obs. of  4 variables:
##  $ Nombres           : Factor w/ 4 levels "Ana","Delfina",..: 3 4 1 2
##  $ Edad              : num  21 46 58 27
##  $ EstadoCivil       : Factor w/ 2 levels "Casado","Soltero": 2 1 1 2
##  $ SecundarioCompleto: logi  TRUE FALSE TRUE TRUE
```

El comando *str* nos devuelve un resumen de las variables del data frame, incluyendo sus clases. Si prestan atención, el vector numérico *Edad* y el lógico *SecundarioCompleto* tienen la clase esperada, pero los vectores de caracteres *Edad* y *SecundarioCompleto* dice **Factor** w/2 o w/4 levels ¿Qué es esto?

La clase *Factor* es un caso específico de vectores. En esta clase de vectores, las variables solo pueden tomar un conjunto de valores predeterminados, es decir que tienen una categoría. Estas categorías se llaman *levels* en R. Internamente R transforma estas variables y les asigna un número entero, que son los valores que nos devolvió el comando str. Sin embargo, estos números hacen referencia a una categoría que tiene un nombre. En nuestro caso ¿Tiene sentido que los dos vectores con texto sean Factores?. Respuesta: no. El estado civil de una persona sí puede segmentarse en categorías mutuamente excluyentes, pero no es el caso de los nombres. Para evitar que R transforme nuestros vectores *character* en *Factor* lo que hay que hacer cuando definimos un **Data Frame** es decírselo a R con el parámetro *stringsAsFactors*:

```r
df <- data.frame(Nombres = c('Juan','Pedro','Ana','Delfina'),
                 Edad = c(21,46,58,27),
                 EstadoCivil = c('Soltero','Casado','Casado','Soltero'),
                 SecundarioCompleto = c(TRUE, FALSE, TRUE, TRUE),
                 stringsAsFactors = FALSE)
df
```

```
##   Nombres Edad EstadoCivil SecundarioCompleto
## 1    Juan   21     Soltero               TRUE
## 2   Pedro   46      Casado              FALSE
## 3     Ana   58      Casado               TRUE
## 4 Delfina   27     Soltero               TRUE
```

