Building Shiny Apps

Pablo Maldonado, Ph.D.

Building Shiny Apps

A hands-on introduction

Pablo Maldonado

This book is for sale at http://leanpub.com/buildingshinyapps

This version was published on 2016-11-19

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Introduction

The purpose of this book is to help you to develop Shiny apps with zero to little knowledge of R. Shiny is a web framework, developed by RStudio Inc., which allows to quickly build prototypes and in many cases even production versions of data-driven applications.

We live on the information age, and that's a curse and a blessing. When I started my own journey into Shiny, I felt a bit overwhelmed by the documentation, both because of its abundance but also by its depth. Yes, the documentation is definitely excellent, and the team in RStudio are really supportive to the user community, but I've always felt that it was a bit too much for beginners. I personally find that the documentation is written for techically-minded people, however, it is really a great tool for business users as well.

During my consulting experience at a global organization, I saw that many of my colleagues were eager to learn, and, quite capable. When I showed them the essential parts of the documentation to get the work done, they could easily get started on their own and answer their own questions.

My intention is not to rewrite the documentation. What I am offering you is the essential parts of it to help you get started, the *curated* documentation, if you wish. These are the parts I learned through trial, error and experimentation with my colleagues (sorry guys).

The structure of the book is as follows: We'll do a quick introduction to R and the package dplyr (version 0.5.0) for data wrangling. We will not cover this in detail, as more advanced users may rely on the excellent documentation provided by Hadley Wickham, Chief Scientist of the RStudio foundation, and dplyr's creator. Then we'll follow with ggplot2, an excellent plotting package (also from Hadley Wickham) that seamlessly integrates with dplyr. From here, thanks to the support for ggplot from plot.ly, there's just one more step to beautiful plots. On Chapter 3, we will cover the main features of Shiny and guide you through an example. Chapter 4 is about building dashboards using the shinydashboard package. Finally, in Chapter 5, we will cover maps using the Leaflet package. The book is example-guided and expects that the reader will put some effort to it.

Source codes and datasets for the exercises are available through Github or on request.

Chapter 1: R and dplyr at a glance

What is R?

The R language was created by researchers Ross Ihaka and Robert Gentleman in 1991 in the Department of Statistics, University of Auckland, New Zealand. It is derived from the S language, originally developed in the 1950's in Bell Labs by John Chambers and his team.

R is one of the most used tools by data scientists. It is free (both as in "free beer" and "free person") statistical package, although there exist commercial versions with support for additional features. R is relatively easy to learn and has lots of packages for advanced statistical computations.

One of the main disadvantages against Python, R's main competitor in the data science landscape, is that objects in general need to reside in physical memory. This is slowly becoming less of a problem, since both the R core team and external package developers have implemented features to get around with this problem.

What is dplyr?

As self-described in its documentation, dplyr is

> "A fast, consistent tool for working with data frame like objects, both in memory and out of memory".

It was created by Hadley Wickham (Rice University/RStudio), and is part of a suite of very useful packages for data manipulation in R, such as ggplot2, which will be covered as well.

I will assume that you have installed R and RStudio in your system and that they are correctly running.

R in a nutshell

Let us jump straight into R syntax and some useful commands.

Basic Syntax

In R, the assignment operator is <- `x <- 1 print(x)`

You can use =, but <- is preferred by the R community, and there are some differences. In fact, Google forbids its use in the [R Style guide] (https://google.github.io/styleguide/Rguide.xml), which I reccommend you to look at. The keyboard shortcut for <- is **Alt** + - in RStudio.

A quick way to get help is to use the **?** command. For instance

```
?print
```

launches a help file in RStudio describing the documentation of the `print` function.

Types

We describe now the most important data types in R, which are the source of silly errors in the beginning.

Vectors and types

The following types are supported in R

```
x <- c(0.5,0.6) #numeric
x <- c(T,F) #logical
x <- c("a","b","c") #character
x <- c(1+0i,2+4i) #complex
```

You can use TRUE or FALSE for logical as well.

R uses compressed notation for vectors with consecutive integers:

```
x <- 9:29
print(x)
```

which prints the numbers from 9 to 29 (both inclusive)

Coercion

Sometimes we need to coerce the types of the input data to be correctly interpreted by the package we want to use.

Let's see how it works with an example:

```
x <- 0:6
class(x)
```

This returns:

```
[1]  "integer"
```

If we do now:

```
as.logical(x)
```

we get

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

Finally,

```
as.character(x)
```

gives us

```
[1] "0" "1" "2" "3" "4" "5" "6"
```

Coercion is not always possible, for instance

```
x <- c("a","b","c")
as.numeric(x)
```

yields

```
Warning: NAs introduced by coercion
[1] NA NA NA
```

which leads us to...

Special values in R

- **NA** is used for existing, but useless values.
- **NaN** is used for undefined values, like 0/0.
- **NULL** means unexistant value.
- There's also **Inf** and **-Inf**.

So, for example:

```
v <- c(1, 2, 3, NA, 5)
sum(v)
```

gives

```
[1] NA
```

while

```
v <- c(1, 2, 3, NaN, 5)
sum(v)
```

returns

```
[1] NaN
```

and

```
v <- c(1, 2, 3, NULL, 5)
sum(v)
```

returns

```
[1] NULL
```

If you think about it for a moment, this convention makes sense. To check for this special values, you can use:

```
is.na(5) #FALSE
is.na(NaN) #TRUE
is.nan(NA) #FALSE
is.null(NULL) #TRUE
is.finite(Inf) #FALSE
is.na(Inf) #FALSE
```

Again, think about this for a moment until it sinks in.

Factors are a special and very useful data structure.

```
x <- factor(c("yes","no", "yes", "no", "no"))
x
```

Factors are special ways of representing data internally, and they are treated specially by modelling functions.

Data Frames

Data Frames are data structures resembling the matrices from your algebra class. The main difference is that data frames can accommodate other kinds of data structures, such as text. For example:

```
df <- data.frame( Weather = c("Cold","Mild","Cold","Ok")
                , Cities = c("Prague", "Brno", "Ostrava", "Zlin")
                )
df
```

creates a small data frame with two columns and 4 rows:

```
> df
  Weather  Cities
1    Cold  Prague
2    Mild    Brno
3    Cold Ostrava
4      Ok    Zlin
```

Doing

```
df$Cities
```

yields

```
[1] Prague Brno Ostrava Zlin
Levels: Brno Ostrava Prague Zlin
```

and we can query the data frame referring to row, columns as with matrices, for instance:

```
df[2,1] # Returns Mild
df[2,"Weather"] #Also returns Mild
```

We can do more complicated things with data frames, such as finding subsets that satisfy certain conditions:

```
df[df$Weather=='Mild',]
```

this gives us all the rows satisfying the condition in the first argument, and leaving a space after the comma tells R to return all the columns. Hence we get:

```
  Weather Cities
2    Mild   Brno
```

However, for our purposes, we will focus on dplyr instead of R base functions. dplyr provides a cleaner interface, which is also easier to debug as our app grows.

Loading and inspecting

We will use the online news popularity dataset from the UCI repository, available here[1] for our examples. This data set consists of the URLs of articles in the online portal mashable.com and the task of the researchers was to predict whether an article would become highly shared or not, and how to tweak it to make it more shareable, based on a number of attributes, such as the number of media content, the topic of the article, the day of the week, etc.

I have downloaded the dataset and stored it into a directory called "/data" in my computer. Please download it and save it in your computer. The data set includes a data dictionary that will explain you each columns.

```
news<-read.csv('./data/OnlineNewsPopularity/OnlineNewsPopularity.csv')

small <- news[2:5]
head(small) # Too many columns
```

[1] https://archive.ics.uci.edu/ml/datasets/Online+News+Popularity

```
  timedelta n_tokens_title n_tokens_content n_unique_tokens
1       731             12              219       0.6635945
2       731              9              255       0.6047431
3       731              9              211       0.5751295
4       731              9              531       0.5037879
5       731             13             1072       0.4156456
6       731             10              370       0.5598886
```

to see quick summary statistics of our data frame, simply do

```
summary(small)
```

which returns

```
   timedelta     n_tokens_title n_tokens_content n_unique_tokens
 Min.   :  8.0   Min.   : 2.0   Min.   :   0.0   Min.   :  0.0000
 1st Qu.:164.0   1st Qu.: 9.0   1st Qu.: 246.0   1st Qu.:  0.4709
 Median :339.0   Median :10.0   Median : 409.0   Median :  0.5392
 Mean   :354.5   Mean   :10.4   Mean   : 546.5   Mean   :  0.5482
 3rd Qu.:542.0   3rd Qu.:12.0   3rd Qu.: 716.0   3rd Qu.:  0.6087
 Max.   :731.0   Max.   :23.0   Max.   :8474.0   Max.   :701.0000
```

Try the following command on your own! “‘ tail(small, n = 3)

```
## dplyr: A grammar for data manipulation

**dplyr** provides a function for each basic action with data:

- filter() (and slice())
- arrange()
- select() (and rename())
- distinct()
- mutate() (and transmute())
- summarise()
- sample_n() (and sample_frac())

which can do many things together with the **%>%**. We suggest to read this oper\
ator as "and then"

We'll provide examples of how to apply each of the verbs above to our data.

### Installing and loading dplyr
Since dplyr is not preinstalled in R, we need to get it using the command:
```

install.packages(“dplyr”) “‘ At the moment of writing, fall 2016, the latest version is 0.5.0.

Let’s load it into R, assuming the installation went without problems: `library(dplyr)` You will see the following warning

```
##
## 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
```

Don’t worry too much about this for now.

filter()

Our first task is to filter the articles from a specific day `oldest <- filter(small, timedelta == 731 ) head(oldest, n = 2)`

which gives us

```
##   timedelta n_tokens_title n_tokens_content n_unique_tokens
## 1       731             12              219       0.6635945
## 2       731              9              255       0.6047431
```

we can get the same result (try it!) using the operator %>%:

```
small %>% filter(timedelta==731) %>% head(n=2)
```

Now you see why %>% is called “and then”. When you try to read the above command:

> small, and then filter where timedelta is 731, and then take the first two

Cool, eh?

In pure R (without dplyr), we can do

```
oldest <- small[small$timedelta==731,]
head(oldest, n = 2 )
```

which again yields the same result, but in a somewhat less clean way.

slice()

slice() filters rows by position, for instance: `slice(small, 16:20)` returns `## timedelta n_tokens_title n_tokens_content n_unique_tokens ## 1 731 12 682 0.4595420 ## 2 731 8 1118 0.5123967 ## 3 731 8 397 0.6246787 ## 4 731 11 103 0.6893204 ## 5 731 8 1207 0.4105793`

arrange()

arrange() orders columns and helps to break ties.

```
small %>%
  arrange(timedelta, n_tokens_title, n_tokens_content) %>%
  head(n=3)
```

```
##   timedelta n_tokens_title n_tokens_content n_unique_tokens
## 1         8              6              682       0.5394933
## 2         8              8             2509       0.3488781
## 3         8             10              157       0.7019868
```

We can use desc() to arrange a column in descending order.

```
small %>%
  arrange(desc(timedelta), n_tokens_title, n_tokens_content) %>%
  head(n=3)
```

```
##   timedelta n_tokens_title n_tokens_content n_unique_tokens
## 1       731              5              356       0.6182336
## 2       731              6              109       0.6666667
## 3       731              6              174       0.6918605
```

select()

We can use select to, well, select specific columns:

```
small %>% select(timedelta,n_tokens_content) %>% head(n=2)

##   timedelta n_tokens_content
## 1       731              219
## 2       731              255
```

or to avoid specific columns

```
small %>% select(-c(n_tokens_title,n_tokens_content)) %>% head(n=2)

##   timedelta n_unique_tokens
## 1       731       0.6635945
## 2       731       0.6047431
```

We can use select() also to rename columns

```
small %>% select(words_in_title = n_tokens_title) %>% head(n=3)
##   words_in_title
## 1             12
## 2              9
## 3              9
```

however, this drops all the non-selected variables.

rename()

This function is useful to rename a column without dropping the other variables

```
small %>% rename(words_in_title = n_tokens_title) %>% head(n=3)
##   timedelta words_in_title n_tokens_content n_unique_tokens
## 1       731             12              219       0.6635945
## 2       731              9              255       0.6047431
## 3       731              9              211       0.5751295
```

distinct()

This function allows us to find unique values in a table “‘ small %>% distinct(timedelta) %>% head(n=3) ## timedelta ## 1 731 ## 2 730 ## 3 729

```
to count how many of those distinct values are, simply do
```

small %>% distinct(timedelta,n_tokens_title) %>% nrow ## [1] 6792 “‘

mutate()

Sometimes we need to add new columns that are function of existing columns, for instance:

```
small %>%
  mutate(title_to_content = n_tokens_title/n_tokens_content
         ,total_unique = n_tokens_content * n_unique_tokens ) %>%
  head(n=3)

  ##   timedelta n_tokens_title n_tokens_content n_unique_tokens
## 1       731             12              219       0.6635945
## 2       731              9              255       0.6047431
## 3       731              9              211       0.5751295
##   title_to_content total_unique
## 1       0.05479452     145.3272
## 2       0.03529412     154.2095
## 3       0.04265403     121.3523
```

A really amazing feature is that we can recycle newly created variables!

```
small %>%
  mutate(title_to_content = n_tokens_title/n_tokens_content
         ,percentage = round(100*title_to_content,2) ) %>%
  head(n=3)
##   timedelta n_tokens_title n_tokens_content n_unique_tokens
## 1       731             12              219       0.6635945
## 2       731              9              255       0.6047431
## 3       731              9              211       0.5751295
##   title_to_content percentage
## 1       0.05479452       5.48
## 2       0.03529412       3.53
## 3       0.04265403       4.27
```

transmute()

This is like mutate(), but keeps only the newly created variables

```
small %>%
  transmute(title_to_content = n_tokens_title/n_tokens_content
         ,total_unique = n_tokens_content * n_unique_tokens ) %>%
  head(n=3)

  ##   title_to_content total_unique
## 1       0.05479452     145.3272
## 2       0.03529412     154.2095
## 3       0.04265403     121.3523
```

Sampling: sample_n() and sample_frac()

These two functions allow us to sample randomly a fixed number of rows or a fraction. Use `replace = TRUE` for a sample with replacement, and you can add weights for the sampling if needed. Since we won't need this functions very much for now, I'll point you to dplyr's help in `?sample_n`.

Grouping functions

All the functions above become really useful when we can apply them to groups. This is the real meat of the package, in a way.

Suppose we are interested in finding out if news about world events have more images than, say, lifestyle. Let's take a look:

```
gps <- news %>%
  group_by(data_channel_is_lifestyle
          ,data_channel_is_world)%>%
  summarise(count=n()
            ,avg_imgs = mean(num_imgs, na.rm = TRUE)
            , avg_videos =mean(num_videos, na.rm = TRUE))
```

The `group_by()` function takes tuples of values (in this case, from lifestyle, world and all the other categories) and calculates the functions hidden inside the `summarise()` function.