```r
str(df) # Resumen del objeto. Ahora ya no tenemos Factors.
```

```
## 'data.frame':    4 obs. of  4 variables:
##  $ Nombres           : chr  "Juan" "Pedro" "Ana" "Delfina"
##  $ Edad              : num  21 46 58 27
##  $ EstadoCivil       : chr  "Soltero" "Casado" "Casado" "Soltero"
##  $ SecundarioCompleto: logi  TRUE FALSE TRUE TRUE
```

Los vectores *Factor* son de mucha utilidad, especialmente cuando se trabaja con modelos estadísticos en R. Vamos a verlos en otras aplicaciones más adelante.

Ahora ya tenemos los elementos suficientes como para definir más concretamente qué es un Data Frame: es un conjunto de listas, que en su versión más usual contienen un vector cada una de un mismo largo (mismas cantidad de observaciones), pero que pueden almacenar variables de distinto dominio. Esto es algo muy útil para la mayor parte de datasets con los que se trabaja.

Exploración visual de los datos

En los últimos años se ha concebido un conjunto de *packages* que se denomina *tidyverse*. Según RStudio, *Tidyverse* es "[...] a coherent system of packages for data manipulation, exploration and visualization that share a common design philosophy". En la práctica, tidyverse nos va a permitir articular e implementar diversas facetas del proceso de análisis de datos de una manera unificada y, en jerga de economista, aprovechando sus

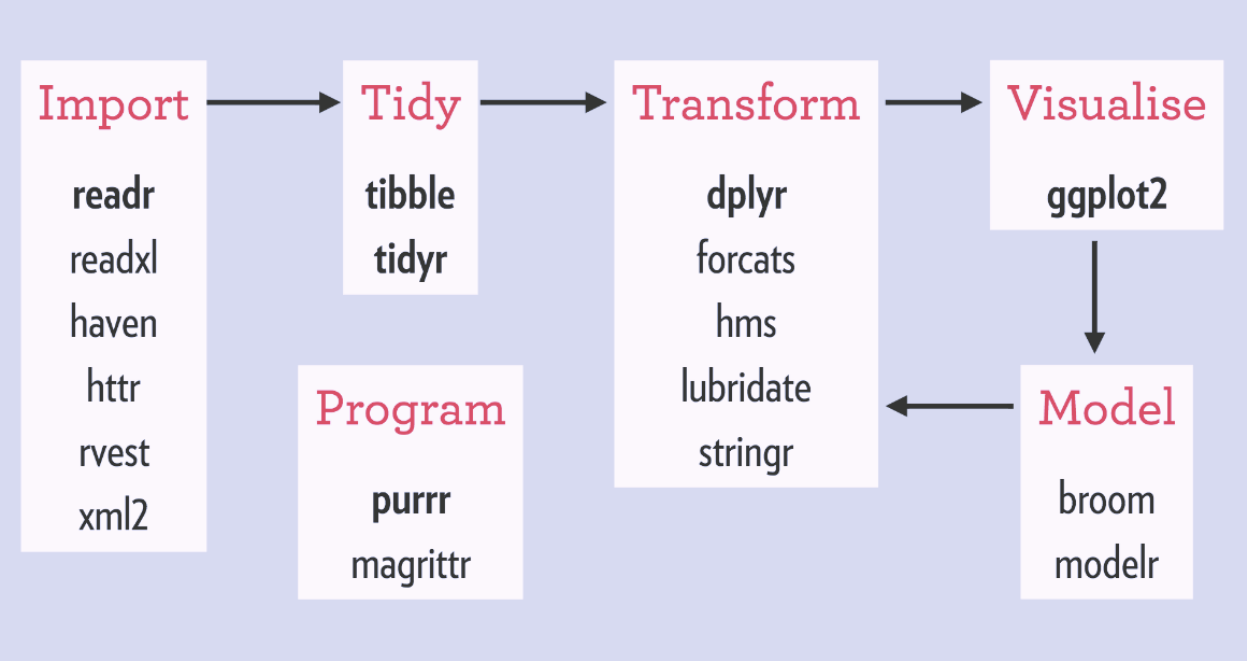


Figura 3: Ecosistema de paquetes que componen *tidyverse* y su vinculación con cada proceso de un proyecto de Ciencia de Datos

economías de escala. Tiene una curva de aprendizaje, por lo cual no se preocupen si en esta primera sección hay algunas cosas que no terminan de entenderse. En las próximas clases todo va a tener sentido.

Para instalar tidyverse tienen que ejecutar las siguientes líneas (y ser pacientes, porque puede tardar unos minutos)

```
install.packages("tidyverse")
install.packages("devtools")
devtools::install_github("tidyverse/ggplot2")
install.packages("tmap")
```

Una vez que se hayan instalado los paquetes que conforman el tidyverse, debemos cargar las funciones a nuestra sesión de R, como ya hicimos anteriormente con el paquete **sf**

```
require(tidyverse)
```

```
## Loading required package: tidyverse
```

```
## -- Attaching packages ---------------------------------------------------------------------------
```

```
## v ggplot2 2.2.1.9000     v purrr   0.2.5
## v tibble  1.4.2          v dplyr   0.7.5
## v tidyr   0.8.1          v stringr 1.3.1
## v readr   1.1.1          v forcats 0.3.0
```

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