That's quite a bit of a query. Let's see the results, which hopefully make it clearer.

```
  data_channel_is_lifestyle data_channel_is_world count avg_imgs avg_videos
                      <dbl>                 <dbl> <int>    <dbl>      <dbl>
1                         0                     0  2923 5.091687  1.4844338
2                         0                     1   833 2.801921  0.6974790
3                         1                     0   208 4.956731  0.3317308
```

To get an clearer picture of this, let's plot the average number of images against each category:

```
barplot(gps$avg_imgs
        , names.arg = c("Other", "Lifestyle", "World")
        , main = "Average number of images")
```

we obtain

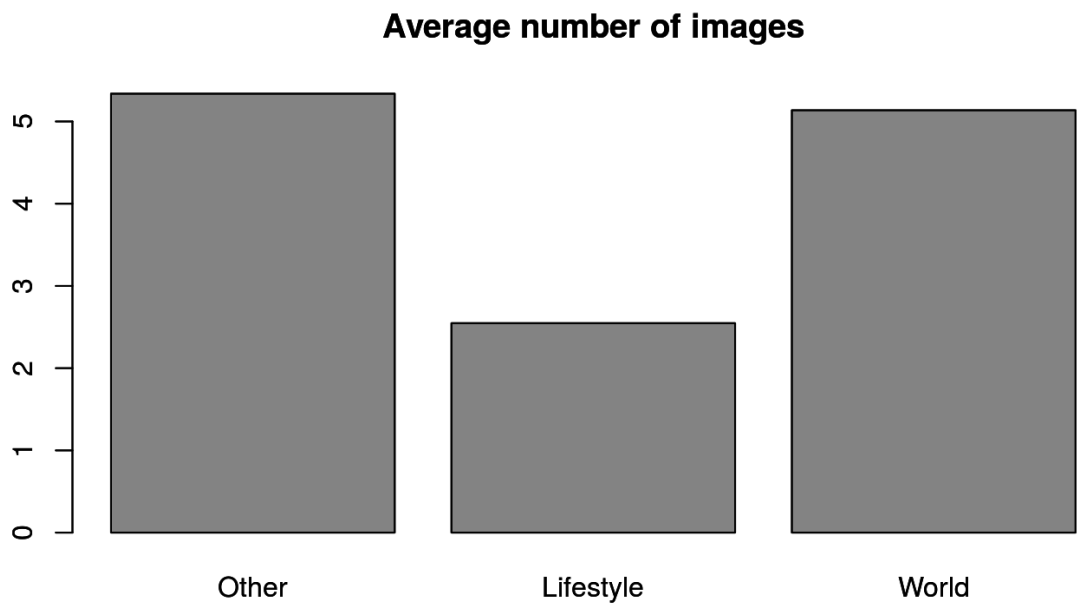


Fig 1.1

Before moving to the exercises, let's draw one more plot that will help you:

```
plot(news$timedelta,news$shares
     , type='l', main = "Number of shares across time"
     , xlab = "Days since acquisition", ylab = "Number of shares")
```

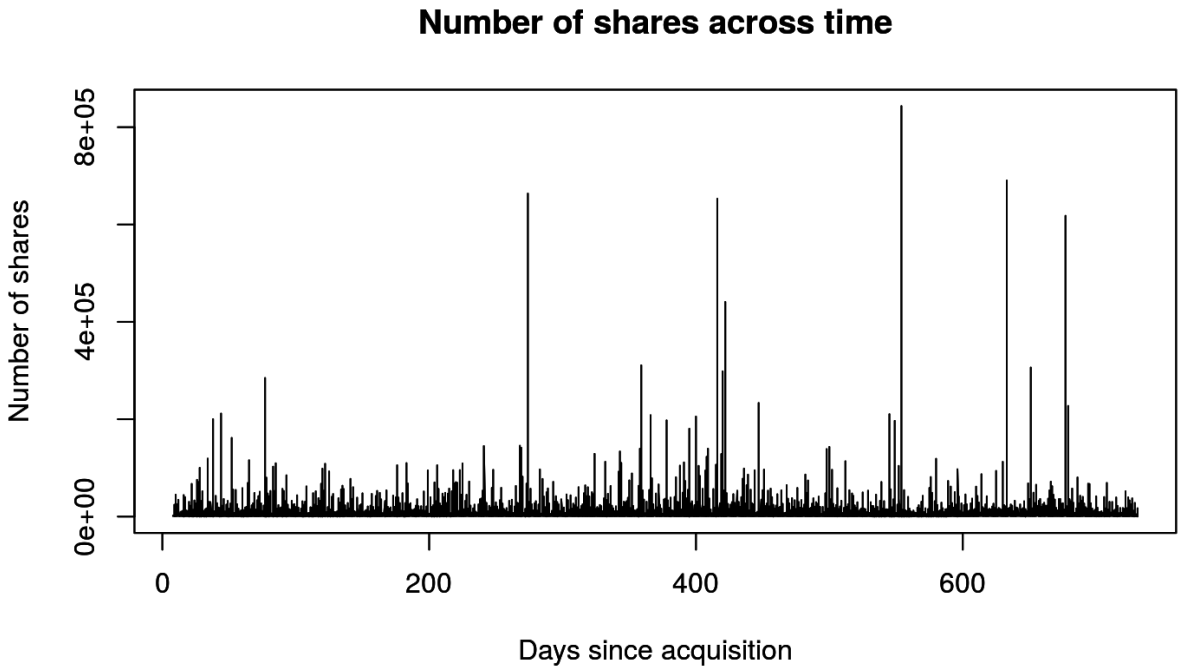


Fig 1.2

Exercises

Try it yourselves! Let's test some hypothesis:

- What are the 5 most shared articles?
- Which channel (among the six described) has the largest average number of shares? does it change across time?
- Which day has the most shares, on average?

Solutions

What are the 5 most shared articles?

A simple application of `arrange()` and `desc`:

```
1 news %>% arrange(desc(shares)) %>% select(url,shares) %>% head(n=5)
```

```
##                                                                 url
## 1                    http://mashable.com/2013/07/03/low-cost-iphone/
## 2             http://mashable.com/2013/04/15/dove-ad-beauty-sketches/
## 3 http://mashable.com/2014/04/09/first-100-gilt-soundcloud-stitchfix/
## 4          http://mashable.com/2013/11/18/kanye-west-harvard-lecture/
## 5                   http://mashable.com/2013/03/02/wealth-inequality/
##   shares
## 1 843300
## 2 690400
## 3 663600
## 4 652900
## 5 617900
```

Which channel has the largest number of shares?

This is a bit more complicated, since we need to use `group_by()` and `summarise()`:

```
ex2a <- news %>%
  group_by(data_channel_is_lifestyle
  , data_channel_is_entertainment
  , data_channel_is_bus
  , data_channel_is_socmed
  , data_channel_is_tech
  , data_channel_is_world)%>%
  summarise(avg_shares = mean(shares))
```

For the plot,

```
barplot(ex2a$avg_shares,
        names.arg =c("Other","LS","Ent","Bus","SM","Tech","World"))
```

which gives the output in Fig 1.3.

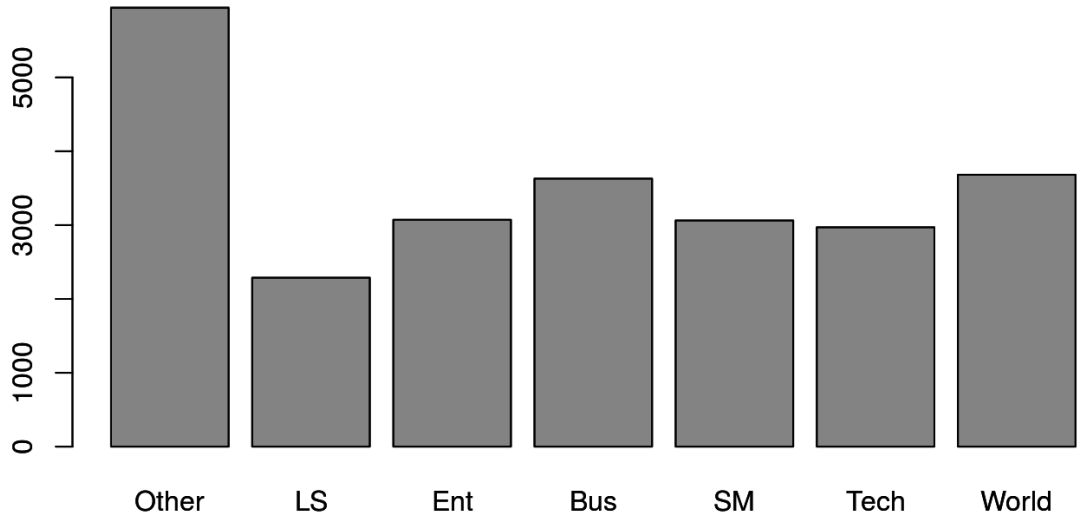


Fig 1.3

So we see that, among the specified categories, World has the most shares in average.

Does it change with time?

We simply need to filter out the category World using `filter()` “‘ ex2b <- news %>% filter(data_-channel_is_world ==1)

plot(ex2b$timedelta,ex2b$shares , type='l', main = “Number of shares across time- World” , xlab = “Days since acquisition”, ylab = “Number of shares”)

```
The output is in Fig 1.4.

![Fig 1.4](images/lp2.png)


*Which day has the most shares, on average?*

This is again a combination of `group_by()` and `summarise()`:
```

ex3 <- news %>% group_by(weekday_is_monday , weekday_is_tuesday , weekday_is_wednesday , weekday_is_thursday , weekday_is_friday , weekday_is_saturday , weekday_is_sunday)%>% summarise(avg_shares = mean(shares)) “‘

The code for the plot is:

```
barplot(ex3$avg_shares,
        names.arg =c("Mon", "Tue",'Wed', "Thu", "Fri", "Sat", "Sun"))
```

which you can see in Fig 1.5.

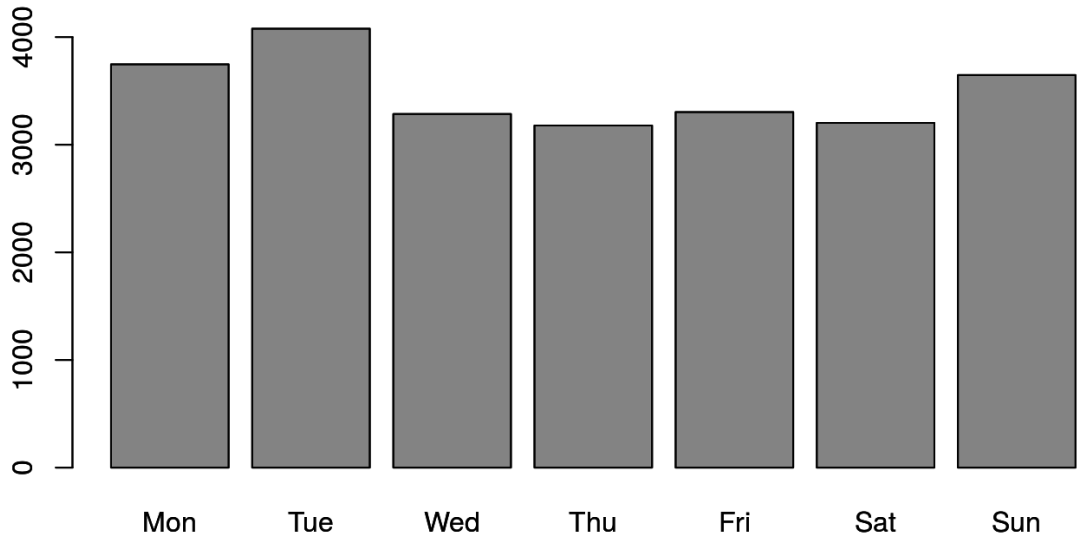


Fig 1.5

Chapter 2: ggplot2

We did some plots on the last chapter, but they where a bit ugly (aesthetically and synthactically). Let's do something else this time. We will work with the same dataset as before.

```
library(ggplot2)
news<-read.csv('./data/OnlineNewsPopularity/OnlineNewsPopularity.csv')
```

Grammar of graphics

A **grammar of graphics** is a structured way to create graphs. We can think of a plot in terms of layers, where each layer has:

- **Data**: No need to explain this one.
- **Aesthetics**: Map variables on the data set to graphic primitives, like size, color, x, y.
- **Geometry**: Visual display and custom parameters.

For example,

```
#Fig 2.1

ggplot(news,aes(x=timedelta,y=shares))+geom_point()
```

tells us to use the `news` dataframe, map the variables `timedelta` and `shares` to the x-y axis respectively, and we will use points to represent the relation between these two. We can see the result in Figure 2.1

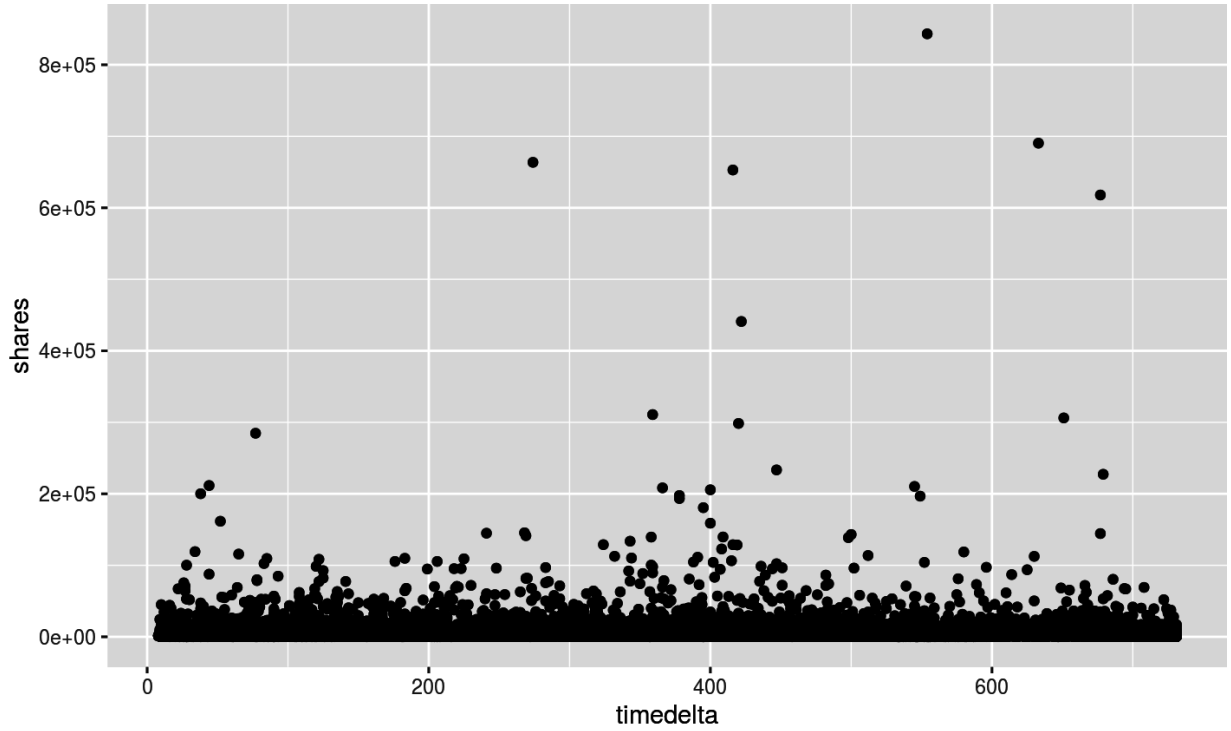


Fig 2.1

We can add color to the aesthetics of our plot, and get the plot in Figure 2.2:

```
#Fig 2.2

ggplot(news,aes(x=timedelta,y=shares,
                color=data_channel_is_lifestyle))+geom_point()
```

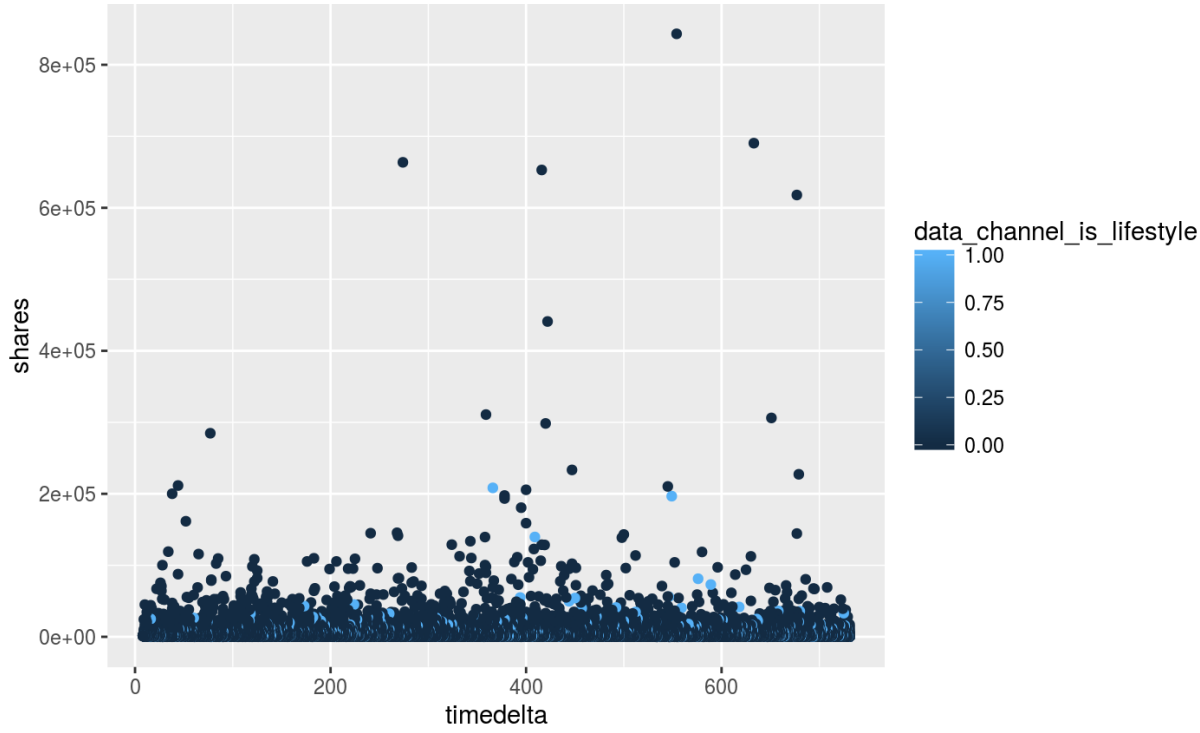


Fig 2.2

Note that, since `data_channel_is_lifestyle` was interpreted as numeric by `dplyr`, we get a continuous color scale. To obtain a discrete color scale, we can coerce this variable to factor, and we get the result in Figure 2.3:

```
#Fig 2.3
ggplot(news,aes(x=timedelta,y=shares,
                color=as.factor(data_channel_is_lifestyle)))+geom_point()
```

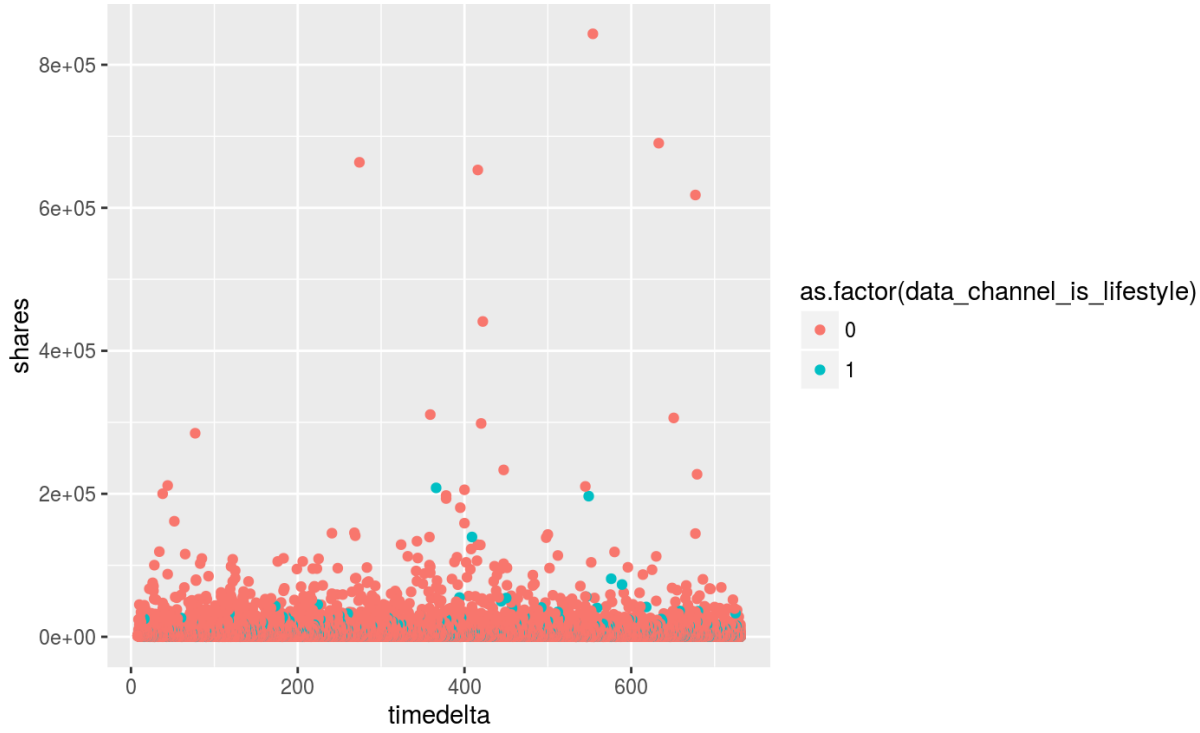


Fig 2.3

We can fit more than two dimensions of our data in a two-dimensional coordinate system, if we use colors and shapes. For instance, Figure 2.4 shows four dimensions!

```
#Fig 2.4

ggplot(news,aes(x=timedelta,y=shares,
                color=as.factor(data_channel_is_lifestyle),
                shape=as.factor(data_channel_is_bus)))+geom_point()
```

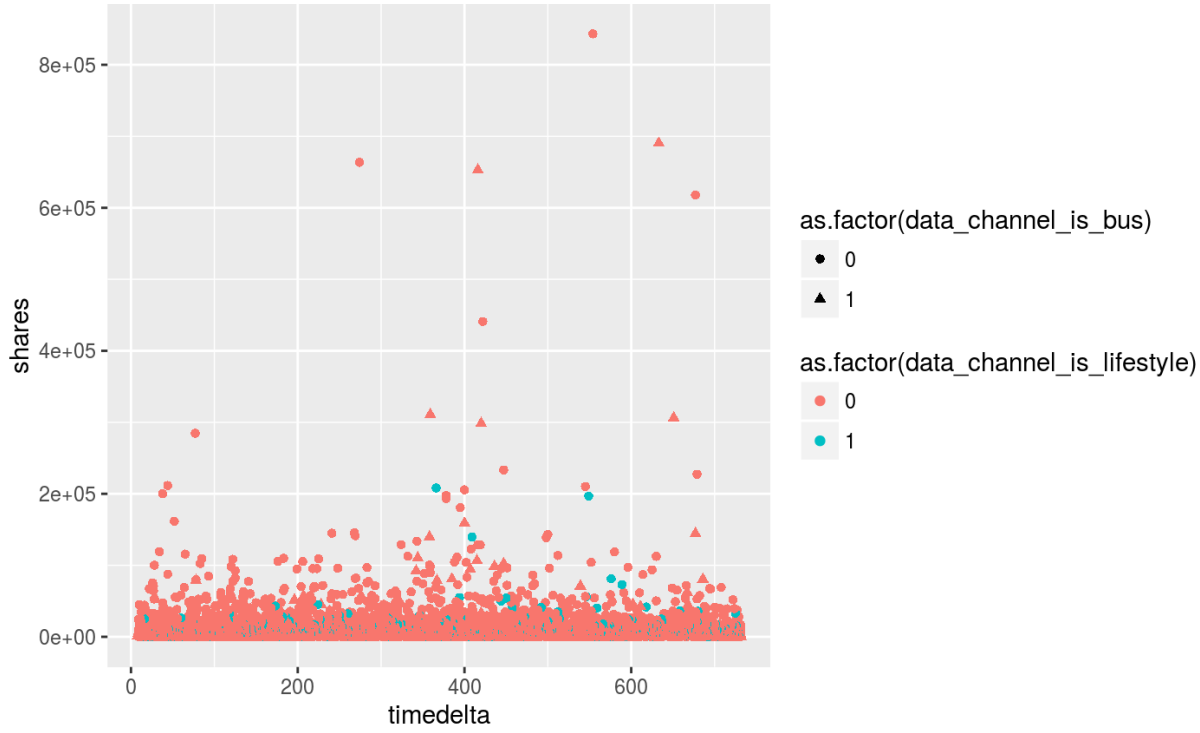


Fig 2.4

ggplot2 and dplyr

A great advantage of `ggplot2` is that is fully compatible with `dplyr`. Let's revisit the plot from our previous exercise:

```
ex2a <- news %>%
  group_by(data_channel_is_lifestyle
  , data_channel_is_entertainment
  , data_channel_is_bus
  , data_channel_is_socmed
  , data_channel_is_tech
  , data_channel_is_world)%>%
  summarise(avg_shares = mean(shares))

ex2a$channel <- c("Other","LS","Ent","Bus","SM","Tech","World")
```

In particular, we can use the pipe operator!

```
#Fig 2.5

ex2a %>%
  ggplot(aes(x=channel, y=avg_shares))+geom_bar(stat = "identity")
```

which yields Fig 2.5,

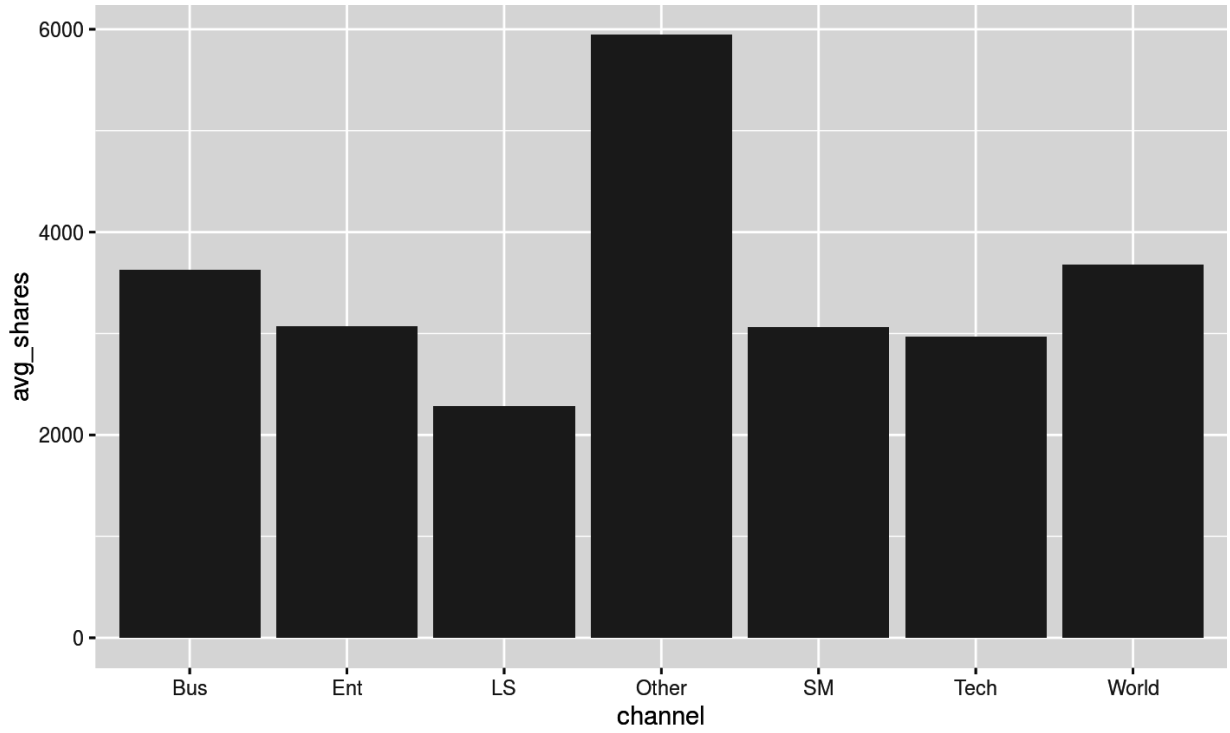


Fig 2.5

and make even nicer plots, such as Fig 2.6:

```
#Fig 2.6

q <- ex2a %>%
    ggplot(aes(x=channel, y=avg_shares,
               fill = avg_shares))+geom_bar(stat = "identity")+
  scale_fill_continuous(name="Average shares")
```

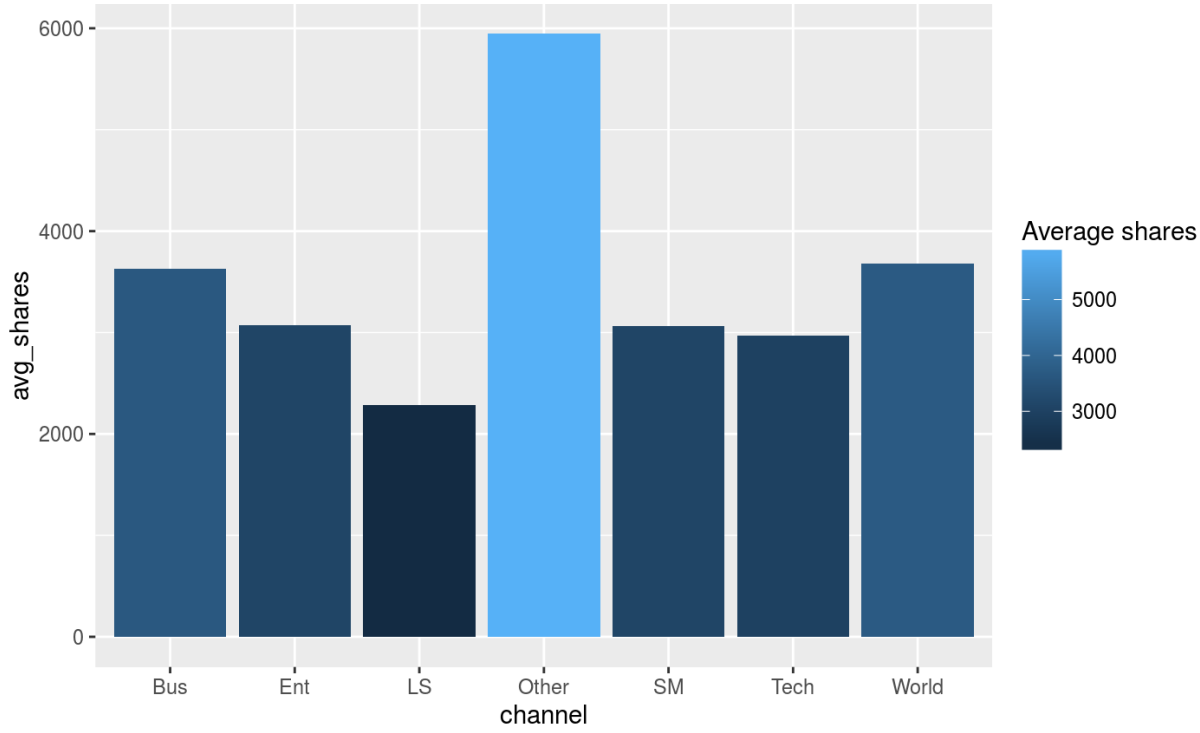


Fig 2.6

Wait, what's the "stats" there? The issue is that some plots visualize transformations of the data. We can combine `stat` with `geom`, for instance:

```
news %>%
  select(is_weekend) %>%
  ggplot(aes(x=as.factor(is_weekend)))+geom_bar(stat="count")
```

and

```
news %>%
  select(is_weekend) %>%
  ggplot(aes(x=as.factor(is_weekend)))+stat_count(geom = "bar")
```

yield the same plot (try it!)

Diving deeper into the grammar of graphics

Let's go a bit more in detail with the grammar of graphics, using instead the `diamonds` dataset, included in `ggplot`, that contains the price of different diamonds and their attributes.

Two basic constructs:

- **plot**: coord, scale, facet and layers
- **layer**: data mapping, stat, geom, position

The plot is the canvas on which we paint, the layer is the things we paint there. Let's see how to do some common plots:

Continuous vs continuous

That's pretty much like the first plot we saw:

```
# Fig 2.7

diamonds %>%
  ggplot(aes(x=carat,y=price))+
  geom_point()
```

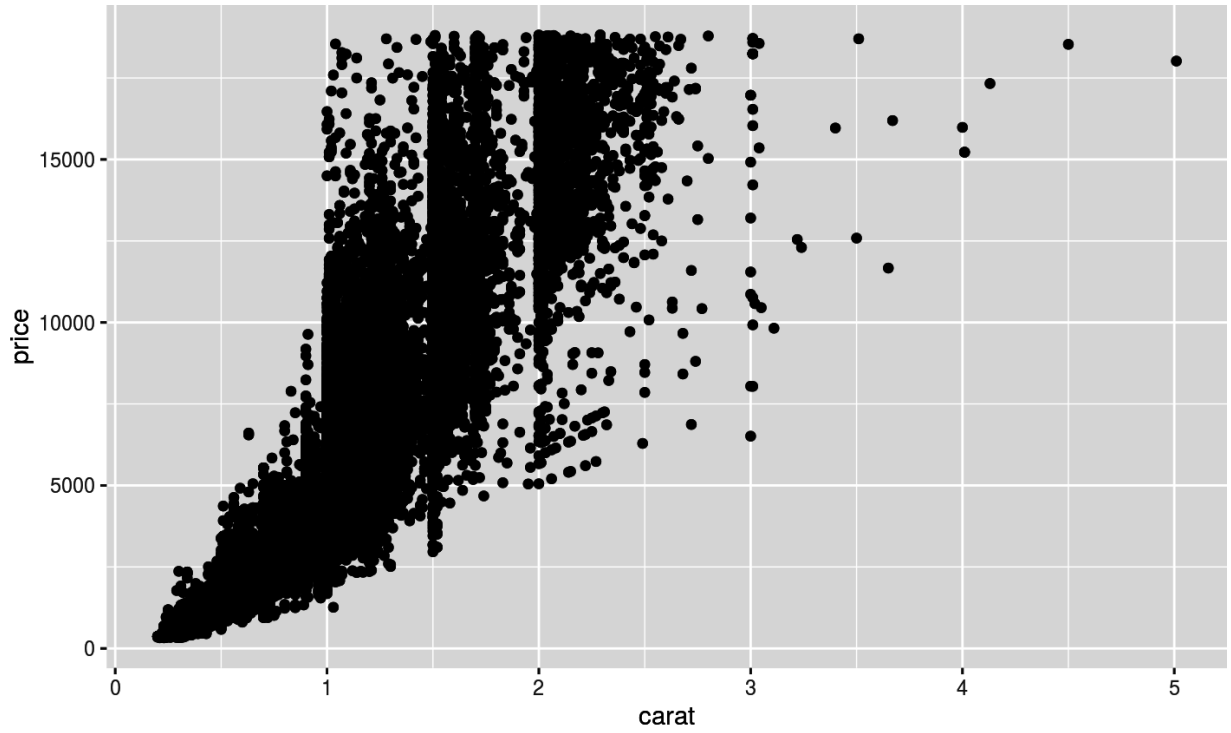


Fig 2.7

Discrete vs Continuous

```
# Fig 2.8
diamonds %>%
  ggplot(aes(x=cut,y=price))+
  geom_point()
```

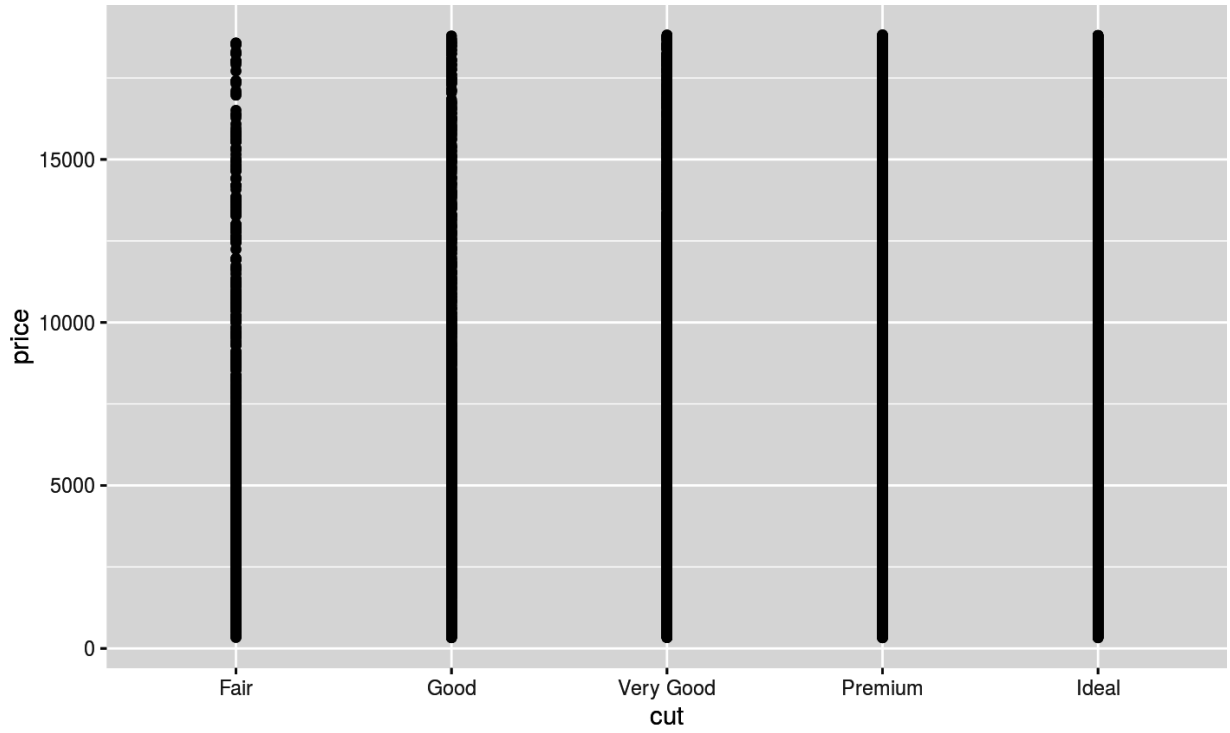


Fig 2.8

This plot is not very useful, because all the dots are on the same vertical line. We can add a bit of jitter on it:

```
# Fig 2.9
p <-
diamonds %>%
  ggplot(aes(x=cut,y=price))+
  geom_point(position = "jitter")
```