```
require(tmap)
```

```
## Loading required package: tmap
```

Antes de realizar alguna visualización de nuestro problema, primero usaremos la función **select**, que nos permite seleccionar solo algunas de nuestras variables. Para nuestro primer análisis solo necesitamos tres variables: el nombre del barrio, el valor de las propiedades en 2016 y en 2017.

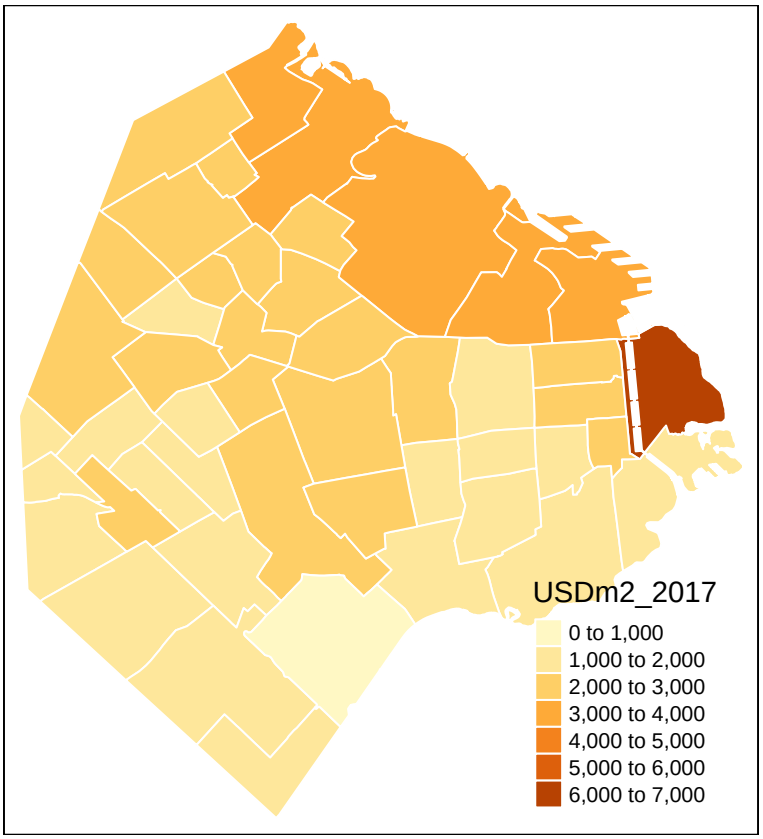
```
barriosPrecioSubset <- barriosPrecio %>% select(BARRIOS, USDm2_2016, USDm2_2017)
str(barriosPrecioSubset)
```

```
## Classes 'sf' and 'data.frame':   48 obs. of  4 variables:
##  $ BARRIOS   : chr  "AGRONOMIA" "ALMAGRO" "BALVANERA" "BARRACAS" ...
##  $ USDm2_2016: num  2057 1950 1834 1751 3287 ...
##  $ USDm2_2017: num  1946 2299 1989 1920 3199 ...
##  $ geometry  :sfc_MULTIPOLYGON of length 48; first list element: List of 1
##   ..$ :List of 1
##   .. ..$ : num [1:34, 1:2] -58.5 -58.5 -58.5 -58.5 -58.5 ...
##   ..- attr(*, "class")= chr  "XY" "MULTIPOLYGON" "sfg"
##  - attr(*, "sf_column")= chr "geometry"
##  - attr(*, "agr")= Factor w/ 3 levels "constant","aggregate",..: NA NA NA
##   ..- attr(*, "names")= chr  "BARRIOS" "USDm2_2016" "USDm2_2017"
```

Como podemos ver, el resultado del código fue un nuevo objeto, `barriosPrecioSubset`, que contiene cuatro variables en lugar de las siete originales. Lo más importante del código anterior es el uso del **pipe** (%>%). En el entorno del **tidyverse**, un **pipe** indica que todo lo que está antes de él será transformado por lo que está a luego de él: en este caso, el dataset `barriosPrecio` será filtrado por aquellas variables que están separadas por comas dentro del comando `select()`

¿Cómo se ve la distribución de los precios por barrios en la capital federal? Podemos hacer un primer gráfico con una sola línea:

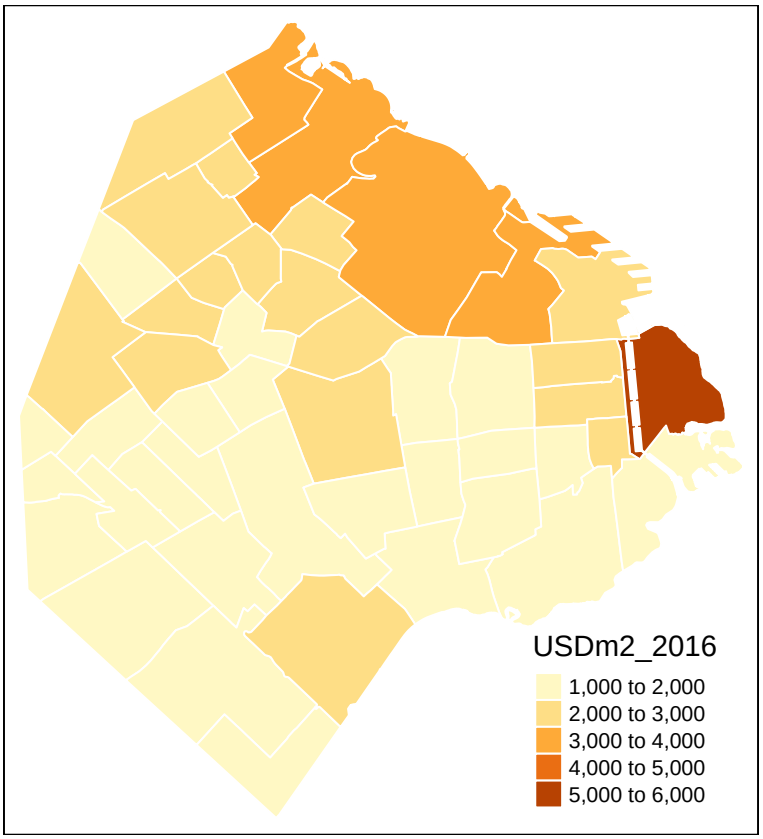
```
tm_shape(barriosPrecioSubset) + tm_polygons(border.col = 'white',col = 'USDm2_2017')
```



¿Qué hicimos? el paquete **tmap** es muy útil para hacer mapas de una manera rápida. Su forma de operar es mediante múltiples funciones mediadas por signos `+`, es decir que trabaja por capas. Con la función `tm_shape()` pasamos el dataset que contiene información espacial (en la columna **geometry**). Luego del `+` agregamos la función `tm_polygons()`, donde indicamos que queremos graficar polígonos, y tiene dos parámetros: `border.col`, donde definimos el color de los bordes de los polígonos, y luego definimos su color **en base al precio promedio de los inmuebles en 2017** con el parámetro `col` ¿Fácil, no?

El patrón es claro, aun si la paleta (y la segmentación de los precios) no es la mejor: la parte norte de la ciudad exhibe los precios de los inmuebles más caros de la capital, mientras que la zona sur los precios más bajos. ¿Y si queremos gráficar la misma información, pero para el año 2016? Muy fácil:

```
# Solo tenemos que cambiar el nombre de la variable en el parámetro col
tm_shape(barriosPrecioSubset) + tm_polygons(border.col = 'white',col = 'USDm2_2016')
```



No se observa una gran diferencia entre ambos mapas, la discrepancia entre norte y el resto de la capital sigue observándose de una manera nítida. Quizás una de las diferencias es que hay un valor de la paleta que en 2017 no existía ¿Cuál es?

Variación de los precios

Una de nuestras preguntas es si los precios de las propiedades habían aumentado entre 2017 versus 2016, teniendo en cuenta que la información de los medios es que habían aumentado un 10% interanual. Responder esta pregunta es realmente fácil con nuestro dataset y puede realizarse en solo una línea:

```
variacion <- (mean(barriosPrecioSubset$USDm2_2017) / mean(barriosPrecioSubset$USDm2_2016) - 1)*100
round(variacion,1) # La función round nos permite redondear un número con los decimales que queramos.
```

```
## [1] 5.3
```

Nuestros datos nos dicen que los precios en 2017 subieron un 5,3% en dólares entre 2017 y 2016, 5 puntos por debajo de lo divulgado por los medios. Esta divergencia puede estar explicada por muchos factores. El principal factor puede haber sido que nuestra variación en los inmuebles es un promedio simple de la variación en cada uno de los barrios (es decir, la variación en cada barrio se pondera en la misma proporción). Sin embargo, una desproporcionada cantidad de operaciones ocurre en un pequeño puñado de barrios, como veremos más adelante. Si la variación entre barrios mostrara una importante dispersión, esta forma de medir los cambios puede introducir un sesgo. Lo que nos lleva a nuestra segunda pregunta ¿Hay una heterogeneidad entre los barrios de la ciudad?

Heterogeneidad en los barrios

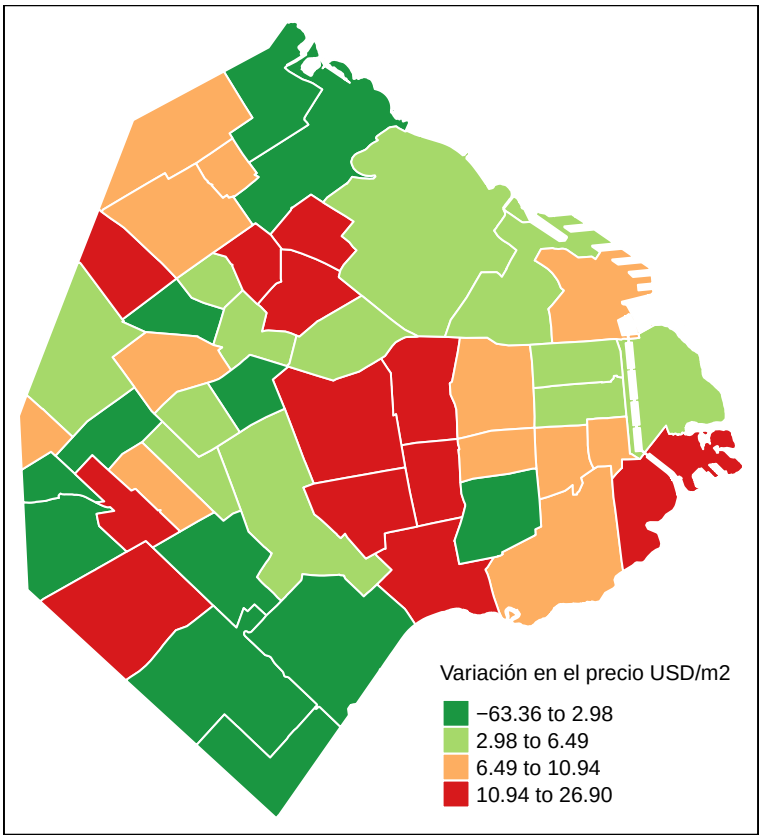
¿Como variaron los precios en la ciudad? ¿Todos aumentaron el 5,3%? Para responder esta pregunta, precisamos primero calcular la variación de cada uno de los barrios entre 2017 y 2016. Esto lo podemos resolver aprovechando la **vectorización** de R y la función `mutate()` de **tidyverse**.