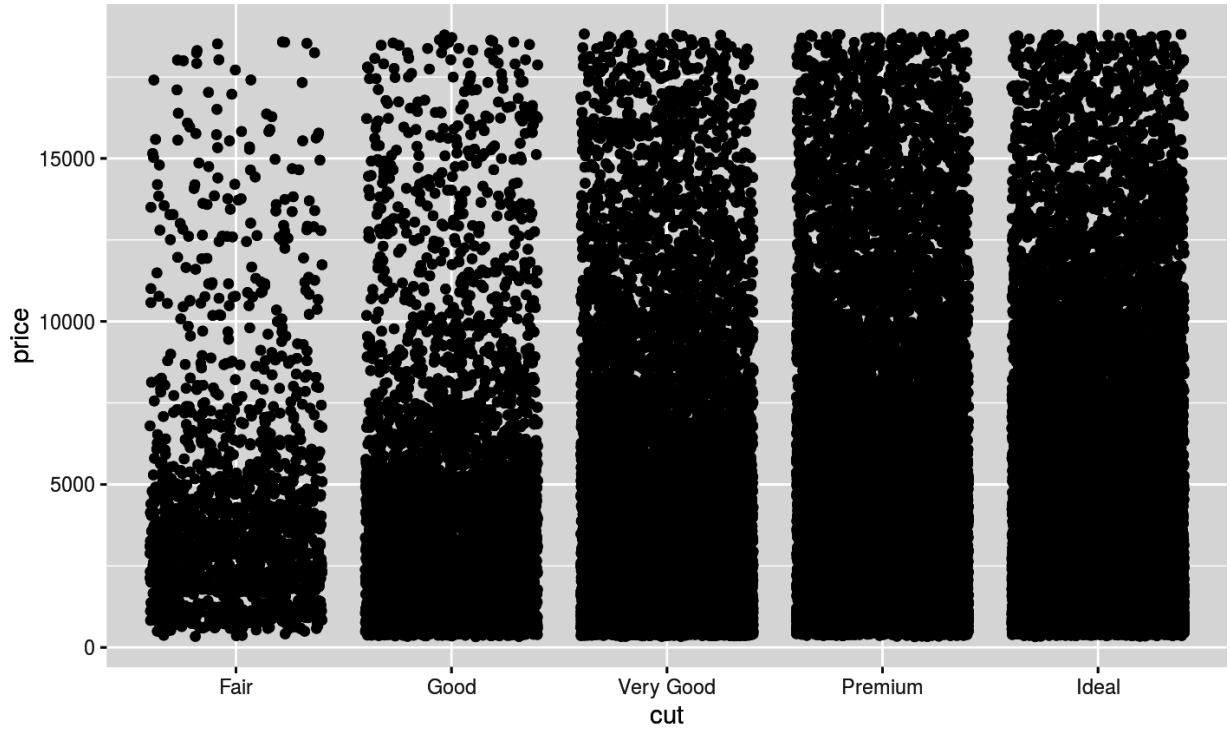


Fig 2.9

We need not restrict ourselves to points or lines, we can change the `geom`, too!

```
# Fig 2.10
p <-
diamonds %>%
  ggplot(aes(x=cut,
      y=price))+
  geom_boxplot(position = "dodge")
```

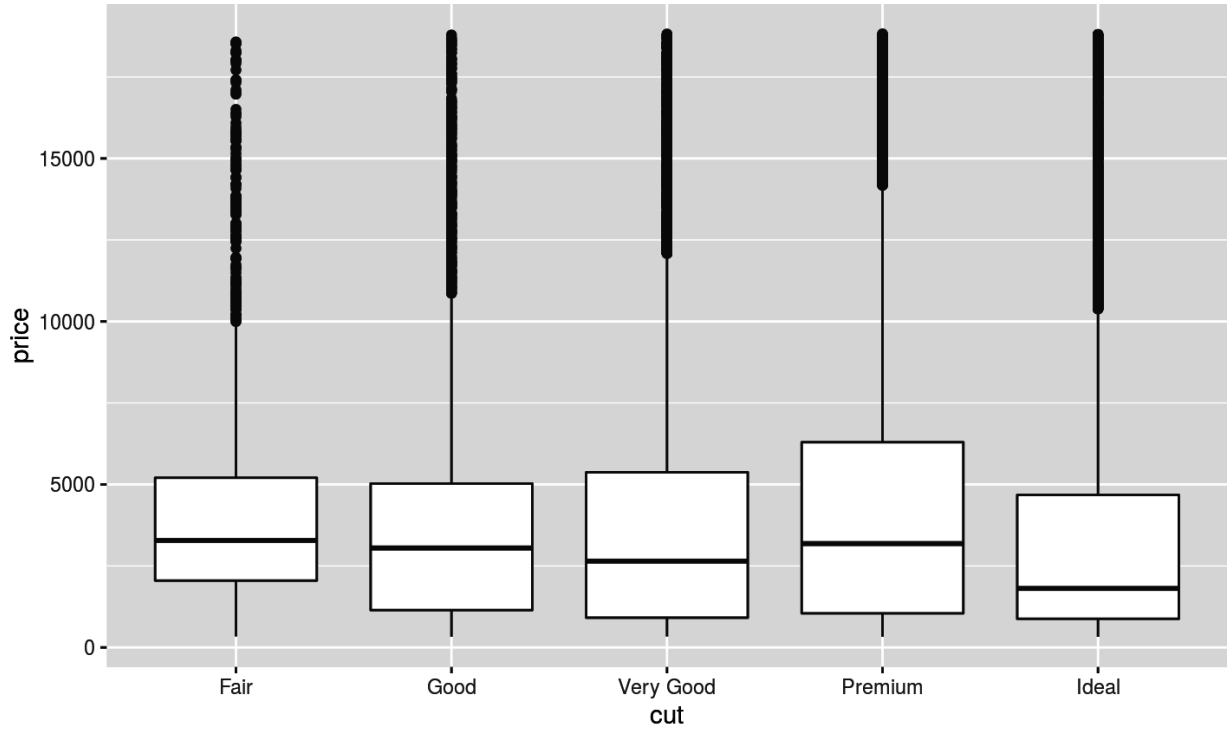


Fig 2.10

A really useful feature that comes from the abstraction is that combining layers is quite natural:

```
# Fig 2.11
p <-
diamonds %>%
  ggplot(aes(x=cut,y=price))+
  geom_point(position = "jitter")+
  geom_boxplot(position = "dodge",
               fill = "blue",
               color="red",
               alpha=0.3)
```

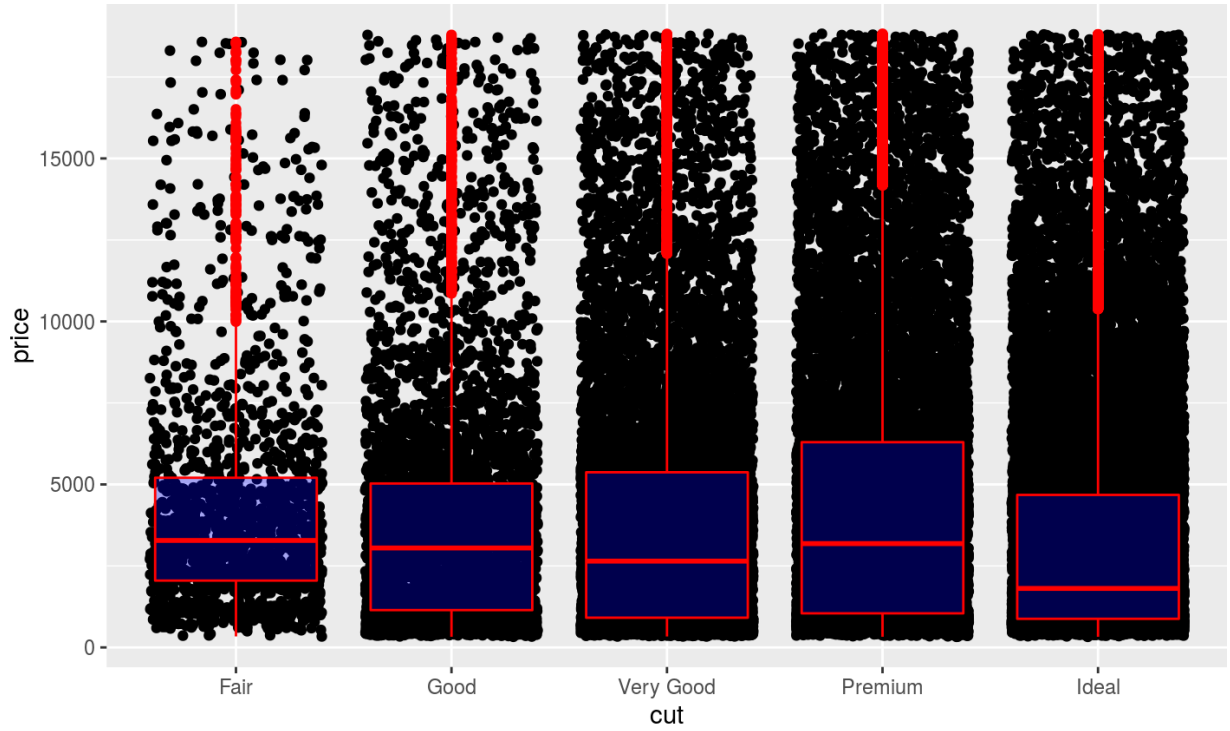


Fig 2.11

We saw already that adding colors or shapes is useful to represent more information. We can go even further on the slicing and dicing of our data for analysis using facets:

```
# Fig 2.12
p <-
diamonds %>%
  ggplot(aes(x=carat,
             y=price,
             color=color))+
  scale_color_hue()+
  geom_point()+
  facet_wrap(~cut)
```

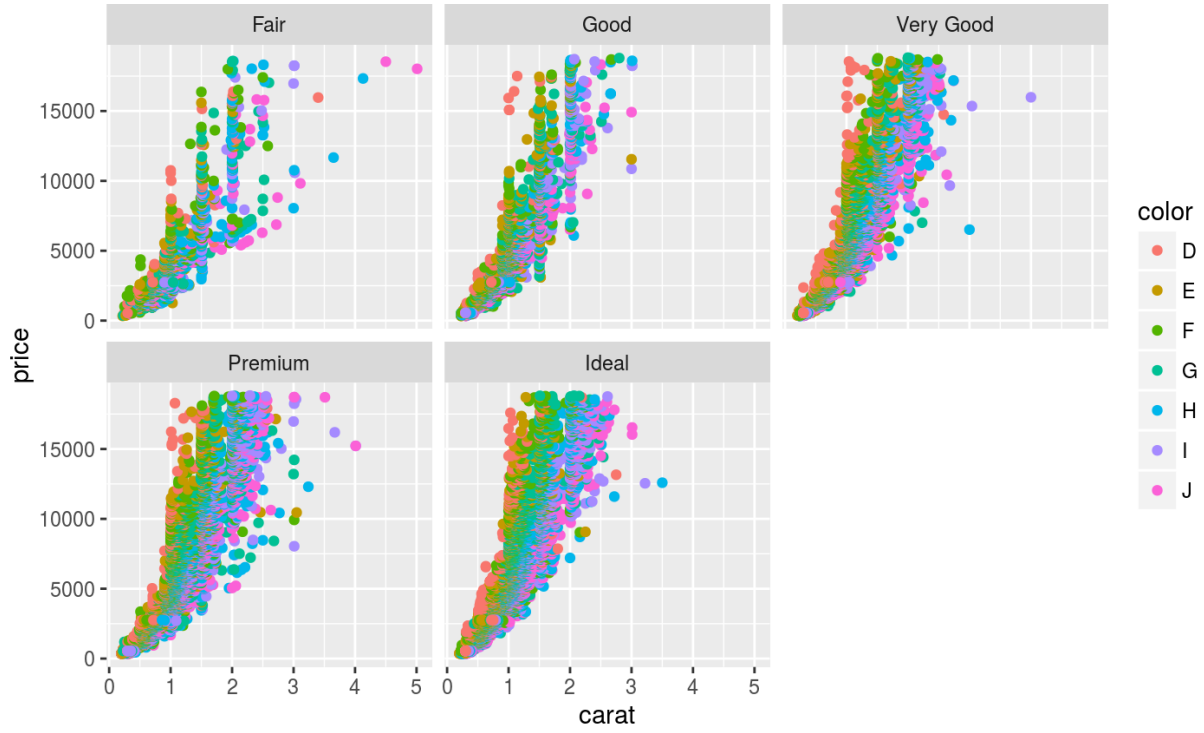


Fig 2.12

We can add a fitted curve, to highlight some trends in our data:

```
#Fig 2.13

p <-
diamonds %>%
  ggplot(aes(x=carat,
             y=price,
             color=color))+
  scale_color_hue()+
  geom_point()+
  geom_smooth(aes(x=carat,y=price),
              stat="smooth",
              method="loess")+
  facet_wrap(~cut)
```

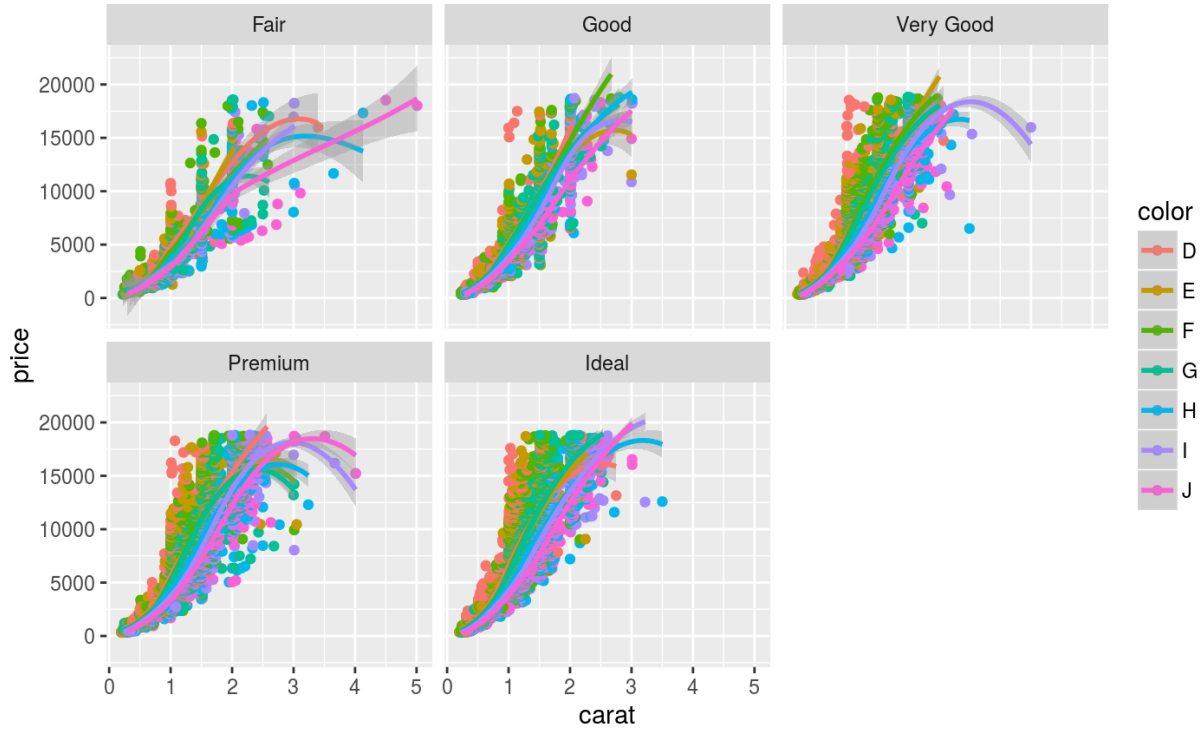
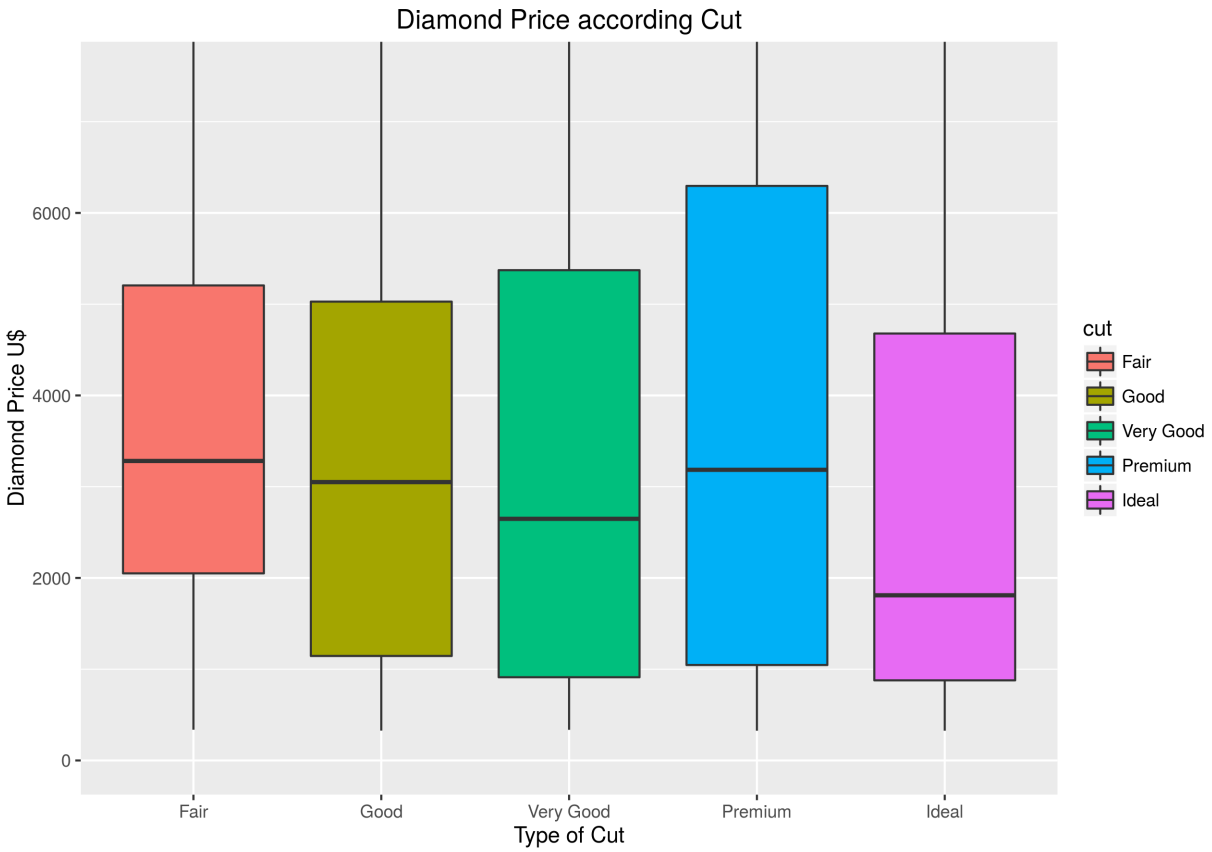


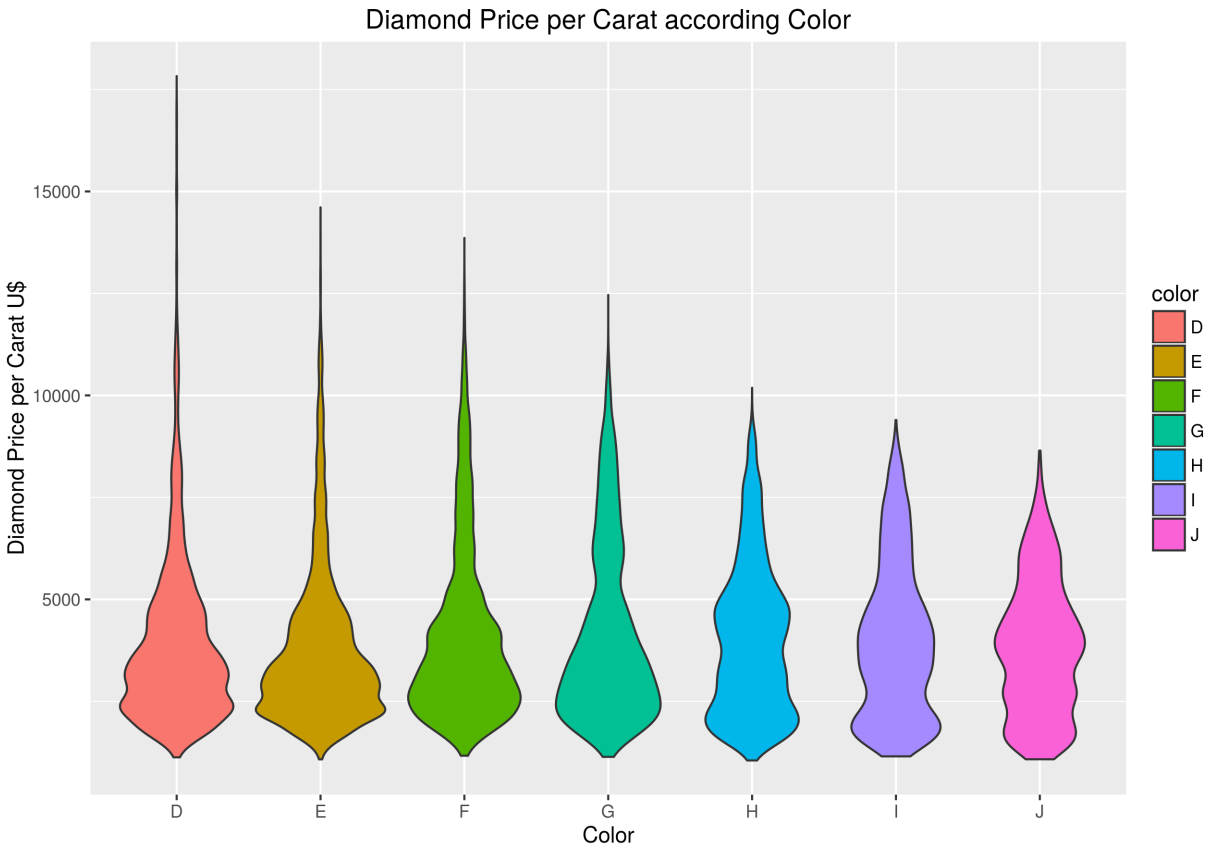
Fig 2.13

Exercises

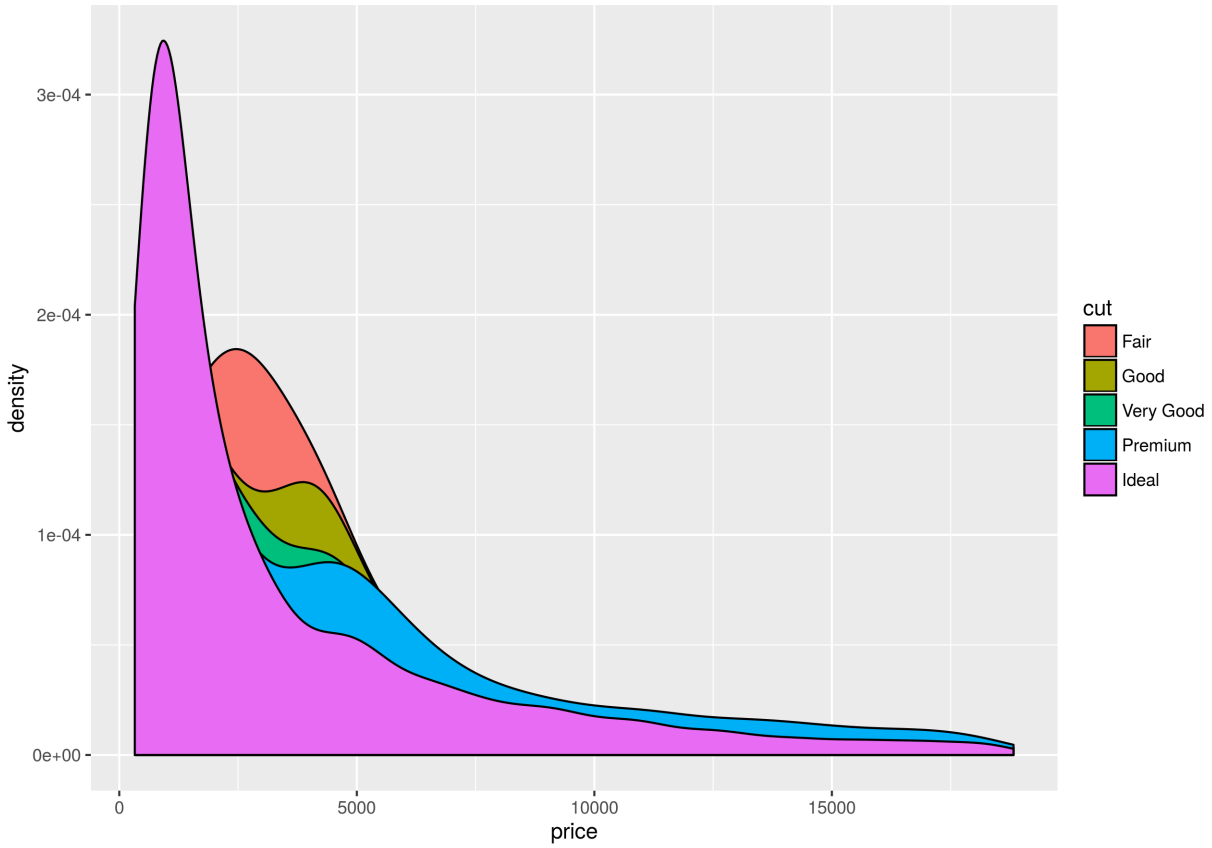
Reproduce the following plots.


Diamond Price according Cut
Diamond Price U$
6000
4000
2000
0
Fair
Good
Very Good
Premium
Ideal
Type of Cut
cut
Fair
Good
Very Good
Premium
Ideal

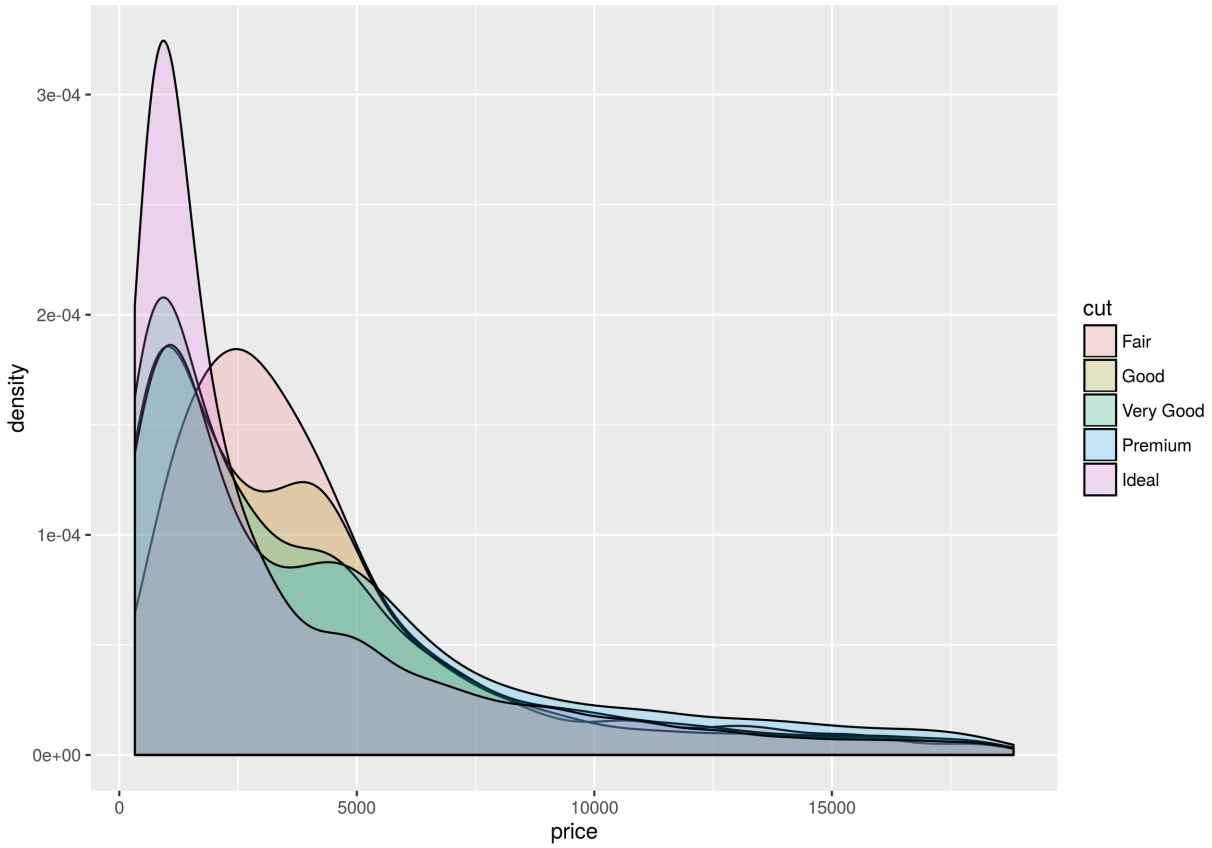
Exercise 1


Diamond Price per Carat according Color
Diamond Price per Carat U$
15000
10000
5000
D
E
F
G
H
I
J
Color
color
D
E
F
G
H
I
J

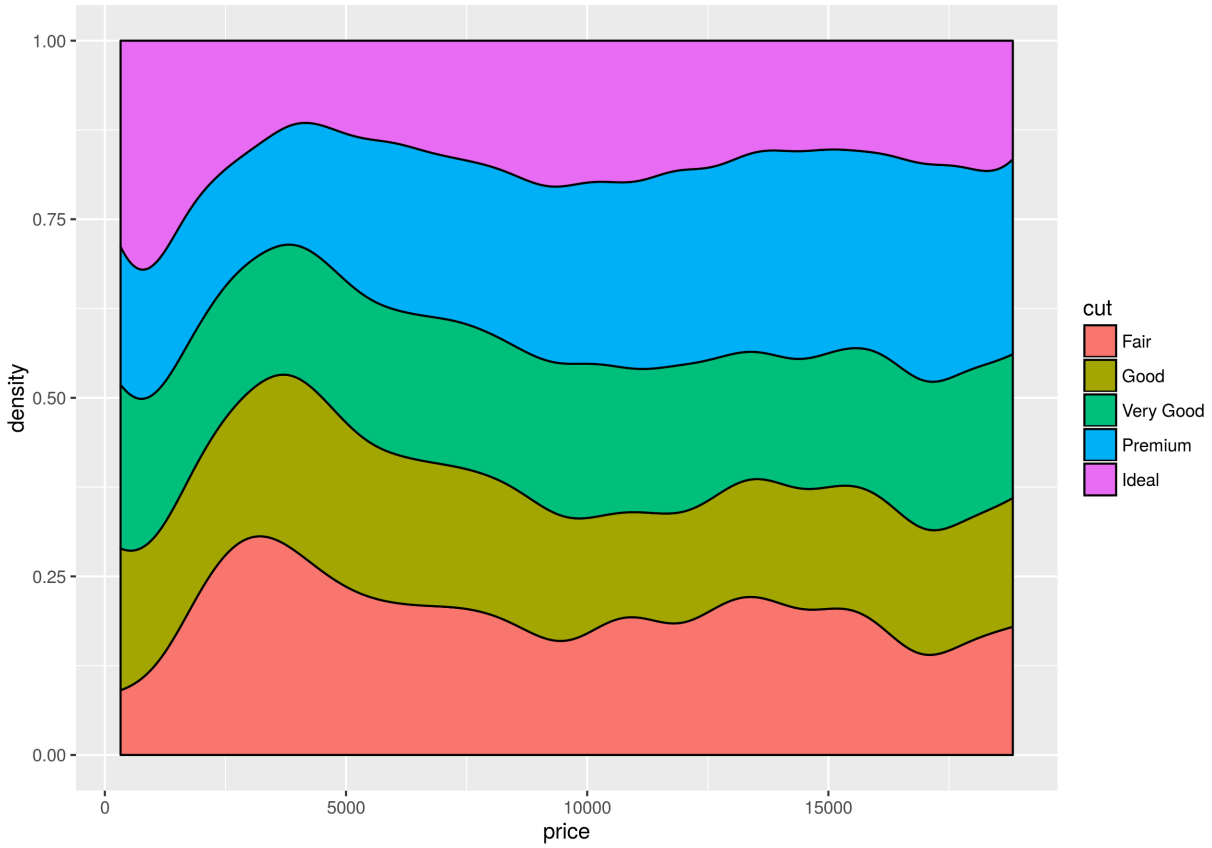
Exercise 2


3e-04
2e-04
1e-04
0e+00
density
0
5000
10000
15000
price
cut
Fair
Good
Very Good
Premium
Ideal

Exercise 3


3e-04
2e-04
1e-04
0e+00
density
0
5000
10000
15000
price
cut
Fair
Good
Very Good
Premium
Ideal

Exercise 4



Exercise 5

Solutions

Exercise 1

diamonds %>% ggplot(aes(factor(cut), price, fill=cut)) + geom_boxplot() + ggtitle("Diamond Price according Cut") + xlab("Type of Cut") + ylab("Diamond Price U$") + coord_cartesian(ylim=c(0,7500))

Exercise 2

diamonds %>% ggplot(aes(factor(color), (price/carat), fill=color)) + geom_violin() + ggtitle("Diamond Price per Carat according Color") + xlab("Color") + ylab("Diamond Price per Carat U$")