```
barriosPrecioSubset <- barriosPrecioSubset %>% mutate(variacion = (USDm2_2017/USDm2_2016-1)*100)
barriosPrecioSubset$variacion
```

```
##  [1]  -5.4035088  17.8714552   8.4576584   9.6127466  -2.6578132
##  [6]  14.7272999  18.8983294  10.9509072  15.3829179   6.7595124
## [11]  11.4799767   6.6757422   5.4713752   5.9094725   1.1777652
## [16]  11.6576231   5.8701266   1.3695028  18.0571806   0.5309427
## [21]   3.0335920  -1.2597886  16.4787925   5.6182089  -0.7066143
## [26]   5.3362407   5.7047785   5.8550888  10.9335670   7.9971871
## [31]   8.2429874   3.7502030   9.3635362   9.4722438  -1.1836759
## [36]   6.3200309   7.7699205   4.1923218   2.8301429  -4.6145371
## [41]  19.0088455  19.1065358  26.8950101   9.1110358 -11.1147102
## [46]   5.6330404 -63.3609689   6.6505565
```

La función **mutate()** nos permite generar nuevas variables (o modificar otras ya existentes) en nuestros data frames. En este caso, creamos la variable *variacion*, que toma el valor de la operación algebraica que está a su derecha. Voy a adelantarme unas clases en el curso y voy a usar un mapa con algunas funciones que todavía nos vimos, pero que nos va a ayudar a ver rápidamente si existió heterogeneidad en la variación de los precios de los inmuebles:

```
tm_shape(barriosPrecioSubset) + tm_polygons(border.col = 'white', col = 'variacion',
                                            breaks = quantile(barriosPrecioSubset$variacion),
                                            palette = c('#1a9641','#a6d96a','#fdae61','#d7191c'),
                                            title = 'Variación en el precio USD/m2')
```



Como se puede observar claramente en el gráfico anterior, no hay un patrón homogéneo en las variaciones: si bien la suba fue prácticamente generalizada, en algunos barrios ha sido mucho más intensa que en otros. Veamos cuales fueron los 10 barrios con mayor suba combinando la función `arrange()` y `select()`. La función `arrange()` solo requiere que indiquemos la variable por la cual ordenar el data frame, por default lo hace de manera ascendente. Por esta razón, agregamos la función `desc()` que genera un ordenamiento descendiente.