Exercise 3

ggplot(data=diamonds,aes(x=price, group=cut, fill=cut)) + geom_density(adjust=1.5)

Exercise 4

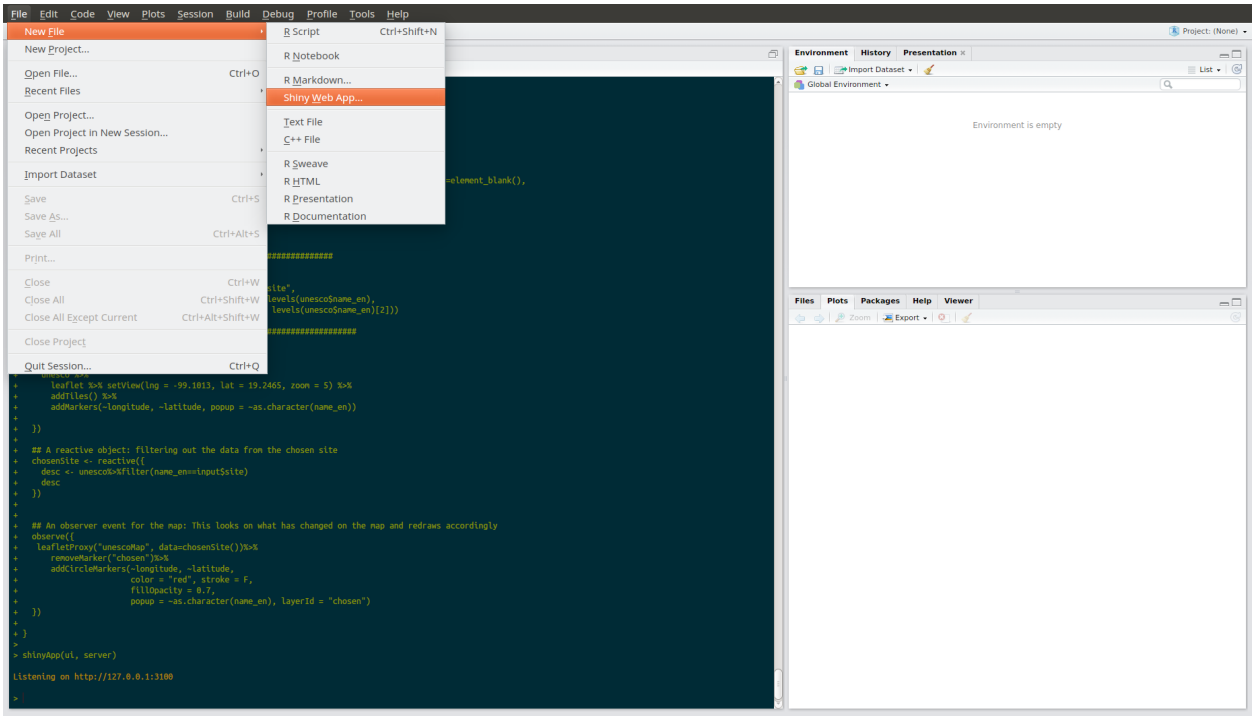
ggplot(data=diamonds,aes(x=price, group=cut, fill=cut)) + geom_density(adjust=1.5 , alpha=0.2)

Exercise 5

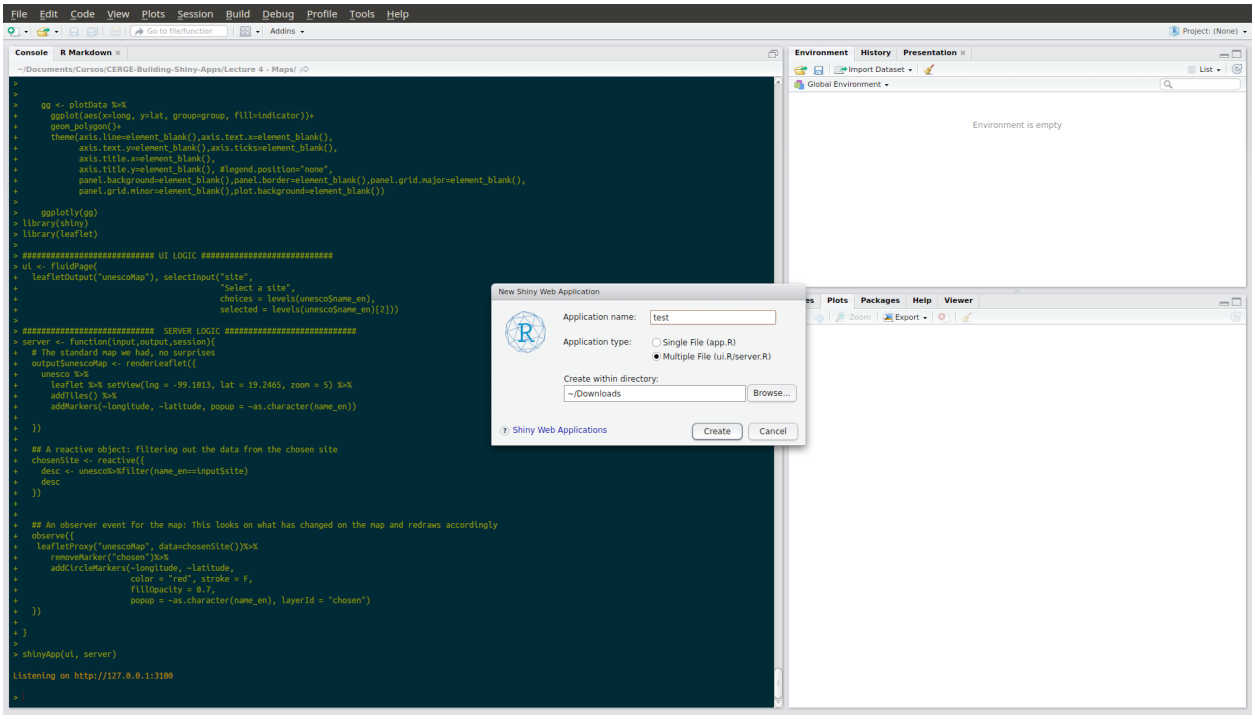
ggplot(data=diamonds,aes(x=price, group=cut, fill=cut)) + geom_density(adjust=1.5, position="fill")

Chapter 3: Shiny

To get quickly started with Shiny, go to RStudio, create a New File and choose “Shiny Web App”



You will see a dialog box as below. Choose `ui.R` and `server.R`.



Now, automatically we will see two files, with code in the `ui.R` file looking like this:

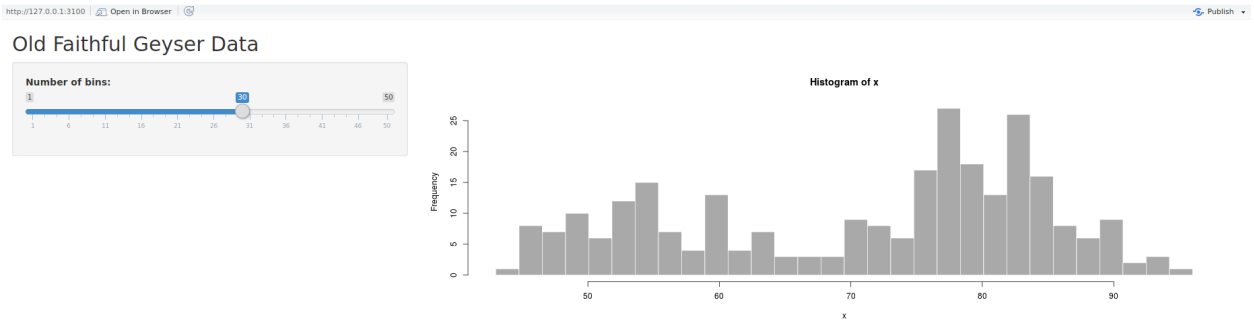
```
library(shiny)

# Define UI for application that draws a histogram
shinyUI(fluidPage(

  # Application title
  titlePanel("Old Faithful Geyser Data"),

  # Sidebar with a slider input for number of bins
  sidebarLayout(
    sidebarPanel(
       sliderInput("bins",
                   "Number of bins:",
                   min = 1,
                   max = 50,
                   value = 30)
    ),

    # Show a plot of the generated distribution
    mainPanel(
       plotOutput("distPlot")
```

```
    )
  )
))
```

whereas the `server.R` file looks like:

```
library(shiny)

# Define server logic required to draw a histogram
shinyServer(function(input, output) {

  output$distPlot <- renderPlot({

    # generate bins based on input$bins from ui.R
    x    <- faithful[, 2]
    bins <- seq(min(x), max(x), length.out = input$bins + 1)

    # draw the histogram with the specified number of bins
    hist(x, breaks = bins, col = 'darkgray', border = 'white')

  })

})
```

If you run the app with the “Run App” button, you will get:



Let's see what's going on here: the ui file has two parts: a `sidebarPanel`, where the slider widget lives, and a `mainPanel`, which holds the plot of a histogram. This plot changes according to the number of bins chosen in the slider.

How is that worked out by R? Well, look at the first argument of the `sliderInput` function in `ui.R`. This argument, "bins" is the **id** of the widget. On `server.R` it is read as `input$bins`, meaning that whatever the value of the slider (passed as a string to the server, more on that later) is stored in `input$bins`. On the server side, this information is processed and a histogram calculated accordingly.

How is this communicated back to the user? Well, by symmetry we have an `output` object, that carries on this communication. Note that in `server.R` the output of the histogram is assigned to `output$distPlot`, and that "distPlot" is called in the `plotOutput` function in `ui.R`!

This is the essential communication mechanism in Shiny, if you get what happens here, you are set!.

Shiny supports tons of stuff and it's really worth going through the documentation in their official website, http://shiny.rstudio.com/[2]. This book is meant to get you quickly started, but once you got the basics, you should really dig on the documentation.

Shiny default graphics is not very nice, so instead we will build a dashboard using an additional package, shinydashboard[3] which gives a nicer layout for business analysts' dashboards.

[2] http://shiny.rstudio.com/

[3] https://rstudio.github.io/shinydashboard/get_started.html

Chapter 4: Building Dashboards in Shiny

Shinydashboard package

We willl use the shinydashboard[4] library, that offers a very nice layout for creating dashboards.

```
# install.packages("shinydashboard")
library(shiny)
library(shinydashboard)
```

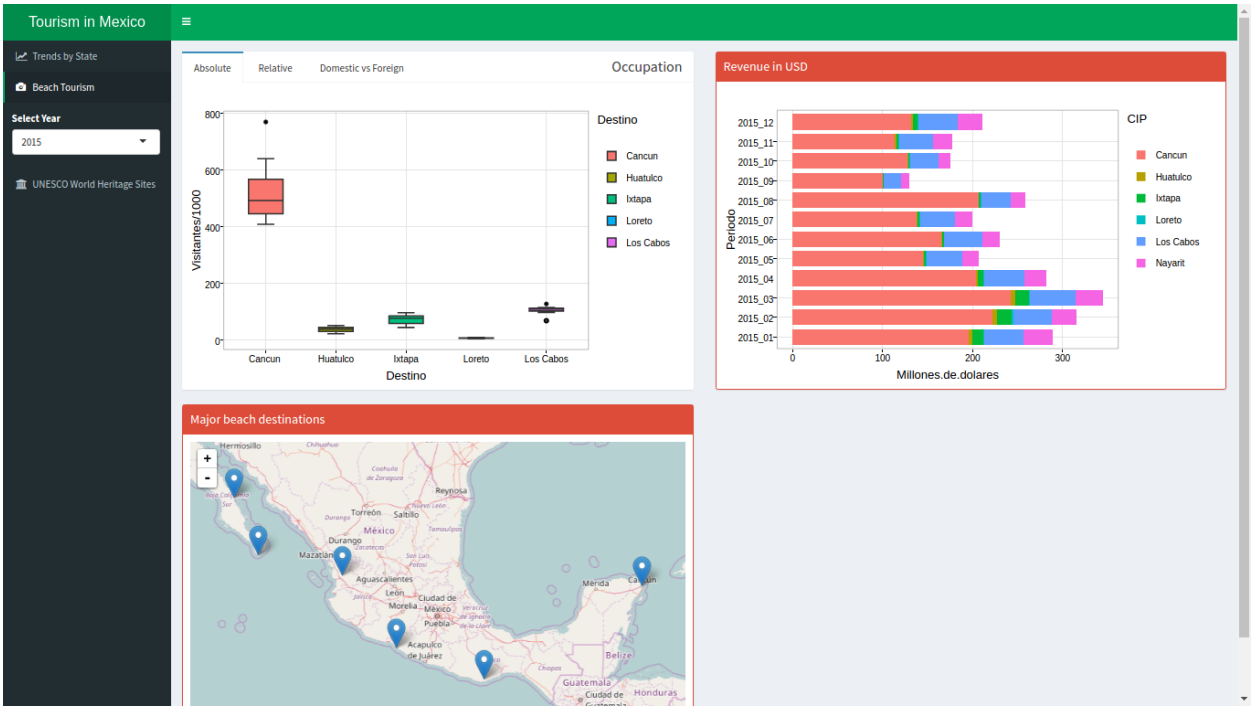
Our goal

We will wrap up our knowledge of R packages by building a dashboard. You will be guided through the building blocks of a dashboard, but try to do the work yourself! The full dashboard is available via Shinyapps in https://pablomaldonado.shinyapps.io/tourism_full/[5] and you can see the source code in the following Github repository: https://github.com/jpmaldonado/tourism_dashboard_mx[6]

Let's start by having an idea of what we want. These are called **wireframes**. This should be the minimum specification we should have in mind befre coding, to have an idea of the functionality we will have to implement. Not having wireframes in the beginning might be inconvenient for small apps, but it's a killer on bigger apps! Many architectural decisions depend on having a vision of the final product before doing any coding.

So we will try to get something like this:

[4] https://rstudio.github.io/shinydashboard/get_started.html
[5] https://pablomaldonado.shinyapps.io/tourism_full/
[6] https://github.com/jpmaldonado/tourism_dashboard_mx



We will not deal with maps now, but leave it for the next session.

Building blocks of our UI

Let's start with the `ui.R` file.

```
# ui.R
library(shiny)
library(shinydashboard)

dashboardPage(
  dashboardHeader(title="HEADER"),
  dashboardSidebar("SIDEBAR"),
  dashboardBody("BODY")
)
```

That should give us something like:

We can customize the sidebar a bit:

```
# ui.R
library(shiny)
library(shinydashboard)

  dashboardHeader(title="HEADER"),
  dashboardSidebar(
    sidebarMenu(
      menuItem("One item", tabName = "tab1", icon=icon("archive")),
      menuItem("Another item", tabName = "tab2",
               icon=icon("camera", "glyphicon"))
    )

  ),
  dashboardBody("BODY")
)
```

On the dashboard we can add custom icons from Glyphicon and FontAwesome. We can also add global controls, as we will show below, and much more. Check `?dashboardSidebar` for more options.

This is all very nice, but we need to bind the user interface with some server logic.

Graphic elements in the user interface are connected to server logic as follows:

- On server side, renderX functions
- On ui side, XOutput functions

For instance, `renderPlot` vs `plotOutput`, `renderUI` vs `uiOutput`, etc.

Drafting our first plot

It's good idea to draft the first plot on a new script before plugging it into the app.

```
    act_df <- read.csv("tourist_activity.csv")

    gg <-   act_df %>%
            filter(grepl('2016', Periodo)) %>%
            ggplot(aes(x=Destino, y=Visitantes/1000,
                       fill = Destino))+
            geom_boxplot()+theme_bw()

    ggplotly(gg)
```

Once we are happy with the plot, we can plug it in the result. In the `ui.R` file, we should have now

```
#ui.R
library(shiny)
library(shinydashboard)
dashboardPage(
  dashboardHeader(title="HEADER"),
  dashboardSidebar(
    sidebarMenu(
      menuItem("One item", tabName = "tab1", icon=icon("archive")),
      menuItem("Another item", tabName = "tab2",
               icon=icon("camera", "glyphicon")),
      , selectInput("year", "Select year",
                    choices = c("2016","2015"), selected = "2016")
    )
  ),
  dashboardBody(plotlyOutput("bpDV"))
)
```

whereas on the `server.R` file we should have:

```
#server.R

shinyServer(function(input, output) {

    output$bpDV <- renderPlotly({
      gg <-   act_df %>%
            filter(grepl(input$year, Periodo)) %>%
            ggplot(aes(x=Destino, y=Visitantes/1000,
                       fill = Destino))+
            geom_boxplot()+theme_bw()

```

```
    ggplotly(gg)
  })
}
```

For small apps like this we can even fit everything on a single `app.R` file. These are called **single-app** files, for obvious reasons, and are no different from usual apps with `ui.R` and `server.R` files.

```
library(shiny)
library(shinydashboard)

ui <- dashboardPage(
  dashboardHeader(title="HEADER"),
  dashboardSidebar(
    sidebarMenu(
      menuItem("One item", tabName = "tab1", icon=icon("archive")),
      menuItem("Another item", tabName = "tab2", icon=icon("camera", "glyphicon"\
)),
      selectInput("year", "Select year",
                  choices = c("2016","2015","2014","2013"), selected = "2016")
    )

  ),
  dashboardBody(
    tabItems(

      tabItem(tabName = "tab1"

      )

      , tabItem(tabName = "tab2",
                fluidRow(
                  tabBox(
                    title = "Occupation"
                    , tabPanel("Absolute"
                               #,plotlyOutput("bpDV")
                               )
                    , tabPanel("Relative"
                               #,plotlyOutput("pctChart")
                               )
                    , tabPanel("Domestic vs Foreign"
                               #, plotlyOutput("barPlots")
                               )
```

```
                    )
                    ,
                    box(status="danger",solidHeader = T,
                        title = "Revenue in USD"
                    )

                    , box(status = "danger",  solidHeader = T, title = "Occupation\
 by State")
                )

      )

    )
  )
)

server <- function(input, output) { }

shinyApp(ui, server)
```

Exercise:

Given the basic layout, try to replicate the structure of the wireframes.

Reactive expressions

One important concept in Shiny apps is that of **reactive** expressions. These are useful to improve the performance of an app. As a side effect, they help to keep the code more organized.

- Reactive expressions save their results on the first run.
- The next time a reactive expression is called, it checks if the widgets (the user interface elements) have changed their inputs.
- If the value is out of date, the reactive object will recalculate it (and then save the new result).
- If the value is up-to-date, the reactive expression will return the saved value without doing any computation.

For example, instead of calculating `act_df %>% filter(grepl(input$year, Periodo))` for every plot, we hide it away on a reactive expression:

```
#server.R
  activity <- reactive({
    act_df %>%
      filter(grepl(input$year, Periodo))
  })
```

when we need it, we can call it as a **function** (adding `()` in the end):

```
  output$bpDV <- renderPlotly({
    gg <- activity()%>%
      ggplot(aes(x=Destino, y=Visitantes/1000, fill = Destino))+
      geom_boxplot()+theme_bw()

    ggplotly(gg)
  })
```

Reactive expressions are useful for the following reasons, not necessarily in that order:

- To factor the code and make it more organized (not the most important use case).
- They help us speed up performance of the app by recalculating only what is *truly* needed (meaning that it changed).
- To reduce, in more complicated apps, unnecessary calls to the database or to external APIs.

Chapter 5: Maps

R is great for graphics and visualizations, including maps.

Although there are tons of packages out there for doing maps in R, we will focus on Leaflet only. Later on this chapter we will briefly mention out to use `ggplot2` and shapefiles to do maps as well.

Leaflet

Leaflet is a very popular open-source JavaScript library for interactive maps. It's used by a number of organizations around the globe, including GIS specialists (for instance, OpenStreetMap).

Let's load an example file (use a separate R script for this)

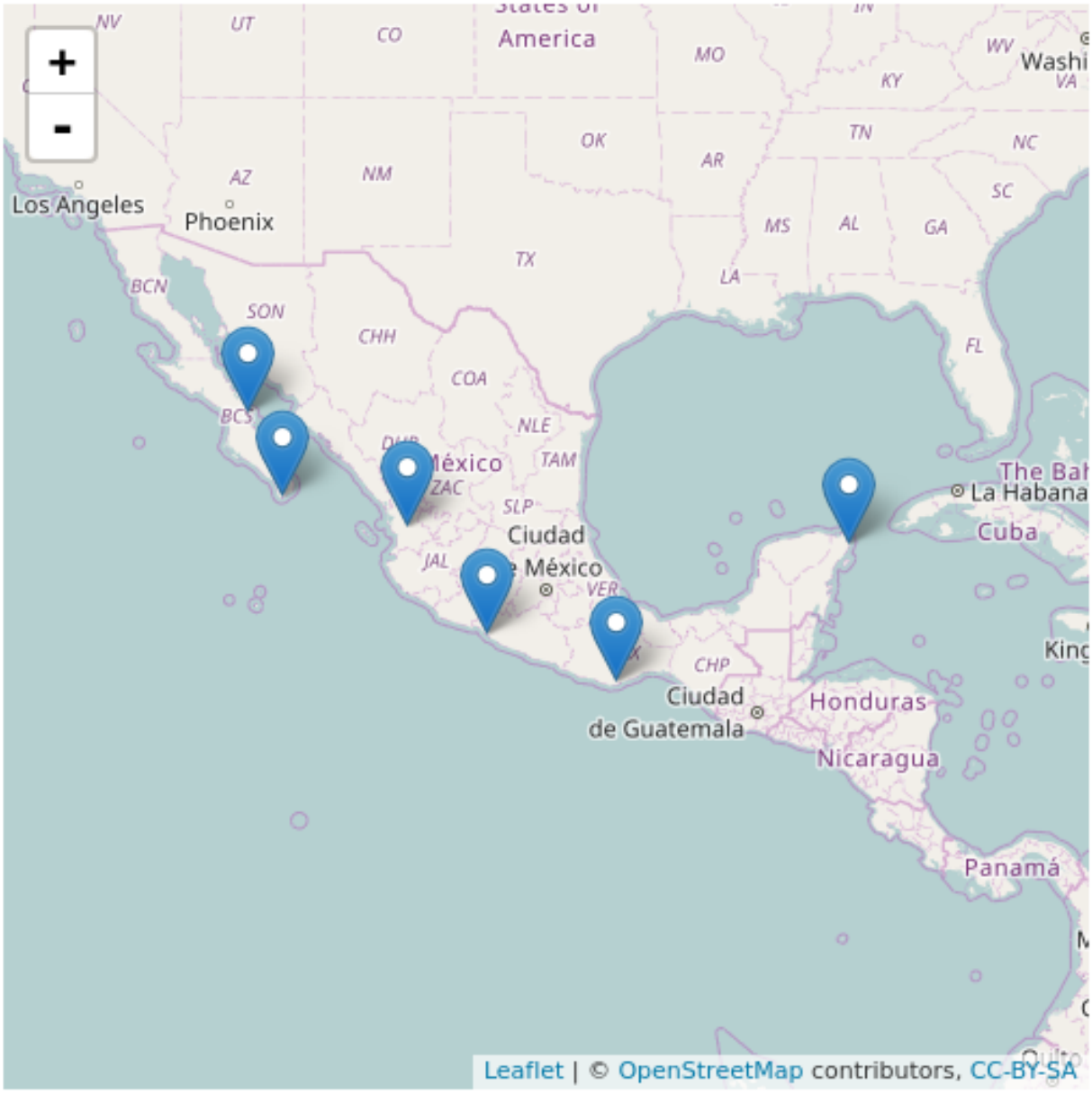
```
library(leaflet)

coords <- read.csv("./dashboard_app/coords.csv")

head(coords)
```

	Destino <fctr>	Lat <dbl>	Long <dbl>
1	Loreto	26.01180	-111.3478
2	Los Cabos	22.89050	-109.9167
3	Ixtapa	17.66260	-101.5873
4	Nayarit	21.75140	-104.8455
5	Huatulco	15.83400	-96.3199
6	Cancun	21.16056	-86.8475

6 rows

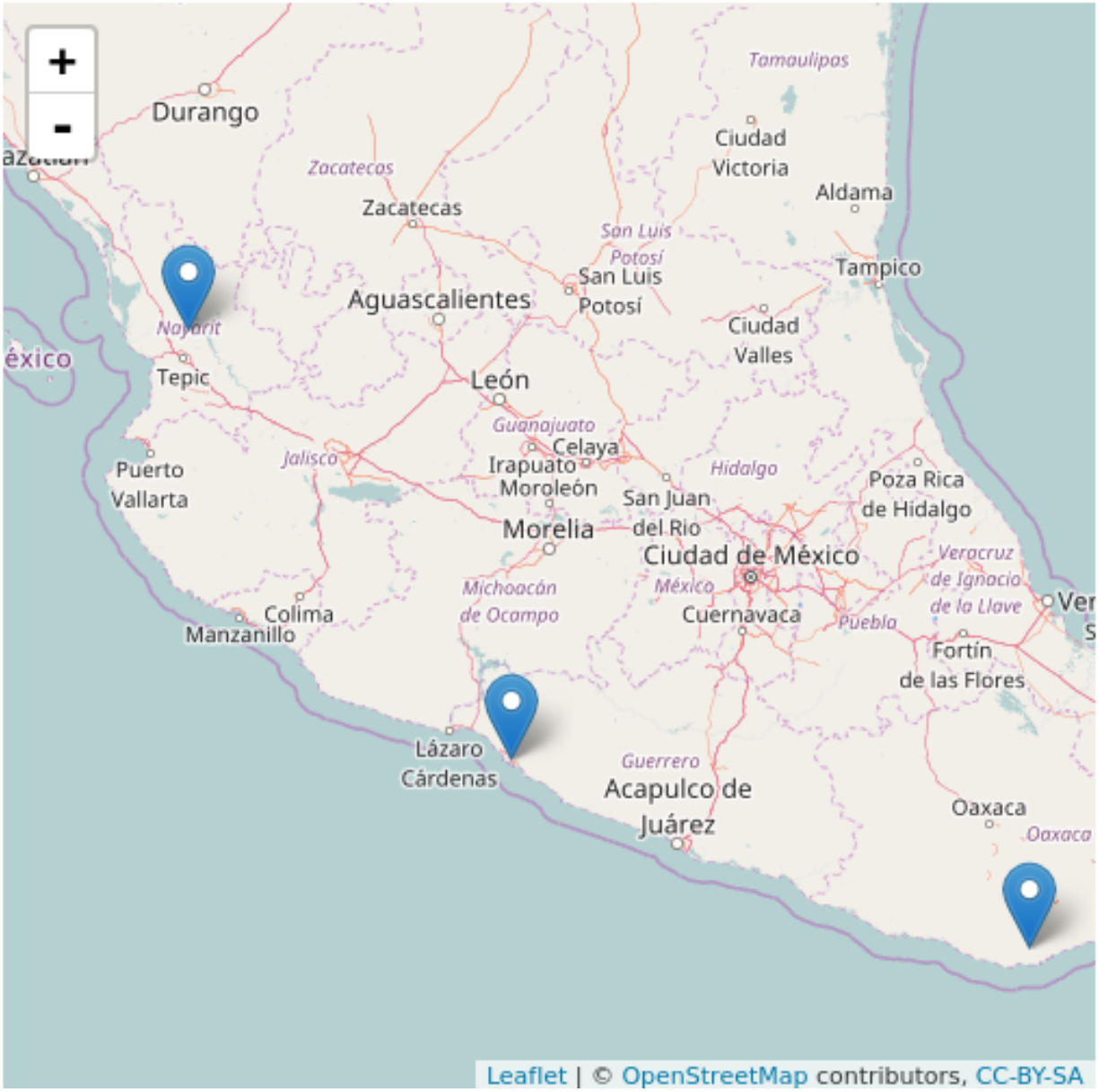
Let's create a map out of these coordinates:

```
coords %>%
      leaflet %>%
      addTiles() %>%
      addMarkers(~Long, ~Lat, popup = ~as.character(Destino))
```



We can specify the zoom level and the center of the view:

```
coords %>%
      leaflet %>%
      setView(lng=-101.1950 , lat=19.7060, zoom=6) %>%
      addTiles() %>%
      addMarkers(~Long, ~Lat, popup = ~as.character(Destino))
```



Maps created with Leaflet can be used seamlessly in Shiny, using the functions `renderLeaflet` and `leafletOutput` in `server.R` and `ui.R` respectively.

Leaflet maps in Shiny

Let's try something else with the maps, to complete the next part of the wireframe.

```
library(leaflet)
library(dplyr)
unesco <- read.csv('./dashboard_app/unesco.csv')
unesco %>% select(name_en, longitude,latitude)%>%head
```

	name_en <fctr>	longitude <fctr>	latitude <fctr>
1	Sian Ka'an	-87.792	19.383
2	Pre-Hispanic City and National Park of Palenque	-92.050	17.483
3	Historic Centre of Mexico City and Xochimilco	-99.133	19.418
4	Pre-Hispanic City of Teotihuacan	-98.842	19.692
5	Historic Centre of Oaxaca and Archaeological Site of Monte Albán	-96.722	17.062
6	Historic Centre of Puebla	-98.208	19.047

6 rows

So we can very easily create the map with markers as before.

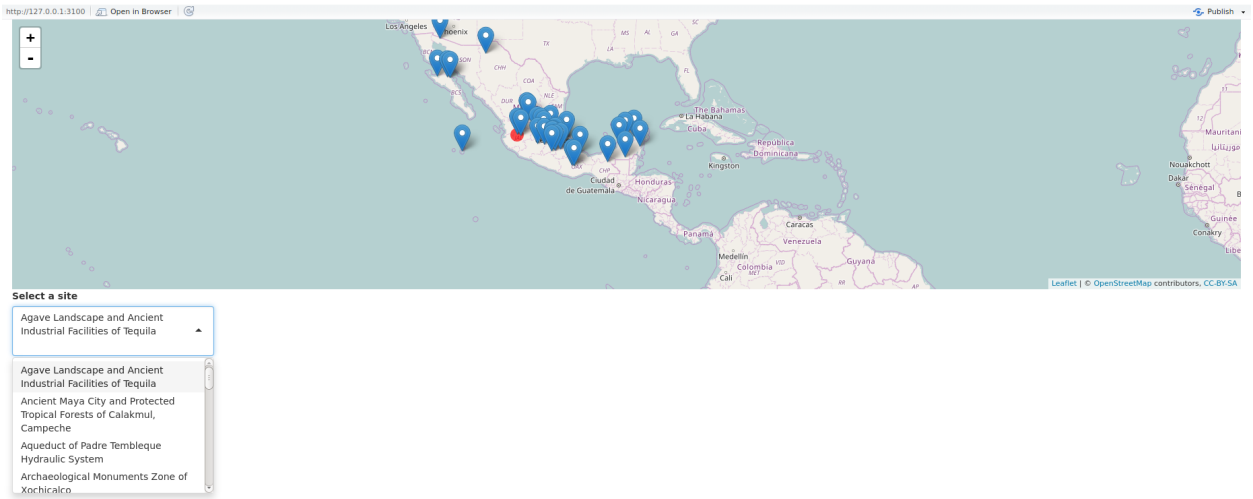
```
unesco %>%
      leaflet %>% setView(lng = -99.1013, lat = 19.2465, zoom = 5) %>%
      addTiles() %>%
      addMarkers(~longitude, ~latitude #, popup = ~as.character(name_en)
                 )
```

The tiles for these maps are coming from OpenStreetMaps. Since it's quite heavy to load over and over again we need to "preload" the map if we would like this map to change the map to user's reaction. In code, we need to do something like this:

```
library(shiny)
library(leaflet)

############################  UI LOGIC ############################
ui <- fluidPage(
      leafletOutput("unescoMap"), selectInput("site",
                            "Select a site",
                            choices = levels(unesco$name_en),
                            selected = levels(unesco$name_en)[2]))

############################  SERVER LOGIC ############################
server <- function(input,output,session){
 # The standard map we had, no surprises
  output$unescoMap <- renderLeaflet({
    unesco %>%
      leaflet %>% setView(lng = -99.1013, lat = 19.2465, zoom = 5) %>%
      addTiles() %>%
      addMarkers(~longitude, ~latitude, popup = ~as.character(name_en))

  })

  ## A reactive object: filtering out the data from the chosen UNESCO site

  chosenSite <- reactive({
    desc <- unesco%>%filter(name_en==input$site)
    desc
  })


  ## An observer event for the map: This looks on what has changed on the map an\
d redraws accordingly
  observe({
    proxy <- leafletProxy("unescoMap", data=chosenSite())%>%
        removeMarker("chosen")%>%
      addCircleMarkers(~longitude, ~latitude,
                       color = "red", stroke = F,
                       fillOpacity = 0.7,
                       popup = ~as.character(name_en), layerId = "chosen")
  })

}

```

```
43 shinyApp(ui, server)
```

After running this code, we should see the following in an external window (or in the browser):



Choropleth maps

Chropleths are maps in which the areas are shaded (or something is drawn into them) in proportion of a measurement being displayed.

```
1 library(rgdal)
2 library(rgeos)
3 library(maptools)
4
5 mexico <- readOGR(dsn="./dashboard_app/",layer ="mexstates", encoding = "UTF-8")
6 hoteles <- read.csv("./dashboard_app/hoteles.csv")
7 hoteles[3:9] <- sapply(hoteles[3:9], function(x) as.numeric(as.character(x))/100\
8 0)
```

```
OGR data source with driver: ESRI Shapefile
Source: "./dashboard_app/", layer: "mexstates"
with 32 features
It has 15 fields
```

We will use now **shapefiles**. This file format was introduced with ArcView GIS in the early 90s. On its current iteration is possible to read and write geographical datasets with a variety of software, including open-source software like R.

The term "shapefile" is a bit misleading, because the file format consists of actually three compulsory files:

- *.shp*: The shapefile, the geometry features itself.
- *.shx*: A positional index of the feature geometry that allows to look information backward and forward quickly.
- *.dbf*: Columnar attributes for each shape, in dBase IV format (one of the oldest database formats!).

Other files might be present, which describe other geographical indices and other information.