```
# ¿Cuáles son los diez barrios en los que registramos las subas más altas?
barriosPrecioSubset %>% arrange(desc(variacion)) %>% select(BARRIOS,variacion)
```

```
## Simple feature collection with 48 features and 2 fields
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: -58.53092 ymin: -34.70574 xmax: -58.33455 ymax: -34.52799
## epsg (SRID):    4326
## proj4string:    +proj=longlat +datum=WGS84 +no_defs
## First 10 features:
##             BARRIOS variacion                       geometry
## 1  VILLA PUEYRREDON  26.89501 MULTIPOLYGON (((-58.50287 -...
## 2     VILLA ORTUZAR  19.10654 MULTIPOLYGON (((-58.46749 -...
## 3        VILLA LURO  19.00885 MULTIPOLYGON (((-58.51577 -...
## 4             BOEDO  18.89833 MULTIPOLYGON (((-58.42719 -...
## 5     NUEVA POMPEYA  18.05718 MULTIPOLYGON (((-58.40024 -...
## 6           ALMAGRO  17.87146 MULTIPOLYGON (((-58.43274 -...
## 7  PARQUE CHACABUCO  16.47879 MULTIPOLYGON (((-58.44232 -...
## 8         CHACARITA  15.38292 MULTIPOLYGON (((-58.43794 -...
## 9              BOCA  14.72730 MULTIPOLYGON (((-58.36127 -...
```

```
## 10        MATADEROS  11.65762 MULTIPOLYGON (((-58.52517 -...
```

¿Cuáles fueron los barrios con menores variaciones?

```
# ¿Cuáles son los diez barrios en los que registramos las mayores caídas?
barriosPrecioSubset %>% arrange(variacion) %>% select(BARRIOS,variacion)
```

```
## Simple feature collection with 48 features and 2 fields
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: -58.53092 ymin: -34.70574 xmax: -58.33455 ymax: -34.52799
## epsg (SRID):    4326
## proj4string:    +proj=longlat +datum=WGS84 +no_defs
## First 10 features:
##              BARRIOS   variacion                       geometry
## 1      VILLA SOLDATI -63.3609689 MULTIPOLYGON (((-58.45238 -...
## 2    VILLA RIACHUELO -11.1147102 MULTIPOLYGON (((-58.4752 -3...
## 3          AGRONOMIA  -5.4035088 MULTIPOLYGON (((-58.47537 -...
## 4       VILLA LUGANO  -4.6145371 MULTIPOLYGON (((-58.48137 -...
## 5           BELGRANO  -2.6578132 MULTIPOLYGON (((-58.43711 -...
## 6  PARQUE AVELLANEDA  -1.2597886 MULTIPOLYGON (((-58.47794 -...
## 7          VERSALLES  -1.1836759 MULTIPOLYGON (((-58.52103 -...
## 8   PARQUE PATRICIOS  -0.7066143 MULTIPOLYGON (((-58.39057 -...
## 9           NU<d1>EZ   0.5309427 MULTIPOLYGON (((-58.45885 -...
## 10           LINIERS   1.1777652 MULTIPOLYGON (((-58.50169 -...
```

En este último comando hubo algo raro ¿En Villa Soldati el precio de los inmuebles cayó un 63%? Esto es imposible: se debe a que la cantidad de datos que se registran de ciertas zonas de la ciudad no son representativas del mercado inmobiliario. Teniendo en cuenta este dato ¿cómo quedaría la respuesta a nuestra primera pregunta si excluímos a Villa Soldati?

```
# En la próxima clase explicaremos en detalle el funcionamiento de la función filter()
excl_soldati <- barriosPrecioSubset %>% filter(!BARRIOS == 'VILLA SOLDATI')
mean(excl_soldati$variacion)
```

```
## [1] 7.302633
```

Ahora la variación es de 7,3%, mucho más cerca de nuestro *benchmark* inicial de 10%.

Conclusiones

Los datos públicos a los que accedimos nos permitieron responder las dos preguntas iniciales:

1. En primer lugar, se observó un incremento en los precios de los inmuebles en USD durante el 2017 en comparación al 2016. Este aumento fue de aproximadamente 7,3%, según un promedio simple de las variaciones barriales.
2. En segundo lugar, las variaciones en los precios exhiben una importante heterogeneidad regional, con algunos barrios ubicados en la parte noroeste (Villa Pueyrredón, Villa Ortúzar, Chacarita, Colegiales) y sudeste (Nueva Pompeya, Parque Chacabuco, Boedo) de la capital aumentando más que el resto.

Ejercicios

Hasta ahora no hemos trabajado con el dataset `comunasPrecio`. Para ejercitar y asentar lo aprendido, replicar el mismo análisis que hicimos a nivel barrial, pero ahora a nivel comunal ¿Qué diferencias se observan?