Let's take a look at the data we loaded:

```
1 head(hoteles)
```

	Ano <int>	Estado <fctr>	Occupation_Pct <dbl>	Total_Visitors <dbl>	Total_Visitors_Domestic <dbl>
1	2014	Aguascalientes	0.05071963	530.503	482.2459
2	2014	Baja California	0.04029333	3640.601	2780.1160
3	2014	Baja California Sur	0.05940306	1740.469	681.2464
4	2014	Campeche	0.05064734	1466.950	1225.9420
5	2014	Chiapas	0.03544209	3542.911	3190.1487
6	2014	Chihuahua	0.03965934	3908.207	3694.9905

6 rows | 1-6 of 9 columns

We will keep

```
1 hotels2014 <- hoteles%>%
2           filter(Ano==2014) %>%
3           rename(id=Estado) %>%
4           mutate(indicator = Total_Visitors)
```

** Warning:** to pass column names in Shiny to `dplyr`, you need to add an underscore(_) to the function name. This is because Shiny passes only text strings, which need to be dealth with differently. So you would write, instead of the above code:

```
hotels2014 <- hoteles%>%
          filter(Ano==input$year) %>%
          rename(id=Estado) %>%
          mutate_(indicator = input$chosenCol)
```

Let's come back to the shapefile. The shapefile object has a data frame, which is accessible via `@data`:

```
head(mexico@data)
```

	OBJECTID <int>	FIPS_ADMIN <fctr>	GMI_ADMIN <fctr>	ADMIN_NAME <fctr>	FIPS_CNTRY <fctr>	GMI_CNTRY <fctr>	CNTRY_NAME <fctr>	POP_ADMIN <int>
0	888	MX06	MEX-CHH	Chihuahua	MX	MEX	Mexico	2656214
1	933	MX07	MEX-CDZ	Coahuila	MX	MEX	Mexico	2145539
2	976	MX19	MEX-NLE	Nuevo Leon	MX	MEX	Mexico	3370912
3	978	MX28	MEX-TML	Tamaulipas	MX	MEX	Mexico	2272724
4	998	MX25	MEX-SIN	Sinaloa	MX	MEX	Mexico	2397706
5	1004	MX10	MEX-DRN	Durango	MX	MEX	Mexico	1467826

6 rows | 1-9 of 15 columns

It often happens, as in this case, that we do not have the data we want to plot readily available in the shapefile, so we need to get it somehow. This is simple to do:

```
mexico@data <- left_join(mexico@data,hotels2014, by= c("ADMIN_NAME"="id"))
```

Now we are ready to plot. Note that we can add custom HTML formating to our map.

```
state_popup <- paste0("<strong>Estado: </strong>",
                      mexico@data$ADMIN_NAME,
                      "<br><strong>Total Visitors in 2014: </strong>",
                      mexico@data$Total_Visitors)

leaflet(data=mexico)%>%
  addProviderTiles("CartoDB.Positron")%>%
  addPolygons(fillColor = ~pal(as.numeric(as.character(mexico$Occupation_Pct))),
              fillOpacity = 0.8,
              color = "#BDBDC3",
              weight = 1,
              popup = state_popup)
```

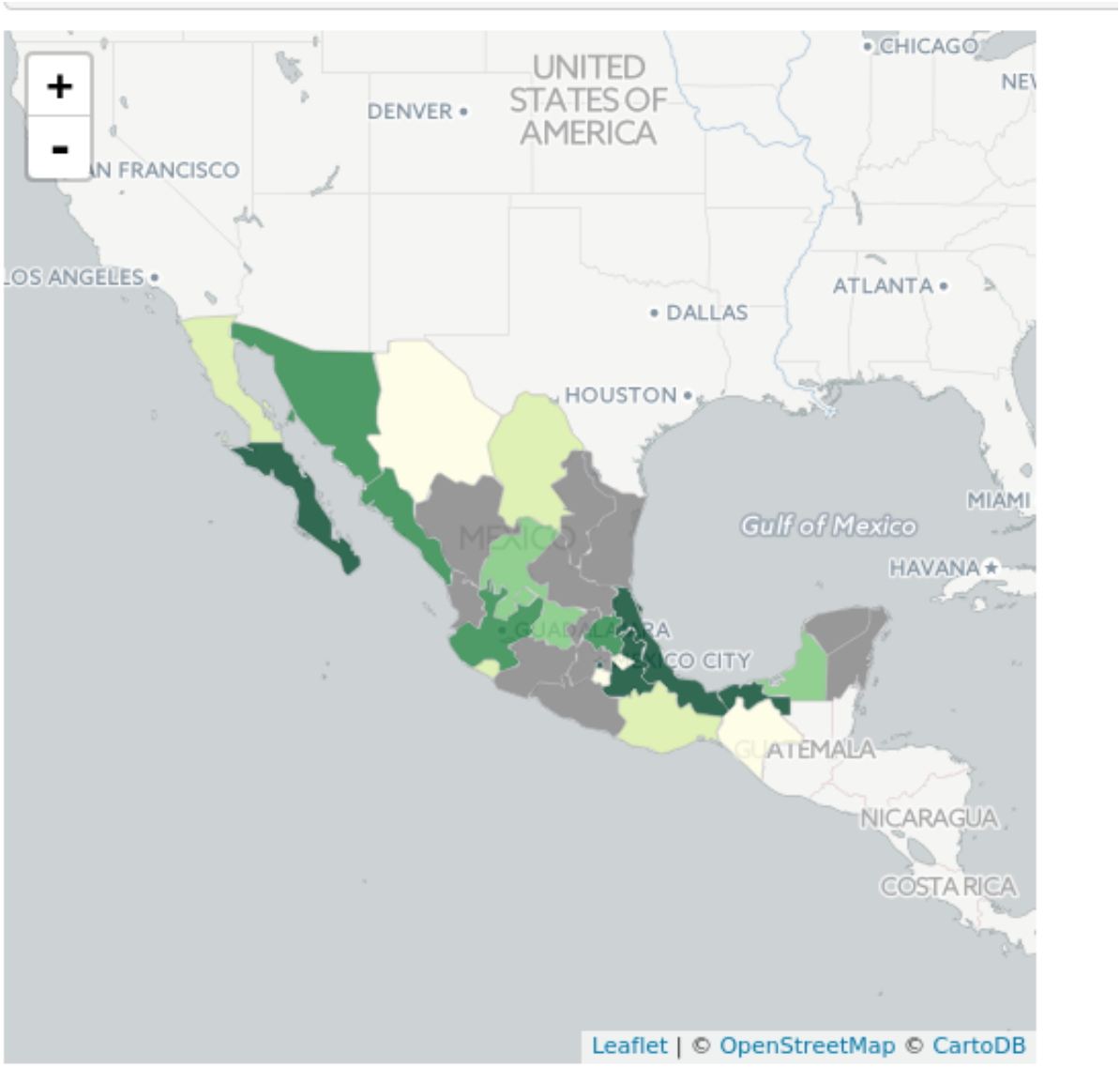
	long <dbl>	lat <dbl>	order <int>	hole <lgl>	piece <fctr>	id <chr>	group <fctr>	Ano <int>
1	-101.8462	22.01176	1	FALSE	1	Aguascalientes	Aguascalientes.1	2014
2	-101.8714	21.98417	2	FALSE	1	Aguascalientes	Aguascalientes.1	2014
3	-101.8886	21.96555	3	FALSE	1	Aguascalientes	Aguascalientes.1	2014
4	-101.9550	21.89444	4	FALSE	1	Aguascalientes	Aguascalientes.1	2014
5	-102.0871	21.76528	5	FALSE	1	Aguascalientes	Aguascalientes.1	2014
6	-102.1492	21.71278	6	FALSE	1	Aguascalientes	Aguascalientes.1	2014

6 rows | 1-9 of 16 columns

Choropleth maps in ggplot2

It's also possible to use the geographic information from the shapefile to get non-interactive maps in `ggplot`.

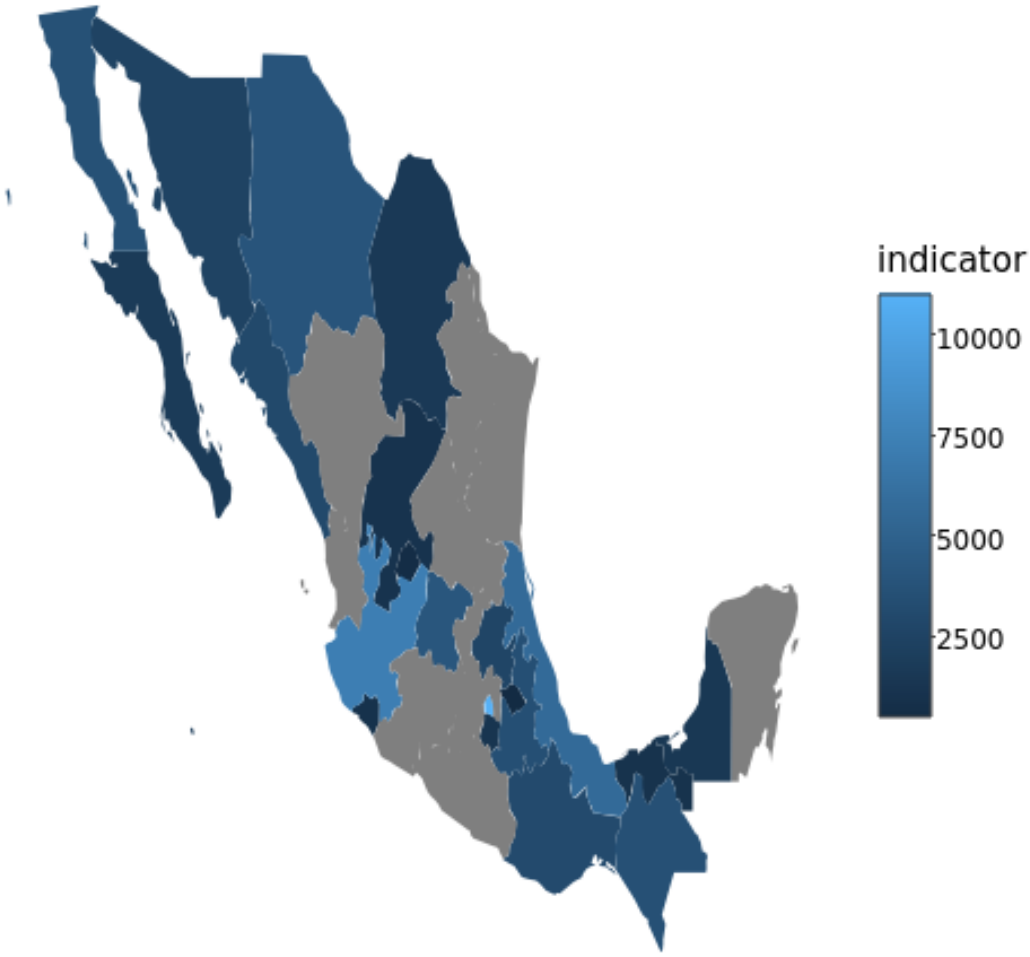
```
library(ggplot2)
library(plotly)
mx <- fortify(mexico, region = "ADMIN_NAME" ) # Convert object to data frame

plotData <- left_join(mx,hotels2014)

head(plotData)
```



And now we can generate the map as any other `ggplot` object!

```
    gg <- plotData %>%
      ggplot(aes(x=long, y=lat, group=group, fill=indicator))+
      geom_polygon()+
      theme(axis.line=element_blank(),axis.text.x=element_blank(),
            axis.text.y=element_blank(),axis.ticks=element_blank(),
            axis.title.x=element_blank(),
            axis.title.y=element_blank(), #legend.position="none",
            panel.background=element_blank(),panel.border=element_blank(),panel.\
grid.major=element_blank(),
            panel.grid.minor=element_blank(),plot.background=element_blank())
```

```
11
12      ggplotly(gg)
```



Chapter 6: Data collection in R

Data collection is a tricky subject, because it might vary a lot on case by case basic. There are two basic ways, either make a request to a website and grab the information you need, or access their application program interface, API, which is a way to ask for specific information to a website.

Whenever you either scrape or use APIs, remember to keep in mind the terms of use of the website! Don't ruin it for the rest of us by sending too many queries.

For this, there are two packages I have found useful for collecting data in R:

- rvest: scraping (so, "borrowing" data from websites) https://github.com/hadley/rvest
- httr: general acces to APIs, do GET/POST request https://cran.r-project.org/web/packages/httr/vignettes/quicl

both packages have Hadley Wickham involved, so they work nicely with other packages from the "Hadleyverse", like ggplot2 and dplyr.

rvest

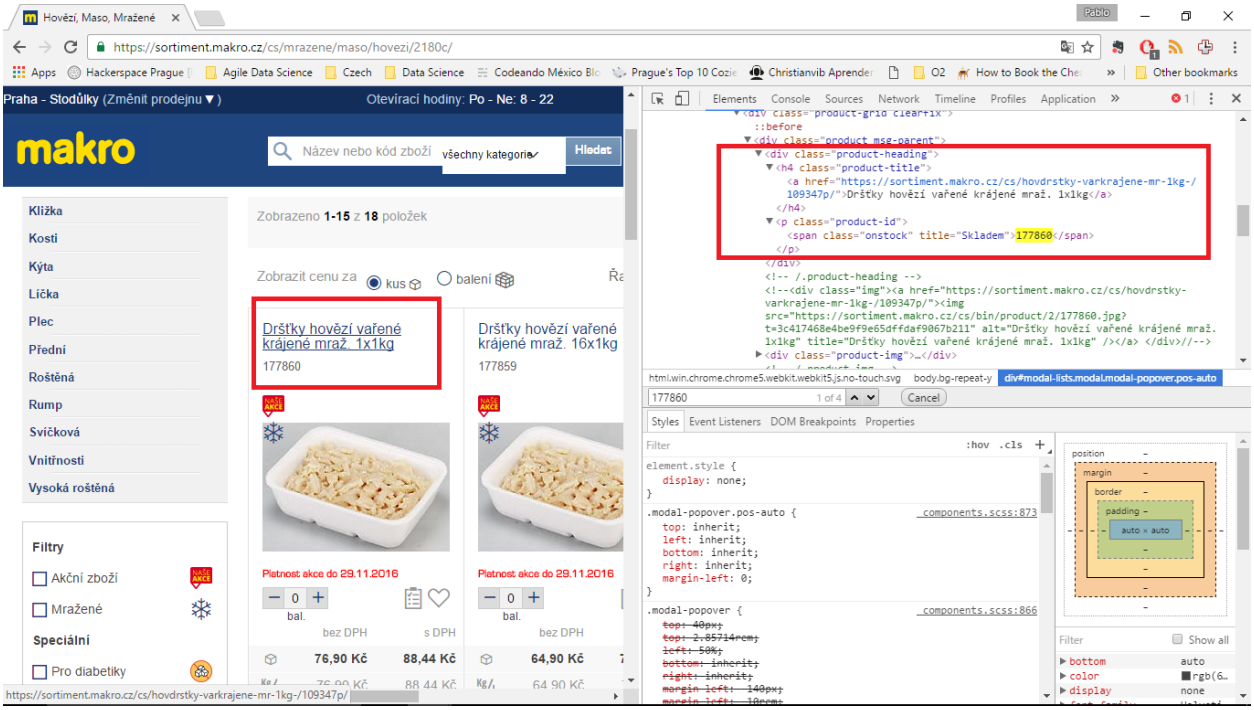
For scraping you need at least a basic knowledge of HTML and CSS.

This is useful to scrape information. When you visualize the source html code from a website with your browser, you need to see where in the HTML code the information you want lies.

Let's say I would like to collect the products shown in the website makro.cz. Here we see a snapshot from Chrome, using Ctrl+Shift+I. I looked for the Product id (177860) to find which element of the HTML has the product name. So I can identify that in the `class="product-heading"`, under the `h4` tag, lies that information. Compare the areas in the red rectangles with the script below and you see quickly the equivalence.

First you load the page with the `read_html` function, and then you go to the specific location using `html_node`, and fetch the information there using `html_text`.

A similar situation happens with the prices, except that in this case sometimes more than one price is shown, so the return object is a list of dataframes. So we need to write a for loop to fetch the information we actually need.




```
1  library(rvest)
2
3  # Seems that you can use any page here
4  page <- read_html("https://sortiment.makro.cz/cs/mrazene/maso/hovezi/2180c/")
5
6
7  products <- page %>%
8              html_nodes(".product-heading h4") %>%
9              html_text()
10
11
12 prices <- page %>%
13           html_nodes(".product-footer table") %>%
14           html_table(header=T)
15
16
17 df <- data.frame(products=products)
18
19 bez_dph <- c()
20 s_dph <- c()
21
22 for(i in 1:length(prices)){
23  bez_dph <- c(bez_dph, prices[[i]][1,2])
```

```
 s_dph <- c(s_dph,prices[[i]][1,3])
}

df$bez_dph <- bez_dph
df$s_dph <- s_dph

View(df)
```

httr

The package `httr` is useful to access APIs

`GET` and `POST` are the two most common "verbs" you can use against an API. You `GET` something when you request stuff, the response is usually in a `JSON` format, and you `POST` when you submit data, for example, if you want to fill in formularies, or you want to submit data to a web service in general.

1) Get an API key from the

2) Install httr.

3) Look at the sample script included for getting information from Mailchimp. `httr` returns the information as an R list, but you have to option to either automatically parse the JSON file, or get a text string and parse it yourself.

Some other APIs have specific packages for them, such as `twitteR` and `Rfacebook`, for Twitter and Facebook, respectively.

```
library(httr)

# on the GET request, X is the number on your API KEY. My api key is something
# like 0a84ebf-us14, so instead of X put 14 here
response <- GET("https://usX.api.mailchimp.com/3.0/lists",
    authenticate(user = "whatever", password = 'YOUR API KEY'))


json <- content(response, "parsed")

# The json content is parsed as an R list, so you can access its elements with t\
he $ sign, as you would do with a dataframe.
```

About the author

Pablo Maldonado is an applied mathematician and data scientist. He has wide experience in mathematical modelling, both in academia and industry. He has collaborated with global organizations in several projects around Big Data, from building predictive models for the online advertising industry to designing and implementing advanced analytical solutions for financial crime, customer risk rating and credit scoring. He holds a Ph.D. degree in applied mathematics (game theory) from the Universite Pierre et Marie Curie in Paris, France. He is fluent in Spanish (native), French, English and has working command of German, Italian and Czech, and aspires to be a less terrible musician (he can not count to 4). You can reach out to him in (www.pablomaldonado.org).